А.Н. ФЛОРЕНСОВ

Методические указания по лабораторным работам дисциплины «ОПЕРАЦИОННЫЕ СИСТЕМЫ»

ОМСК 2017

УДК 681.3

ББК 32.973.73

Ф 73

Составитель: Флоренсов А.Н.

Преамбула

Лабораторные работы по дисциплине «Операционные системы» представляют собой разработку программ, запуск этих программ и взаимодействия программ с пользователем. Текст программ лабораторных работ должен создаваться и модифицироваться студентом непосредственно в учебной аудитории. Тексты программ, разрабатываемых вне аудитории, могут быть только курсовыми работами или домашними заданиями студентов. Не допускается прием в качестве лабораторных работ по данной тематике тех программ по типовым заданиям лабораторных работ, которые были получены вне аудиторного места лабораторных занятий. Причина этого ограничения обусловлена априорной недостоверностью самостоятельных усилий студентов по таким работам ввиду потенциальной возможности копирования чужих программ и разработок. Лабораторные работы выполняются фронтально, так что все обучаемые в лаборатории выполняют одно и то же задание. При защите лабораторных работ могут и даже, как правило, должны использоваться задания преподавателем дополнительных изменений в функционировании программ.

Лишь в порядке исключения наиболее развитым в программировании студентам – по усмотрению преподавателя – допускается разрешать выполнять вне аудитории те задания, которые относятся к следующим за текущей темам, но уже с изменением исходного типового задания. Это разрешение должно быть отражено в лабораторном журнале, а в самом индивидуальном задании лабораторной работы должно присутствовать указание персональной модификации общего задания, также отраженной в лабораторном журнале. Это дополнительное требование отражает объективную особенность работ, выполняемых вне аудитории, – они должны быть персональны и отличаться друг от друга.

Бальное оценивание лабораторных работ - при отсутствии явных оригинальных решений в тексте программ - только на половину общей бальной оценки дается самим фактом формирования и демонстрации преподавателю правильно работающей программы. Вторая половина бальной оценки формируется содержанием защиты лабораторной работы, которое включает быстроту ориентации в демонстрируемом материале, правильные ответы на дополнительные вопросы и модификации функционирования программы по указанию преподавателя.

Задания на лабораторные работы даются в двух вариантах: первый из них содержит минимальное задание, выполнение которого оценивается минимальным числом баллов по расчетному рейтингу на соответствующую работу, второй вариант дает усложненное задание работы, выполнение которого должно обеспечивать получение максимального числа баллов по используемому рейтингу, с учетом, конечно, самостоятельного выполнения и проявленном при защите работы пониманием материала. При затрате студентом на выполнение минимального варианта лабораторной работы более 4 часов аудиторного времени следует переходить к выполнению следующей лабораторной работы, а

возможность выполнения расширенного варианта оставить на конец семестра, если для этого останется учебное время. Это обусловлено необходимостью охвата всех запланированных тем лабораторных работ, иначе могут быть не освоены заключительные темы цикла и к аттестационному оцениванию обучаемый подойдет с тематическими пробелами, что не может не отразиться на итоговой оценке.

Рекомендуется вначале сосредоточить внимание на минимальном варианте, который ориентирован на формирование базовых представлений и опыта по теме работы, затем, после демонстрации преподавателю этого минимального варианта, переходить уже к усложненному варианту. Иной порядок оказывается практически неэффективным и не рекомендуется студентам, поскольку без обратной связи с преподавателем в конце первого этапа может образоваться искаженно-неполное представление об основном предмете работы, которое способно помещать правильному направлению усилий на втором этапе.

В начале лабораторной работы рекомендуется внимательно и не менее двух раз прочитать задание в минимальном варианте, затем открыть конспект лекций или материал учебного пособия по теме работы, бегло просмотреть необходимый материал, или при недостаточном знакомстве через восприятие непосредственно во время лекции или от предварительной подготовке к работе в самостоятельной работе вне аудитории, просмотреть печатный или электронный вариант необходимого учебного материала. После этого можно при желании прочитать задание второй части задания, дающей усложненный вариант, но не следует сразу же на нем концентрировать свои силы.

Лабораторная работа №1.

Цель работы. Ознакомление со средствами стандартного ввода-вывода и их внешнего к исполняемой программе использованию путем переадресации стандартного ввод-вывода из командной строки. Работа относится к общей методической теме изучения взаимодействия выполняемых программ с информацией по их запуску.

Краткие теоретические сведения.

Процесс текстового взаимодействия с пользователем при непосредственном командном общении с операционной системой выполняется в терминах записи и чтения в файл. При этом вывод в текстовую консоль представляется как запись в файл, а ввод — как чтение файла. Файл, из которого осуществляется чтение, называется *стандартным потоком ввода или просто стандартным вводом* (standard input или **stdin**), а в который осуществляется запись — *стандартным потоком вывода* (упрощенно – *стандарным выводом* - standard output или **stdout**). Стандартные потоки — воображаемые файлы, позволяющие осуществлять взаимодействие с пользователем как чтение и запись в файл. Кроме потоков ввода и вывода, существует еще и *стандартный поток ошибок*

(*стандарный вывод ошибок* – standard error или **stderr**), на который выводятся все сообщения об ошибках и те информативные сообщения о ходе работы программы, которые не могут быть выведены в стандартный поток вывода.

По умолчанию входной поток связан с клавиатурой, а выходной поток и поток сообщений об ошибках направлены на терминал пользователя. Другими словами, вся выходная информация запущенной пользователем команды или программы, а также все сообщения об ошибках, выводятся в окно терминала.

Для ввода из или вывода в стандартный поток не требуется открытие файла, поскольку благодаря предусмотрительности разработчиков такие абстрактные файлы открываются самой операционной системой, причем незаметно для пользователей.

Стандартные потоки в большинстве операционных систем, к которым не относятся ОС Windows, связаны с файловыми дескрипторами, имеющими номера 0, 1 и 2. В операционной системе Windows использован более сложный и громозкий для программиста подход. Для получения хэндлов стандартного ввода-вывода следует использовать специальную функцию API с именем GetStdHandle. Функция имеет единственный аргумент, задаваемый одной из системных констант STD_INPUT_HANDLE, STD_OUTPUT_HANDLE, STD_ERROR_HANDLE. Возвращаемое по выполнению этой функции значение и дает реальное текущее значение хэндла, которое далее можно использовать в системных функциях ввода и вывода ReadFile и WriteFile. Сами эти возвращаемые значения сложным образом зависят от текущих обстоятельств.

Стандартный ввод-вывод не обязательно использовать по умолчанию, выдавая с помощью него текст в терминал, а вводя данные только с клавиатуры. Замечательной возможностью является переназначение его составляющих частей на любой обычный файл. Это переназначение выполняется на этапе вызова исполняемой программы, а детали переназначения указываются в командной строке вызова программы. В простейшем и большинстве профессиональных вариантов общения с ОС такой вызов задается с помощью командного интерпретатора непосредственно в командной консоли. Он заключается в том, что в вызов программы, кроме собственно ее имени, включается дополнительный служебный текст со специальным синтаксисом. В простейших случаях такой вызов имеет в командной строке общения с ОС следующий вид:

имя_исполняемой_программы параметры_вызова_программы

Все семантические части такого вызова разделяются пробелами, а не скобками или запятыми, как в большинстве языков программирования.

Для переадресации стандартного ввода и стандартного вывода в командной строке используются вспомогательные символы '<' и '>', соответственно.

Для переадресации стандартного вывода используется запись вида

имя_исполняемой_программы >имя_файла_для_вывода

Для переадресации стандартного ввода используется запись вида

имя_исполняемой_программы <имя_файла_с_вводимыми_данными

Для переадресации стандартного потока ошибок используется запись вида

имя_исполняемой_программы 2>*имя_файла*

Предоставляется также возможность направлять данные об ошибках в стандартный поток обычного вывода. Для этого предназначена запись вида

имя_исполняемой_программы 2>&1

При использовании переназначений с помощью символов обозначений >*имя_файла* предыдущее содержимое файла с именем *имя_файла* теряется (внутренними процедурами файл открывается для записи с усечением до нуля предыдущего содержимого). В ряде случаев, особенно при протоколировании ошибок, может представлять интерес сохранение предыдущего содержимого указанного таким образом файла и приписывание в его конец новых данных. Для решений этой задачи предлагается использовать вместо одного символа '>' два таких символов подряд и без разрыва между ними.

В частности команда

имя_исполняемой_программы <*файл1* >*файл2* 2>>*файл3*

указывает брать исходные данные в файле *файл1*, помещать данные стандартного вывода в файл *файл2* и дописывать в конец файла *файл3* сообщения стандартного потока ошибок текущей команды.

В операционных системах Windows *имя_исполняемой_программы*, при нахождении пользователя, взаимодействующего с ОС именно в том каталоге, где находится запускаемая программа, задается просто ее собственным именем, причем допускается как запись с расширением имени, так и без него. В частности, запуск программы с именем lab1.exe может осуществляться как командной строкой

lab1.exe

так и командной строкой

lab1

В операционных системах Unix/Linux перед собственным именем программы должен явно указываться каталог ее нахождения. В частности, для текущего каталога используется сокращенное обозначение в виде

./собственное_имя_исполняемой_программы

При это символ «точка» обозначает текущий каталог, а разделитель «/» используется как служебный для обозначения файла с учетом имени каталога.

Трансляция программы, написанной на языке Си с помощью свободно распространяемого транслятора GNU задается в простейшем случае командным вызовом

gcc *имя_файла*.c

В результате, если компиляция произошла без ошибок и сформирован исполняемый файл, то этот файл получается с именем a.exe в ОС Windows и с именем a.out в ОС Linux. Получение результирующего файла с другим наименование достигается командным вызовом в форме

gcc *имя_файла*.c -o *имя_исполняемого_файла*

где *имя_исполняемого_файла* для Linux может быть любое, а для Windows должно иметь расширение exe (иметь вид *имя*. exe).

В отличие от непрофессионального использования ОС с помощью щелчков мыши на графических изображениях файлов, профессиональное взаимодействие с операционными системами осуществляется преимущественно в тестовом режиме. Это содержательно полностью соответствует профессиональному общению между людьми, полноценное взаимодействие между которыми осуществляется с помощью естественных языков и специальных текстов, в частности математических. Использование графических образов в общении и специальных жестов характерно скорее для чисто биологических систем, при отсутствии у них «второй сигнальной системы», как термина, введенного русским физиологом И.П. Павловым. Заметим, что мозг животного отвечает лишь на непосредственные зрительные, звуковые и другие раздражения или их следы, возникающие ощущения составляют первую сигнальную систему действительности. Общение исключительно через графический интерфейс и манипуляторы типа «мышь» используют практически только эту древнюю систему.

В процессе эволюции животного мира на этапе становления и начального развития вида Homo sapiens произошло качественное видоизменение системы сигнализации, переход к символам и знакам, имеющим текстовый последовательный характер, появились специализированные языки общения. Частным случаем их являются текстовые языки общения с операционными системами.

В операционных системах типа Linux и Unix действующее значение хэндла стандартного ввода или вывода позволяет внутри выполняемой программы узнать, связан ли этот хэндл с обычным файлом или же с консолью, называвшейся ранее «терминалом». Для этих целей в составе API OS присутствует вспомогательная функция с именем isatty. Эта функция возвращает ненулевое значение, когда указанный в ней аргументом хэндл отвечает терминалу, а не файлу. В частности, проверив на очередном шаге выполнения нулевой хэндл, соответствующий стандартному вводу путем выполнения оператора

if (isatty(0)) *оператор_действий*

можно выполнить те действия, которые программист планирует на случай текущего использования для ввода данных непосредственно клавиатуры.

Задание. Разработать в Windows программу, которая получает хэндлы стандартного ввода и вывода, выводит числовые с комментариями значения этих хэндлов, применяя для этого функцию printf. Затем, используя стандартный ввод и вывод системными функциями небуферизованного ввода-вывода ReadFile и WriteFile, делает приглашение для ввода, вводит любой текст и выводит его с предуведомлением, что он предварительно введен в программу. Продемонстрировать работу программы, запуская ее как с использованием стандартного ввода вывода по умолчанию, так и с переназначением этого ввода на файл для ввода исходных данных и вывода данных вместо экрана. (Базовый вариант задания).

Разработать аналогичную по действиям программу в Linux, обеспечивая в этой системе вывод приглашения на ввод данных со стандартного ввода только в случае использовании ввода с консоли, при переадресации этого ввода на входной файл приглашение отображаться не должно. При переадресации стандартного вывода в файл отображение приглашения в случае ввода с консоли должно принудительно появляться на экране, а не в файле, на который переадресуется вывод. Числовые значения хэндлов стандартных ввода и вывода в этом варианте выводить не нужно. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Где в ходе построения исходной программы, использующей стандартный ввод-вывод, записываются файловые дескрипторы или хэндлы, в каких частях или функциях?
2. Зачем используется переадресация стандартного ввода-вывода ?
3. Сколько существует разновидностей стандартного ввода-вывода?
4. Чем отличается вызов исполняемой программы в командной строке ОС Windows от аналогичного вызова исполняемой программы с тем же именем в командной строке Linux?

Лабораторная работа №2

Цель работы. Изучение связи файлов и управляющих элементов этих файлов, связанных с операционной системой.

Краткие теоретические сведения. При работе программы с файлом нужно где-то поддерживать рабочие параметры этой работы. Особенно очевидна необходимость вести учет до какого места в файле он прочитан. Самые ранние реализации компьютерных файлов использовали в качестве хранилища данных (*носителя данных* – в терминологии 60-х годов XX в.) магнитные ленты. Тогда фиксация места, где остановилось чтение или запись в такой файл, осуществлялась физически – механической остановкой устройства. Позже реализация файлов была качественно и количественно усовершенствована, но логически-временная последовательность действий как существо работы с файлами, была сохранена. Отчасти в силу преемственности, но в большей степени по прагматике смысловых действий, поскольку это соответствует привычным человеческим действиям над бумажными носителями информации. Кроме того, отказ от сложившейся традиции немедленно приводил к необходимости при каждой операции над файлом явно указывать с какого численно место далее читать или писать. Причем принятие такого усложняющего требования невольно должно было порождать большое количество ошибок программиста по указанию места, как не привычное, однообразное и сложное для контроля.

Поэтому более-менее совершенные реализации инструментов работы с файловой системой стали включать специальные структуры данных для работы

с последовательно используемым файлом, которые в свое время получили название *file control block* – управляющий блок файла. В данном контексте слово *control* в одном из его значений на английском языке обозначает именно контроль, контролирование и учет, а не собственно действия по принудительному управлению.

В блоке управления файлом важнейшим элементом (полем структуры данных) является именно порядковый номер байта в файле, с которого реально будет выполняться следующая по времени операция чтения или записи. Эта величина называется текущей позицией ввода-вывода в файле (и иногда называется смещением – offset).

Для того, чтобы начать читать (или писать внутри уже записанного файла) не с достигнутого ранее места, достаточно принудительно установить эту позицию с помощью специально предназначенных для этого функций. В ОС Linux основной из этих функций является функция lseek. Функция имеет три аргумента, в первом указывается файловый дескриптор (хэндл), второй задает величину используемого в ней смещения, последний обозначает точку отсчета для этого смещения – от начала файла, от предыдущего позиции или от конца файла.

Сам файловый дескриптор должен быть получен от функции открытия файла, которая в Linux называется open. В качестве первого аргумента ее задается имя файла в файловой системе, т.е. в общем случае с учетом каталогов, через которые организуется доступ к этому файлу. В простейших случаях достаточно собственного имени файла, если он находится в текущем каталоге. Вторым аргументом задаются опции, которые указывают режим работы с файлом – открытие для чтении или записи (в простейших случаях). Целый ряд служебных констант для этого аргумента позволяет создавать файл и задавать специальные настройки или особенности использования. Полученный как результат успешного выполнения open файловый дескриптор, который на элементарном уровне информации представляет собой просто некоторое беззнаковое число, используется в дальнейшим почти для всех других системных функций, оперирующих с файлами.

Замечательным свойством использования внутренней для ОС связи дескрипторов и хэндлов файлами оказывается гибкая возможность связывания числового номера хэндла (файлового дескриптора) с управляющим блоком открытого файла.

Имея уже открытый файл, а следовательно введенный в использование его управляющий блок, можно получить другой номер файлового дескриптора, которым можно пользоваться как средством доступа к тому же управляющему блоку, а следовательно, и к тому же файлу.

Для этого в Unix/Linux служат системные функции **dup** и **dup2.** Они создают действующую файловый дескриптор к тому же файлу, точнее к тому же управляющему блоку файла. Функция dup() просто предоставляет другой номер для

доступа к тому же управляющему блоку. Причем, как второстепенное свойство, ОС выделяет для этого минимально доступный (не занятый) номер. После такого действия операции с файлом могут осуществляться как с помощью старого, так и нового файлового дескриптора.

Функция dup2 с прототипом int dup2(int old_handle, int new_handle) предоставляет возможность использовать желаемый номер в качестве файлового дескриптора уже открытого файла. Именно, первый параметр этой функции задает действующий файловый дескриптор, а второй параметр должен быть численно равен желаемому номеру. Обе функции при невозможности выполнения возвращают служебное значение, равное -1, что является признаком ошибки.

Задание. Результат выполнения лабораторной работы должен состоять из двух программ для Linux. Первая программа должна создавать текстовый файл, вводя данные со стандартного ввода. (Более детально: открывает файл для записи, читает текст со стандартного ввода и выводит этот прочитанный текст в файл; в получаемом файле должны после этого находиться только те данные, которые были только что введены со стандартного ввода.) Вторая программа открывает тот же файл (созданный перед этим другой программой) для чтения и хэндл, полученный при этом открытии, запоминает в 1-й переменной для хэндла. Используя этот хэндл, далее с помощью функции dup() получается новое значение хэндла для доступа к тому же файлу (2-й хэндл). Еще раз открывается тот же файл, запоминая 3-е значение хэндла. После получения действующих значений всех трех хэндлов программа позиционирует чтение для 10-й позиции файла от начала этого файла, используя при этом первый хэндл. Далее программа должна выводить числовые значения всех трех хэндлов на экран. Используя по очереди все 3 хэндла, из файла читаются по 7 символов и тут же эти три прочитанных текста выводятся на экран, каждая в своей строке. Результаты вывода объяснить.(Базовый вариант)

Выполнить то же задание для операционной системы Windows, используя для дублирования хэндла файла системную функцию DublicateHandle, вместо dup в Linux. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Чем открытый файл в операционной системе отличается от неоткрытого, обосновать ответ.
2. Зачем закрывается файл, что происходит, если программист не указал явную функцию закрытия файла?
3. Что содержится в управляющем блоке файла?
4. Где относительно остальной части программы находится управляющий блок файла, используемый в этой программе?

Лабораторная работа №3

Цель работы. Изучение особенностей использования файлов в многопрограммной операционной системе в Windows.

Краткие теоретические сведения. В ОС Windows отсутствует единая система информирования об ошибке выполнения системной функции. Это связано как с традиционной многочисленностью внутрифирменного коллектива разработчиков, так и отсутствие действительно хорошо продуманной идеологии операционной системы. Различные системные функции возвращают в качестве собственных значений различные типы данных. Краткий перечень возникающих при этом ситуаций перечислен в табл. 1.

Таблица 1. Стандартные типы значений, возвращаемых функциями Windows

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно. Таких функций в Windows очень мало.
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0. (Не пытайтесь проверять его на соответствие TRUE или FALSE)
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL, в остальных случаях HANDLE идентифицирует объект, которым Вы можете манипулировать. Будьте осторожны: при ошибке некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным -1. В документации для каждой функции четко указывается, что именно она возвращает при ошибке — NULL или INVALID_HANDLE_VALUE.
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL, в остальных случаях PVOID сообщает адрес блока данных в памяти.
LONG/DWORD	Это значение — "крепкий орешек". Функции, которые сообщают значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что Вы хотели, она обычно возвращает 0 или -1 (все зависит от конкретной функции). Если Вы используете одну из таких функций, проверьте по документации, каким именно значением она уведомляет об ошибке.

При обнаружении, что при выполнении системной функции произошла ошибка, следует далее детально установить причину этой ошибки. Для этих целей предназначена специальная функция GetLastError(), которая не имеет аргументов и возвращает значение типа DWORD. В интегрированных средах разработки программ, в частности, Visual Studio, инструментальная среда берет на себя заботу по информированию разработчика о ситуации при запуске исполняемой программы, когда она запускается под управлением интегрированной среды, в частности сообщает текстом описание причины ошибки. При использовании исполняемых программ вне подобной среды действия по информированию разработчика и пользователя должно быть вынужденно возложено на

саму исполняемую программу. Для этого следует возвращаемое значение от GetLastError() сравнивать с теми или иными документированными значениями кодов ошибок, которые в среде разработки находятся в заголовочном файле winerror.h. Для относительного удобства пользователя по преобразованию числового кода ошибки в содержательный текст предназначена служебная функция FormatMessage. Использование этой функции не всегда удобно из-за относительной сложности и перегруженности ее вспомогательными аргументами, которых начинающий программист нередко склонен избегать.

С учетом того, что современная ОС может одновременно выполнять множество различных программ, а некоторые из них при этом могут использовать или пытаться использовать одни и те же файлы, возникла проблема одновременного доступа к одним и тем же файлам со стороны одновременно и нередко несогласовано работающих программ. Ситуация, когда более чем одна одновременно выполняемая программа пишет данные в один и тот же файл, по своему поведению подобна действиям двух и более преподавателей одновременно записывающих свой материал на одной лекционной доске. Очевидно, что такая ситуация неизбежно должна приводить к перемешиванию данных от различных источников записи в непредусмотренном записывающими порядке. Как следствие, недопустимо разрешать такую возможность без специального дополнительного управления и, в простейших или большинстве случаев не допускать саму подобную ситуацию.

Решение подобных проблем возлагается на операционную систему с помощью специальных встроенных в нее средств. Простейшими из них являются средства ограничения одновременного использования файлов.

В Windows, с учетом большого опыта и накопленного осмысления проблем, имеется два подхода к их решению. Во-первых, в функцию открытия доступа к файлу вложены дополнительные возможности ограничения доступа ко всему файлу другим программам или других экземплярам запуска той же программы, одновременно пытающимся работать с этим файлом. Для этого в функции CreateFile предназначен третий аргумент. В нем с помощью системных констант FILE_SHARE_READ и FILE_SHARE_WRITE можно задать разрешение другим вычислительным процессам на выполнение той или иной операции с тем же файлом. Задание обоих констант путем объединения их кодов с помощью операции ИЛИ в том же аргументе позволяет разрешить и чтении, и запись (что, как правило, не имеет смысла, поскольку возникает риск не обусловленности результата одновременных операций). Если задать в этом параметре нулевое значение, то никаких разрешений не одновременную работу не действует и файл, если он успешно будет открыт текущей программой, становится недоступным другим процессам, пытающимся в дальнейшем одновременно работать с ним. Запрет исчезает после прекращения связи с файлом текущего процесса, что равнозначно выполнению операции закрытия файла этим процессом.

Вторым вариантом ограничения доступа к одновременному использованию файла служат специальные функции ограничения доступа к участку файла. Применение их целесообразно, когда желательно ограничить доступ не ко всему файлу, а только к некоторому его участку. Основными системными функциями для этого служат LockFile и UnlockFile. Первая из них предназначена для установки блокировки на участок файла, вторая – для снятия блокировки с подобного участка. При этом функция LockFile имеет пять аргументов, причем для задания 64-битных характеристик для файлов в современных версиях используются раздельно аргументы для старшей и младшей половины таких характеристик. Пара из второго и третьего параметров задает начало участка для установки блокировки (так называемое *смещение* в файле), последние два параметра задают число байтов в таком участке. Подробности об приемах использования такого подхода следует извлекать из документации по системным функциям. При этот блокировка устанавливается без учета особенностей предполагаемых действий, не детализируя для чтения или записи.

Особенностью функции LockFile является то, что при невозможности для текущего вычислительного процесса установить блокировку на задаваемый участок, когда он уже захвачен в использование другим процессом, функции возвращает значение 0, как смысловое значение типа BOOL. Действие ожидания освобождения участка этой функции не свойственно. Поэтому в практическом программировании такое ожидание, если оно необходимо по смыслу программируемых действий, разработчику следует задавать и организовывать другими доступными ему средствами.

В последующих за ранними версиями Windows были дополнительно введены расширенные функции выборочной блокировки LockFileEx и UnlockFileEx, имеющих еще более сложное строение по организации аргументов, но позволяющих учесть варианты чтения или записи в участок файла.

Последующая дополнительная информация предназначена только для дополнительной части задания с целью получения максимального рейтинга за лабораторную работу.

Для управления блокировкой открытого файла в Linux может быть использована функция fcntl, которая для этого применяется в форме со следующим набором параметров

fcntl(int handle, int cmd, struct flock *lock);

Использование функции требует заголовочного файла fcnt.h, задаваемого в директиве include.

Второй аргумент этой функции константами F_SETLK и F_SETLKW задает действия по установке блокировки на участок файла. Третий аргумент является указателем на структуру, которая описывает участок файла. В этой структуре с типом flock присутствуют поля l_type, l_whence, l_start, l_len. Поле l_type задает тип блокировки и может задаваться константами F_RDLCK, F_WRLCK, F_UNLCK. Первая из этих констант задает «блокировку по чтению

для текущего процесса», вторая – «блокировку по записи для текущего процесса». Последняя константа задает «снятие блокировки на указанный участок». Поле l_start дает «начальное смещение для блокировки», а поле l_whence – как интерпретировать это смещение. При использовании константы SEEK_SET смещение отсчитывается от начала файла, при константе SEEK_CUR – от текущей позиции в файле, а при SEEK_END – отсчет смещения ведется с конца файла. Поле l_len задает количество байт для блокировки.

При использовании команды F_SETLK и задании в поле l_type типов F_RDLCK или F_WRLCK делается попытка установить блокировку на указанный участок. Если это удается, функция возвращает нулевое значение, если попытка неудачна, то функция возвращает значение -1, которое нужно обязательно проверять, поскольку на момент составления программы реальная ситуация не может быть предопределена. Если же использовать команду F_SETLKW, то при невозможности при выполнении функции установить блокировку, она переходит в режим ожидания (что и отражается последним символом константы – от слова Wait). Этот вариант во многих случаях предпочтительней для программиста, так как никаких дополнительных усилий по организации ожидания освобождения требуемого участка файла от него не требует. По ситуации невозможности установить блокировку требуется применить альтернативные варианты действий или выдать сообщение о временной невозможности доступа. Тогда следует использовать вариант функции с командой F_SETLK.

Задание. Разработать программу для Windows, которая должна запускаться в двух экземплярах - каждый в своем окне командной оболочки FAR или из ПРОВОДНИКа операционной системы. Программа использует заранее подготовленный текстовый файл. Она пытается открыть этот файл для чтения с указываемым при этом запрете для других использовать этот файл. По результатам выполнения системной функции открытия на экран выдается сообщение, если не удалось открыть файл и причину этой неудачи, когда файл не существует или уже используется другим запуском программы. При отсутствии указанного в программе файла после сообщения об этом отсутствии программа прекращает работу. При его наличии, но невозможности продолжения действий из-за блокировки, установленной другим экземпляром запущенной программы, выполняется ожидание освобождения файла от блокировки. Как только программа получает возможность работать по чтению с файлом (либо сразу, либо после ожидания), она читает из этого файла все находящиеся в нем данные и выводит их на экран. Сообщения должны выводиться цветные и в середине консольного окна. После вывода текста из файла программа должна ждать 7 секунд до своего завершения, удерживая тем самым занятым используемый файл. (Базовый вариант.)

Разработать аналогичную по поведению программу в ОС Linux, которая использует блокировку файла средствами этой системы и не позволяет одновре-

менно работать с ним более чем одному экземпляру запущенной программы. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Чем стандартный ввод-вывод отличается от нестандартного, привести примеры использования того и другого.
2. Зачем в общем случае закрывать файл, к каким возможным последствиям и когда приводит не закрытие файла.
3. Какой код ошибки возвращает функция открытия файла, если в ней задано ограничение по одновременной работе с файлом и этот файл в момент выполнения попытки открытия уже захвачен для работы другим вычислительным процессом.

Лабораторная работа №4

Цель работы. Изучение средств событийного программирования и особенностей взаимодействия в текстовом режиме ОС Windows.

Краткие теоретические сведения.

Программы, работающие в современных ОС, могут получать воздействия от различных типов внешних событий. Наиболее широко используемыми типами событий являются нажатия клавиши клавиатуры, от мыши, сообщения об изменении размера окна, о переходе активности к текущему окну или о потере такой активности. Свойство активности визуально отражается изменением цвета заголовка окна и содержательно состоит в том, что только активное окно получает данные от клавиатуры.

В Windows программа для текстового окна может запросить сообщение путем вызова системной функции ReadConsoleInput. Эта функция имеет прототип

```
BOOL  ReadConsoleInput(HANDLE hConsInput,
            INPUT_RECORD* buffer, DWORD len, DWORD* actlen).
```

Кроме хэндла для указания консольного буфера ввода (в частности хэндла стандартного файла ввода) эта функция содержит адрес буфера, который представляет собой в общем случае массив записей типа INPUT_RECORD для размещения некоторого числа записей сообщений ввода. Размер массива выбирается программистом. Размер этого массива записей задается при вызове в параметре *len*. В простейших случаях, массив *buffer* состоит из единственного элемента - для размещения единственного очередного сообщения и параметр *len* берется поэтому равным 1. В общем случае, когда при вызове функции задается *len* не равное 1, следует в программе после обращения к ReadConsoleInput проверять сколько записей о вводе было действительно получено (с помощью возвращаемого параметра *actlen*). И принимать соответствующие действия с учетом этого фактического значения. Заметим, что функция ReadConsoleInput

возвращает управление в вызвавшую ее программу только после появления сообщения о вводе. До этого момент вычислительный процесс выполнения программы, содержащей такую функцию, приостановлен (блокирован).

Сообщения как структуры данных типа INPUT_RECORD, получаемые от этой функции, имеют довольно сложное строение. Это строение описывается в заголовочном файле следующим образом:

```
typedef struct _INPUT_RECORD {
   WORD EventType;
   union {
      KEY_EVENT_RECORD KeyEvent;
      MOUSE_EVENT_RECORD MouseEvent;
      WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
      MENU_EVENT_RECORD MenuEvent;
      FOCUS_EVENT_RECORD FocusEvent;
   } Event;
} INPUT_RECORD, *PINPUT_RECORD;
```

Ключевым полем в этой записи является тип события EventType, его значения задаются предопределенными константами, описанными как

```
//  EventType flags:
#define KEY_EVENT         0x0001 // Event contains key event record
#define MOUSE_EVENT       0x0002 // Event contains mouse event record
#define WINDOW_BUFFER_SIZE_EVENT 0x0004
                          // Event contains window change event record
#define MENU_EVENT 0x0008 // Event contains menu event record
#define FOCUS_EVENT 0x0010 // event contains focus change
```

Это ключевое поле своим значением определяет более детальное строение сообщения. Если значение типа равно KEY_EVENT, но на самом деле в этой универсальной структуре вся остальная часть (обобщенно называемая Event) есть структура типа KEY_EVENT_RECORD. Если же значение типа равно MOUSE_EVENT, то остальная часть есть в действительности структура типа MOUSE_EVENT_RECORD. (Остальные типы сообщений и их структуры рассматриваться не будут.)

Поэтому типовое использование системной функции ReadConsoleInput может быть описано наиболее характерной схемой вида

```
. . .
ReadConsoleInput( hInput, &inpbuf, 1, &actlen);
if (inpbuf. EventType == KEY_EVENT)
  {обработка события от клавиатуры,
    структура которого представляется в программе обозначением
    ipnbuf.Event.KeyEvent }
```

```
if (inpbuf. EventType == MOUSE_EVENT)
  {обработка события от клавиатуры,
    структура которого представляется в программе обозначением
    ipnbuf.Event.MouseEvent }
. . .
```

в которой предполагается, что использованные информационные объекты данных где-то раньше описаны как

```
HANDLE hInput;
INPUT_RECORD inpbuf;
DWORD actlen;
```

Дополнительные возможности в Windows предоставляются путем выборочного задания типов событий, которые будут далее поступать при вызове функции ReadConsoleInput. Этот выбор задается функцией SetConsoleMode с прототипом

```
BOOL SetConsoleMode(HANDLE hConsHandle, DWORD mode);
```

Параметр *mode* задает в ней новый режим управления вводом (или выводом, если в качестве первого параметра *hConsHandle* задан хэндл буфера консольного вывода). Возможные режимы для буфера ввода с консоли задаются константами

```
#define ENABLE_PROCESSED_INPUT      0x0001
#define ENABLE_LINE_INPUT           0x0002
#define ENABLE_ECHO_INPUT           0x0004
#define ENABLE_WINDOW_INPUT         0x0008
#define ENABLE_MOUSE_INPUT          0x0010
```

Если с помощью функции SetConsoleMode задать в качестве режима значение константы ENABLE_MOUSE_INPUT, а другие константы при этом задании режима не использовать, то по запросу событий будут поступать из управляемых функцией SetConsoleMode только сообщения от мыши.

Отключив режим ENABLE_ECHO_INPUT мы откажемся от использования эха символов, но делать это следует временно - с последующим восстановлением стандартного режима. Поэтому вначале целесообразно запомнить предыдущее значение режима для буфера ввода консоли. Для этого действия предназначена функция

```
BOOL  GetConsoleMode(HANDLE hConsHandle, DWORD* pmode),
```

в которой возвращаемое значение текущего режима передается через второй параметр ее вызова.

Режим консоли, задаваемый константой ENABLE_PROCESSED_INPUT, указанной выше, позволяют при необходимости для пользователя прекращать выполнение программы в консольном окне с помощью управляющего сигнала, который порождается нажатием клавиши C с одновременно нажатой и удерживаемой клавишей Ctrl. Это так называемый управляющий сигнал Ctrl-C. Поскольку такая возможность очень удобна для пользователя и разработчика, сле-

дует без серьезной необходимости не отказываться от нее. Следствием такого заключения оказывается целесообразность задания для работы с мышью не только константы ENABLE_MOUSE_INPUT при вызове функции SetConsoleMode, но и указанной константы ENABLE_PROCESSED_INPUT. С учетом того, что они задают различные биты многоразрядного управляющего кода режима, для их одновременного задания используется операция ИЛИ, которая записывается между этими двумя константами для формирования действующего значения второго параметра функции.

События от клавиатуры поступают в текстовую консоль всегда, их нельзя запретить или ограничить.

Другим и более совершенным вариантом решения указанной проблемы является предварительное получение текущих режимов консоли с помощью функции GetConsoleMode и использование полученного кода в виде записи

полученный_код | ENABLE_MOUSE_INPUT

во втором аргументе функции SetConsoleMode. Это в общем случае позволяет учесть действие всех предыдущих режимов при такой установке.

Графический режим ОС Windows требует от программиста указания множества деталей при его задействовании в программе. Эта сложность скрывается в многих современных средствах разработки и языках программирования путем включения в действующий код исполняемой программы объемных и непростых компонентов, незаметных для большинства начинающих или посредственных программистов. Особенно это глубоко оказывается скрытым в объектно-ориентированном программировании. Изучение операционных систем для профессионального владения и даже знакомства программистам требует сознательно понимания хотя бы ведущих моментов внутренних программных особенностей текстового и графического режима.

Задание. Программа для Windows должна открыть текстовый файл для чтения и все его содержимое вывести в текстовую консоль (текстовое окно на экране). Размер этого текстового файла условием задачи не ограничен (может быть любой). Затем в программе запускается опрос событий текстового режим для консоли. Работающая программа должна сообщать, какое событие произошло. Как минимум должна быть реакция на события от мыши и клавиатуры. Информация о щелчке мыши выводится, начиная с той позиции текстового окна, на которой находится курсор мыши во время щелчка. Тут же выводится номер позиции в виде двух чисел — номера строки и номера столбца этой позиции. По событию от клавиатуры выводится сообщение о нажатой клавиши. Для алфавитно-цифровых клавиш выдается символ клавиши, а для управляющих их обозначение. Достаточно выдавать детальное обозначение только двух-трех управляющих клавиш. Действия программы с указанными реакциями на события могут осуществляться многократно. Завершение программы осуществляется по нажатию правой клавиши мыши. (Базовый вариант)

Усложненный вариант для максимального рейтинга требует по щелчку мышью на любом слове, находящемся на экране вместо информации о месте щелчка отобразить символ, над которым был щелчок, изменив «регистр» символа (строчные буквы заменить на прописные и наоборот), одновременно перекрашивая указанные мышью символы в яркий цвет (ярко-красный или желтый).

Контрольные вопросы.

1. Чем событийное программирование отличается от жестко алгоритмического, опишите сферу применений того и другого?
2. Чем отличается программная обработка нажатий на управляющую клавишу от аналогичной обработки нажатий на алфавитно-цифровую клавишу при использовании функции ReadConsoleInput в Windows?
3. Почему в современных ОС для получения информации от мыши в прикладных программах используется событийно организованное программирование, а для получения информации от клавиатуры - не всегда?

Лабораторная работа №5

Цель работы. Изучение средств событийного программирования и особенностей взаимодействий в графическом режиме ОС Windows.

Краткие теоретические сведения.

Графический режим во взаимодействиях программы поддерживается в специальных окнах, называемых графическими (если они профессионально явно называются, поскольку непрофессионалы неявно могут считать все окна графическими). Для их создания используется функция CreateWindow. Но в связи со сложной историей становления и развития графических приложений требуются дополнительные функции и действия по формированию параметров для CreateWindow.

В первых графических разработках Microsoft, которые стали фундаментом дальнейшего ПО этого производителя, были приняты специальные меры обеспечения быстрой взаимодействия с компьютером даже для простых и не очень быстрых аппаратных оснований (платформ). Это решение для программистов на уровне непосредственно API вылилось в использование специальных предварительных описаний классов окон для однотипных графических окон и специальных процедур обслуживания окна отдельных от основной программы.

Поэтому общая схема программных действий для подготовки и запуска графического окна имеет следующую последовательность действий:

- получение уникального идентификатора исполняемой программы;
- определение класса описания однотипных окон с общей процедурой обслуживания;
- регистрация класса окон;

- создание окна как структуры данных;
- отображение окна;
- цикл получения и пересылки сообщений.

Некоторые из перечисленных этапов действий осуществляются вызовами более чем одной соответствующей системной функции и требуют последовательности операторов реализации.

Более детально обсуждаемые этапы подготовки функционирования графического окна описываются следующим упрощенным фрагментом программной схемы

```
#include <windows.h>
int main()
 {HINSTANCE hInstance;
  HWND hwnd; // хэндл графического окна
  MSG msg;  // экземпляр структуры данных сообщения
  WNDCLASS ws; // экземпляр класса окон
  STARTUPINFO si;
  int nCmdShow;
 .. //получение идентификатора исполняемой программы и
 // целочисленного флага варианта отображения (первый этап)
 . . . // заполнение и вычисление полей структуры ws (второй этап).
 RegisterClass(&ws);              // (третий этап).
 hwnd=CreateWindow( . . .);            // (четвертых этап).
 . . . // сюда вставить проверку hwnd на "не нуль"
 ShowWindow(hwnd, nCmdShow);   // (пятый этап).
 while(GetMessage(&msg, 0, 0, 0)) // (шестой этап).
     DispatchMessage(&msg);
 return 0;
 }
```

Еще более детализированное до собственно исходного кода задание рассматриваемых действий дается следующим кодом

```
#include <windows.h>
{HINSTANCE hInstance;
  STARTUPINFO si;
  int nCmdShow;
 HWND hwnd; // хэндл графического окна
 MSG msg;  // экземпляр структуры данных сообщения
```

```
WNDCLASS ws; // экземпляр класса окон
LRESULT WINAPI  WinProc(HWND, UINT, WPARAM, LPARAM);

 GetStartupInfo(&si);
 if (si.dwFlags & STARTF_USESHOWWINDOW)
  nCmdShow=si.wShowWindow;
 else nCmdShow=SW_SHOWDEFAULT;
 hInstance = GetModuleHandle (NULL); // конец 1-го этапа

  memset(&wc, 0, sizeof(wc));
  wc.lpszClassName="MyClass";
  wc.lpfnWndProc= WinProc;
  wc.hCursor=LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground=(HBRUSH)(COLOR_WINDOW+1);
  wc.hInstance= hInstance;   // конец 2-го этапа
  if (!RegisterClass(&wc)) return; // 3-й этап


  hwnd=CreateWindow("MyClass", "Our Window",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,CW_USEDEFAULT,
                CW_USEDEFAULT,CW_USEDEFAULT,
                0, 0, hInstance, NULL); // 4-й этап
   if (!hwnd) return;
   ShowWindow(hwnd, nCmdShow); // 5-й этап

   while (GetMessage(&msg,0,0,0)) // 6-й этап
     DispatchMessage(&msg);
}
```

Реакции графического окна на поступающие к нему события описывается специальной процедурой, имя которой должно быть использовано в поле lpfnWndProc перед регистрацией структуры класса окон. В нашем примере это имя было обозначено как WinProc. Сама эта процедура очень специализирована и, в частности, должна иметь четыре строго типизированных параметра.

В качестве примера обеспечения вывода текста "Pressed left botton of mouse" в позицию окна (100,50) можно использовать следующий код. Для реакции текстом сообщения "Pressed left botton of mouse" на щелчок мыши над графическим окном эта процедура может задаваться следующим кодом

```
LRESULT WINAPI WinProc(HWND hwnd, UINT tmsg,
                       WPARAM wParam, LPARAM lParam)
{HDC hdc;
 switch(tmsg)
 {case WM_LBUTTONDOWN:
     {hdc=GetDC(hwnd);
      TextOut(hdc, 100 /* координата X в пикселах*/,
       50 /* координата Y */,
      "Pressed left botton of mouse", 28 /* длина текста*/);
      ReleaseDC(hwnd, hdc);
      return 0;
     }
  case WM_DESTROY:
    {PostQuitMessage(0);
     return 0;
    }
 }
 return DefWindowProc(hwnd, tmsg, wParam, lParam);
}
```

Трансляция подобной программы для графического приложения должна выполнятся с опцией -mwindows, так что трансляция программы с исходным кодов в файле prog.c должна задаваться вызовом

gcc -mwindows prog.c

В приведенном примере каждое нажимание на левую кнопку мыши вызывает вывод информирующего текста, начиная с одного и того же фиксированного места, а именно позиции (100, 50). Для вывода с учетом места конкретного воздействия следует воспользоваться информацией из сообщения от мыши. Эта информация, как сохранилось от очень старых теперь конструктивных решений, передается в специализированную процедуру не через экземпляр структуры данных, а через 3-й и 4-й практически не типизированные параметры этой процедуры. По существу в эти параметры общего назначения сама ОС заносит специфические для сообщения параметры из структуры, используемой в функции DispatchMessage.

Так для сообщения от клавиш мыши координаты воздействия, представляющие собой числовые значения пиксельных координат, заносятся и передаются через последний параметр. Причем оба они как машинные 16-битные коды складываются в общий 32-битный код. Фактически выполняется операция, задаваемая на языке Си в виде записи

(LPARAM)((координата_Y <<16)|(координата_X))

Использование же значений переданных через параметр процедуры координатпредполагает применение специально предназначенных для этого макросов LOWORD и HIWORD. Так для вывода текста о сообщении не в фиксированном месте окна, а точно с того места, над которым произошел щелчек, может быть использован оператор

```
TextOut(hdc, LOWORD(lParam),  HIWORD(lParam),
 "Pressed left botton", 19);
```

Для получения сообщений от любой клавиши используется тип сообщения WM_KEYDOWN, при котором в параметре wParam поступает виртуальный код клавиши. Если требуется получать символьный код от алфавитно-цифровой клавиши, то следует использовать разветвление по типу WM_CHAR, при котором в том же параметре wParam поступает сам код символа. Но по историческим особенностям формирования графической системы Windows, этот тип сообщения формируется только тогда, когда в главном цикле обработки сообщений после вызова функции DispatchMessage(&msg) вызывается функция ; TranslateMessage(&msg). Так что цикл обработки сообщений должен строить в виде

```
while (GetMessage(&msg,0,0,0)) // 6-й этап
   {TranslateMessage(&msg); DispatchMessage(&msg);  }
```

Описание всех использованных в примере функций следует смотреть и изучать по программной документации Windows, в частности с помощью справочной системы win32.hlp, и использовать более подробное описание необходимых функций и параметров графического режима по учебным пособиям [9,10].

Задание. Разработать программа для Windows, которая должна создавать графическое окно стандартного размера и реагировать на события для нее. Работающая программа должна сообщать, какое событие произошло. Как минимум должна быть реакция на события от мыши и клавиатуры. Информация о щелчке мыши выводится, начиная с той позиции графического окна, на которой находится курсор мыши во время щелчка. Тут же выводится номер позиции в виде двух чисел — номера строки и номера столбца этой позиции. По событию от клавиатуры для алфавитно-цифровых клавиш в графическое окно выдается сообщение о нажатой клавиши и символ клавиши. Завершение программы осуществляется с помощью традиционных средств графического окна. (Базовый вариант)

Усложненный вариант для максимального рейтинга требует при изменении размера окна выдавать в окно текст, сообщающий новый размер в виде числового значения ширины и длины, выраженных числом пикселов.

Контрольные вопросы.

1. Что общего в системных функциях ReadConsoleInput и GetMessage, в чем они по существу отличаются?

2. Как вместо макросов LOWORD и HIWORD использовать обычные операторы языка Си, добиваясь тех же результатов?

Лабораторная работа №6

Цель работы. Изучение средств событийного программирования и особенностей взаимодействия в текстовом и графическом режимах ОС Linux.

Краткие теоретические сведения. Для организации многофункционального вывода в текстовое окно (консольное окно) Linux используются управляющие последовательности. Управляющие последовательности, более точно называемые «управляющими последовательностями ANSI», предназначены для управления экраном и клавиатурой компьютера и состоят из символа начала управляющей последовательности (с числовым кодом 0x1B в шестнадцатеричной системе счисления), за которым следует символ «[» левой квадратной скобки, далее данные и код команды. Размер и тип данных зависит от кода команды. Указанный специальный числовой код, задающий начало управляющей последовательности, записывается на языке Си в составе текстовых строк как последовательность символов \033, что соответствует требуемому числовому значению кода в восьмеричной системе. Для относительной краткости в следующей таблице он будут условно изображаться именем esc. Основные типы управляющих последовательностей приведены в табл. 2.

Таблица 2. Управляющие последовательности.

Последовательность ANSI	Описание
\033[строкиA	(Переместить курсор вверх) Перемещает курсор вверх на заданное число строк, при этом номер столбца не изменяется. Если число строк не задано, то курсор перемещается вверхна одну строку. Когда курсор находится в верхней строке экрана, эта последовательность игнорируется.
\033[строкиB	(Переместить курсор вниз) Перемещает курсор вниз на заданное число строк, при этом номер столбца не изменяется. Если число строк не задано, то курсор перемещается вниз на одну строку. Когда курсор находится в нижней строке экрана, эта последовательность игнорируется.
\033[столбцыC	(Переместить курсор вправо) Перемещает курсор вправо на заданное число столбцов, при этом номер строки не изменяется. Если число столбцов не задано, то курсор перемещается вправо на одну позицию. Когда курсор находится в самой правой позиции строки, эта последовательность игнорируется
\033[столбцыD	(Переместить курсор влево) Перемещает курсор влево на заданное число столбцов, при этом номер строки не

	изменяется. Если число столбцов не задано, то курсор перемещается влево на одну позицию. Когда курсор находится в самой левой позиции строки, эта последовательность игнорируется.
\033[строки;столбцыH	(Установить позицию курсора) Перемещает курсор в заданную позицию экрана.
\033[2J	(Очистить экран) Очищает экран и перемещает курсор в исходное положение (строка 0, столбец 0)
\033[K	(Очистить строку) Удаляет все символы, начиная с позиции курсора до конца строки (включая символ в позиции курсора).

Общие атрибуты текста задаются следующими константами: 0 – отменить все атрибуты , 1 – повышенная яркость, 2 – пониженная яркость

Константы задания цвета даются в табл. 3. Параметры 30-47 в ней соответствуют стандарту 6429 Международной организации по стандартизации (ISO).

Таблица 3. Коды цветов изображения и фона

Цвета изображения		**Цвета фона**	
Цвет	Описание	Цвет	Описание
30	Черный	40	Черный
31	Красный	41	Красный
32	Зеленый	42	Зеленый
33	Желтый	43	Желтый
34	Голубой	44	Голубой
35	Малиновый	45	Малиновый
36	Бирюзовый	46	Бирюзовый
37	Белый	47	Белый

Графический режим поддерживает не собственно ОС Linux, а специальная универсальная внешняя по отношению к ОС графическая система, называемая X Window System (или просто система X). Она также требует отдельно задаваемых системных функций создания окна и ряд вспомогательных, но несколько проще для программирования на уровне API.

Простой пример использования такой системы дает следующий пример программного кода.

```
#include <X11/Xlib.h>

Display *dspl;
int screen;
Window hwnd;
XEvent event;

void main()
{ dspl = XOpenDisplay(NULL);
  if (dspl==0) {printf("Error XOpenDisplay\n"); exit(1);}
  screen = XDefaultScreen(dspl);

  hwnd=XCreateSimpleWindow(dspl, RootWindow(dspl,screen),100,50,300,200,3,
            BlackPixel(dspl,screen),WhitePixel(dspl,screen));
  if (hwnd==0) {printf("Error XCreateSimpleWindow\n"); exit(1);}
  XSelectInput(dspl, hwnd, ExposureMask | KeyPressMask | ButtonPressMask);
  XmapWindow(dspl, hwnd);
  while(1)
  {XNextEvent(dspl,&event);
   printf("Event!!\n");
   switch(event.type)
    {case KeyPress:
      printf("KeyPressed\n");    break; // вывод в текстовое окно
     case ButtonPress:
      printf("ButtonPressed\n"); break;   // вывод в текстовое окно
    }
   }
}
```

Для получения информации о событии при работе с X Window System следует после получения структуры данных события (возвращаемого параметра функции XNextEvent) извлечь содержимое соответствующих полей. Структура данных XEvent описана в заголовочном файле Xlib.h и содержит кроме поля type объединение (union) ряда разнообразных структур данных, специализированных для конкретного типа события. Эта часть структуры XEvent специфицируется обозначением **xkey** - для сообщений от клавиатуры (для типа события KeyPress), обозначением **xbutton** - для сообщений от мыши (для типа события ButtonPress) и обозначением **xconfigure** - для сообщений об изменении размера или позиции окна (для типа события ConfigureNotify). Во всех перечисленных типах событий имеются внутренние поля с именами **x** и **y** (для указания пози-

ции мыши, положения окна, а для клавиатуры - на всякий случай, который кстати, бывает, что имеет место).

При обработке сообщений от мыши следует анализировать значение поля, которое для структуры события с именем event запишется как **event.xbutton.button** . Значение 1 этого поля отвечает нажатию левой клавиши, значение 3 - нажатию правой клавиши, а значение 2 - нажатию средний клавиши мыши. Заметим, что как правило, в приложениях для X Window System используется 3-клавишная мышь, а при ее аппаратном отсутствии третья клавиша мыши имитируется одновременным нажатием правой и левой клавиш. Нажата или отпущена клавиша в данном событии от мыши - определяет поле state. В сообщении об изменении размера окна поля width и height дают информацию о текущих на момент события значениях ширины и высоты окна. Следует иметь в виду, что в событии типа ConfigureNotify поля с именами **x** и **y** в действительности несут информации о позиции только тогда, когда значение поля send_event не равно нулю (оно здесь - в частном случае - является индикатором действительности полей положения). Так что перед использованием полей положения окна при обработки события, структура которого обозначена именем event, следует проверять значение **event.xconfigure.send_event**.

Во всех структурах данных событий присутствует поле с именем **window**. Это поле содержит идентификатор (хэндл) того окна, которое послало сообщение. Очень широко и совершенно необходимо это значение используется при многооконной организации графического приложения, когда отдельные окна пересылают другим, в частности родительскому рабочую информации.

Первое по необходимости изобразительное действие в графическом окне - это вывод текста. Для него предназначен ряд функций, из которых самая простая и употребительная имеет прототип

XDrawString(Display* dspl, Window hwnd, GC gc, int x, int y,
char* text, int size).

Прежде чем строить изображение в окне программе для X Window System, необходимо иметь в оперативном распоряжении набор инструментов для такого рисования. Этот набор инструментов и текущих характеристик называется в X *графическим контекстом* - graphic context и имеет для своего описания структуру данных с типом GC указателя на нее. Заметим, что сама структура графического контекста (типа XGC) строится, заполняется и изменяется не явно, а с помощью вызова соответствующих функций X. Графический контекст содержит все необходимое для дальнейшего рисования, в частности фоновый цвет (background color), передний цвет (foreground color), ширину пера (pen width), тип линии, тип соединения линий (join type) и т.д.

Перед использованием любой функции построения изображения следует получить или построить графический контекст. Самый простой способ для этого - просто добраться до графического контекста по умолчанию для используе-

мого экрана. Для этих целей служит макрокоманда XDefaultGC(dspl, *номер_экрана*), которая возвращает значение типа GC.

Задание. Разработать программа для Linux, которая должна создавать графическое окно стандартного размера и реагировать на события для нее. Работающая программа должна сообщать, какое событие произошло. Как минимум должна быть реакция на события от мыши и клавиатуры. Информация о щелчке мыши выводится кроме текста в текстовое окно, еще и начиная с той позиции графического окна, на которой находится курсор мыши во время щелчка. Тут же выводится номер позиции в виде двух чисел — номера строки и номера столбца этой позиции. По событию от клавиатуры для алфавитно-цифровых клавиш выдается сообщение о нажатой клавиши и символ клавиши. Завершение программы осуществляется с помощью традиционных средств графического окна. (Базовый вариант)

Усложненный вариант для максимального рейтинга требует при изменении размера окна выдавать в окно текст, сообщающий новый размер в виде числового значения ширины и длины, выраженных числом пикселов.

Контрольные вопросы.

1. Что общего в системных функциях GetMessage и XNextEvent, а в чем они по существу отличаются?
2. Что общего в системных функциях TextOut и XDrawString, а в чем они по существу отличаются?
3. Что общего в системных функциях SetConsoleMode и XSelectInput?

Лабораторная работа №7

Цель работы. Изучение средств использования множественных процессов в Windows.

Краткие теоретические сведения. Излишняя изоляция процессов друг от друга в Windows влечет проблему группового использования процессов для общей работы. Ее решение разработчики MS стали искать не только на пути усложнении набора системных функций, но и введением новых «сущностей». Такой новой сущностью является понятие *задания* (Job) в последних версиях ОС Windows.

Новый вид внутрисистемного объекта Job позволяет собрать под одной «крышей» такого объекта произвольную группу процессов, выбираемых программистом. Это предоставляет возможность задать этой группе процессов специфические общие характеристики выполнения (приоритеты, время максимального непрерывного владения процессором и т.п.), а, главное, предоставить возможность в будущем одним простым приказом прекратить все эти процессы, т.е. задать их немедленное уничтожение.

Следует учесть, что эта возможность появилась только в 5-й программной версии Windows, а в предыдущих отсутствовала. Указанная особенность требует для указания системных функций, действующих на этот объект, в ряде универсальных систем разработки явного указания программных возможностей библиотек трансляции для указанной версии. На уровне прямых указаний языка С это приводит к необходимости задания специальной константы для препроцессора, которая в данном конкретном случае записывается в отдельной строке программы как

#define _WIN32_WINNT 0x0500

Эта директива препроцессора должна задаваться до остальных заголовочных файлов, если среда разработки не формирует ее автоматически, как, например, в MS Visual Studio.

В самом тексте программы на Си задается значение внешней текстовой константы WINVER компилятора. Имеются и средства задания этой константы извне собственного текста программы.

Использование программной единицы работы Job в качестве исходного шага требует создание объекта ядра типа Job. Для этого предназначена функция API с прототипом

HANDLE CreateJobObject(SECURITY_ATTRIBUTES **lpJobAttributes*, // SD
CTSTR **lpName);* // job name

При простейшем использовании единицы работы Job только для последующего уничтожения всей группы процессов второй параметр практически не нужен и задается значением NULL, первый параметр при неиспользовании сложной системы защиты объектов также задается не действующим с помощью той же константы NULL.

Собственно подключение процесса в такую единицу работы осуществляется с помощью функция API с прототипом

BOOL AssignProcessToJobObject(HANDLE *hJob*, // handle to job
HANDLE *hProcess*);// handle to process

В качестве действующего значения первого аргумента в ней используется полученное от функции CreateJobObject.

Имеется тонкая особенность включения процесса в единицу работы. В некоторых сложных ситуациях для этого необходимо, чтобы процесс в момент включения был не активным. Для этого в простейших случаях его можно создать приостановленным путем задания в шестом параметра функции CreateProccess значения CREATE_SUSPENDED (параметре флагов и опций создания процесса). Альтернативой является прямая приостановка нитей процесса на время выполнения функции AssignProcessToJobObject.

После включения процесса в состав единицы выполнения, все создаваемые этим процессом дочерние процессы и последующие их потомки оказываются принадлежащими к той же единице работы, и в последующем могут быть уни-

чтожены одним простым распоряжением. Оно задается в виде системной функции TerminateJobObject с прототипом

BOOL TerminateJobObject(HANDLE *hJob*, UINT *uExitCode*);

Последний параметр в ней имеет тот же смысл, что и также называемый параметр в функции ExitProcess. После исчезновения потребности в единице работы рекомендуется закрыть доступ к нему с помощью системной функции CloseHandle.

В графических оболочках Windows имеется один из важнейших компонентов ручного управления компьютером, вызываемый через комбинацию Alt-Ctr-Del, называемый «диспетчером задач». На его второй вкладке, поименованной *Процессы*, находится информация о основных процессах, выполняемых в настоящее время. В графическом окне этой вкладки во второй колонке содержатся значения PID, а в первой – имя процесса.

Задание. В работе нужно разработать пять программ – одну для процесса-родителя, две – для дочерних процессов, еще две – для «внучатых» процессов. Все программы осуществляют вывод сообщений о своем «родственном» положении (например, «родитель», «дочерний», «внук» и номер шага в цикле вывода таких сообщений, между которыми следует задавать задержки в пределах 1-2 секунд. Число повторений таких выводов сообщений должно составлять 10 – 20 для каждой программы, задержки для них следует выбирать различные, но незначительно – отличающиеся на 30–70%). На 7 шаге для вывода сообщений родительская программа отдает приказ об уничтожении первой дочерней и соответствующее дополнительное сообщение об этом на экран, на 11 шаге родительская программа выдает приказ об уничтожении второй дочерней и ее потомка. Все такие программы, создающие новый процесс, должны передавать в дочерний процесс текстовую информацию о присваиваемом ею текстовом имени дочернего (аналогичном «человеческим» именам). Это имя имеет исключительно иллюстративное значение и должно выводиться каждым процессом кроме исходного родительского с соответствующим примечанием о его назначении. (Базовый вариант.)

Каждый новый процесс расширенного варианта должен запускаться с отдельном текстовом окне, так что для пяти процессов должно оказаться пять работающих окон. Кроме того, каждый процесс, запускаемый разрабатываемой дочерней программой, должен передавать своему дочернему («внучатому») не только присваиваемое иму имя, но и свое собственное, присвоенное ему ранее текстовое имя, а внучатый сообщать не только свое полученное имя, но и текстовое имя своего родителя, в аналогии с использование отчества в русском языке. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Объясните, чем принципиально отличаются идентификаторы системных объектов от хэндлов, приведите пример использования тех и других в ОС Windows.

2. Какая информация возвращается в родительскую программу после создания нового процесса с помощью функции CreateProcess?

3. Как создать новый процесс, чтобы он использовал для вывода текстовую консоль, отличную от той, которую использует родительский процесс?

4. Что произойдет после закрытия хэндла дочернего процесса, выполненного родительским процессом, как это повлияет на сам дочерний процесс?

Лабораторная работа №8

Цель работы. Изучение средств и методов использования многопоточных программ в Linux.

Краткие теоретические сведения. Многопоточные программы развивают возможности программных процессов. Разработка многопоточных программ дело не простое, поскольку функционирование таких программ существенно отклоняется от классического понятия алгоритма. В обычной программе последовательность действий однозначно определяется ее алгоритмом или, для программ управляемыми данными, – текущими значениями этих данных. В многопоточных программах, использующих общие данные, результаты некоторых действий, а, как следствие, и сами эти действия не предопределены ни алгоритмом, ни значениями данных. Результаты некоторых взаимодействий между отдельными потоками (threads) в них принципиально не детерминированы, а, если разбираться в текущих деталях, существенно зависят от временных характеристик элементов таких взаимодействий. В частности, зависят от времени выполнения отдельных команд, от ожидания асинхронных сигналов, от времени доступа к данным на внешних носителях и т.д.

Все это принципиально усложняет разработку таких программ. Но их широко используют в решении многих задач, в частности в моделировании явлений реального мира, поскольку эти особенности глубоко аналогично поведению частей этого мира. Дело в том, что функционирование относительно автономных (в частности движущихся) естественных систем реального мира в большинстве своих проявлений происходит существенно независимо друг от друга. Например, движение, в частности, отдельные действия животных, собранных даже в стада и стаи, происходят в основном независимо. Более того строго согласованное поведение автономных систем требует специальных средств, нередко сложных или нетривиальных для согласования индивидуальных действий. В частности конкретные действия множества военных для согласования требуют специальных команд и специальных процедур согласования, а не возникают непроизвольно «сами по себе».

Разработка простейших мнопоточных программ в качестве первоочередной проблемы требует согласованного доступа различных потоков — нитей (threads) к общему средству отображения информации. Традиционными инструментами вывода информации на уровне API операционной системы являются функции или управляющие последовательности трех типов. Во-первых, задания позиции вывода, во-вторых, задания цвета, в общем случае атрибутов вывода, и, наконец, вывод собственно текста. В ОС Windows для этих целей используются, как правило, различные системные функции. В POSIX для этого служат различные типы управляющих последовательностей. Теоретически возможно все эти различные типы воздействий включить в общий формат вывода и выводит единственным оператором форматированного вывода. Но на уровне элементарных действий (уровнем ниже чем операторы), реализуемых машинными командами может оказаться и как правило оказывается, что даже такое обобщенное действие осуществляется не одной, а несколькими командами и на этом реально используемой аппаратурой уровне возможно возникновение указанных проблем.

Но реальная многопоточность может проявляться в том, хотя это происходит и не часто, что одна нить выполнит установку позиции вывода для нее, но выполнить остальные части действий по выводу сразу же не сможет, поскольку операционная система определит исчерпание этой нитью ее очередного кванта времени. Тогда получает «управление» и процессор (или одно из ядер процессора) другая нить и та, в свою очередь, производит установку позиции вывода, задание цвета и собственно вывод. Когда «управление» вернется к первой из рассматриваемых нитей, то по ходу ее программы выполненное ранее действие установки позиции оказывается уже не действующим, так как изменено другой из рассматриваемых нитей. Оно уже «пройдено» в программе первой нити и поэтому будут выполняться сразу же последующие действия — установка цвета и собственно вывод текста. Таким образом, оказывается, что переключение аппаратуры процессора между нитями приводит к незапланированному в программе месту вывода в консольном окне. Фактически вывод оказывается по месту не соответствующим запланированному в программе для первой нити, а фактически осуществляется по месту за выводом второй. Аналогичные последствия могут произойти не с местом вывода, а с его цветом — вывод первая нить станет осуществлять не своим запланированным цветом, а «вклинившимся» цветом второй нити.

Поэтому участок программы (процедуры) нити, в котором осуществляется последовательность указанных действий по выводу, необходимо задать как неразрывную — атомарную последовательность с помощью средств синхронизации. Для этого проще всего использовать мьютексы. Практически следует применить в программе один мьютекс для указанного согласования в различных нитях, в том числе, в главной. При этом перед последовательностью установки позиции, задания атрибутов и собственно вывода текста запрашивается захват

мьютекса функций pthread_mutex_lock, а после конца этой последовательности указывается освобождение этого мьютекса.

При использовании в многопоточной программе функций pthread_cancel отмены нитей при указанной организации процедур вывода в консольное окно может возникать реальная ситуация, когда нить отменяется на стадии ее нахождения внутри интервала программы, ограниченного захватом и освобождения мьютекса. Тогда получается, что эта отменяемая (уничтожаемая) нить по ходу своих действий захватывает мьютекс, но не успевает его освободить до своего уничтожения. В результате внутреннее состояние мьютекса соответствует его «захваченности», иначе говоря, невозможности его захвата другой нитью, претендующим на него. Причем временный «хозяин» этого мьютекса не может отдать его в «общее пользование». Как следствие, другие нити, в частности главная нить, рано и поздно пытающиеся осуществить свой вывод в консольное окно, а именно вывод, защищаемый от одновременного вывода других нитей, не может фактически даже начать его, останавливаясь на вызове функции pthread_mutex_lock в своих программах. Возникает непредусмотренная программистом блокировка всего дальнейшего выполнения такой многопоточной программы.

Чтобы справиться с такой ситуацией, необходимо запретить отмену нити внутри критического интервала доступа к консольному окну для вывода. С этой целью следует использовать функцию

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, ...)

перед таким критическим интервалом и функцию

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, ...)

если после него по запланированным действиям следует разрешать уничтожение этой нити по требованию другой нити.

При построении критических интервалов доступа к ресурсам, используемых параллельно и одновременно различными нитями, следует исходить из общего принципа минимизации нахождения в таких интервалов. Дело в том, что относительно длительное нахождение нити в таком интервале потенциально тормозит выполнения других нитей. Они вынуждены останавливаться в начале своих аналогичных критических интервалов по доступу к тому же ресурсу. Из этого общего принципа следует требование не включения в критический интервал протяженных по времени действий, а вынос подобных действий программистов за пределы таких интервалов. Это в первую очередь относиться к использованию временных задержек («засыпания» нитей) с помощью системных функций типа sleep и им подобных.

Задание. Разработать программу с тремя дополнительными нитями (threads) относительно главной нити. Каждая из нитей должна использовать общие для всех нитей данные, представленные массивом символов, в которых записаны 20 первых букв латинского алфавита. Каждая из этих нитей на своем k-м шаге выводит со своей случайной задержкой на место «своего» столбца экрана k-ю

букву из указанного массива латинских букв, причем с числом повторений, равному условному номеру нити, умноженному на два. Каждая из используемых нитей должен осуществлять вывод своим цветом, отличным от остальных нитей. На 6-м шаге главная нить делает попытку отмены первой из дополнительных нитей, а на 11-м делает попытку отмены третьей из дополнительных нитей. Первая и третья дополнительная нити в начале своей работы запрещают свою отмену. Третья нить на 13 шаге разрешает отмену, но в отложенном режиме. Точку отмены эта нить устанавливает между 16 и 17-м шагом своей работы. Все управляющие указания должны отображаться сообщениями без прокрутки экрана (в фиксированные позиции экрана). (Базовый вариант.)

Первая и третья дополнительная нити должны использовать общую процедуру, которая исходя из информации своего аргумента, поступившей в нее при запуске, принимает правильные решения о горизонтальной позиции своего вывода и цвете вывода. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Укажите по возможности больше отличий в понятии программного процесса от понятия нити.
2. Укажите общие черты в понятии программного процесса и в понятии нити.
3. В каких программных разработках целесообразней использовать программные процессы, а в каких – программные нити.
4. Чем по содержательным последствиям отличается уничтожение программного процесса от уничтожения программной нити.

Лабораторная работа №9

Цель работы. Изучение методов построения многопоточных программ с синхронизацией.

Краткие теоретические сведения.

Основные материалом для данной работы служит глава девятая учебного пособия [1]. Одной из классических задач использования средств синхронизации параллельно выполняемых процедур является «задача читатели-писатели» (Readers-Writers). В этой задаче требуется обеспечить правильное функционирование многих абстрактных процессов, которые могут читать данные из общей области, и абстрактных процессов, которые могут записывать данные в эту область. Причем допускается параллельное чтение несколькими «читателями» общих данных, когда никакой абстрактный процесс не записывает данные в эту область, но допускается одновременная работа только одного абстрактного процесса, записывающего данные, причем одновременно с ним не должен работать ни один абстрактный процесс чтения из общей области данных.

Следует особо заметить, что для реализации указанной задачи недостаточно использовать единственное средство ограничения доступа к общей области данных. В частности, недостаточно такого единственного средства при многопоточной реализации. Причиной является существенное различие разновидностей доступа для нитей-читателей и нитей-писателей. Вторые из них выполняют доступ к общей области данных по записи и, поэтому не могут одновременно работать с ней. Если какая-то нить записывает в эту область данные, то любая другая, одновременно пишущая в нее смешивает свои и данные первой нити в совершенно непредусмотренном программами порядке, подобно двум преподавателем, пишущим на одной и той же аудиторной доске, не обращая внимания друг на друга. Поэтому логично организовать с помощью средств синхронизации полную невозможность одновременных действий по записи в эту область.

Для нитей-читателей ситуация принципиально иная. Любая из этих нитей может читать данные одновременно с любой другой, читающих их. Поскольку при чтении информация в общей области никак не меняется, нет никакого смысла требовать от них раздельного во времени доступа по чтению. Но возникают еще парочка проблем. Если при попытке чтения данных какой-то нитью-читателем реально оказывается, что эти данные одновременно с ее чтением меняет некоторая нить-писатель, то никаких достоверных данных получено быть не может: ни старых данных, которые заменяет нить-писатель, ни новых данных, формируемых последней нитью. Будет какая-та заранее не предопределенная смесь из байтов старых и новых данных, которые в общем случае ни для чего не пригодны. Поэтому нить-читатель может начинать чтение, в более детализированном описании «входить в критический интервал доступа к общим данным», только в том случае, если в тот же момент с ними не работает никакая нить-писатель. В свою очередь, нить-писатель, делающая попытку добраться до общей области данных должна быть приостановлена средствами синхронизации, если в тот же момент какая-то нить-читатель читает данные из этой области и эта нить-писатель должна пребывать в состоянии ожидания возможности приступить к своим действиям записи, пока не прекратят работу с этой область данных все нити-читатели, уже приступившие к чтению этих данных.

Найденное на заре компьютерной эры решение использует для решения этой задачи не одно, а целых два системных средства синхронизации. В абстрактной форме ими являются два двоичных абстрактных семафоров [1]. Один из них условно обозначаемый для формализации как W, предназначен в основном для блокировки операции записи данных. Второй, обозначаемый условно как просто S, предназначен не для блокировки доступа по чтению, а для временной блокировке доступа к вспомогательной переменной N. Эта переменная используется одновременно только одним из «читателей» и служит

для точного подсчета одновременно работающих с общими данными «читателей».

Поэтому обнаружение в программе, что значение переменной N изменилось от нуля к значению 1 однозначно свидетельствует, что нужно закрыть доступ всем писателям по доступу к общим данным. Обнаружение же ее значения равным нулю, после уменьшения на 1, свидетельствует о возможности разрешить писателям доступ по записи, если они того потребуют – запросят на следующих шагах своих действий или если они ожидают получения такого доступа, находясь в ожидании его.

После введения указанных обозначений до средств синхронизации, общие схемы действий процедур Reader и Writer могут быть описаны, как показано на рис. 1.

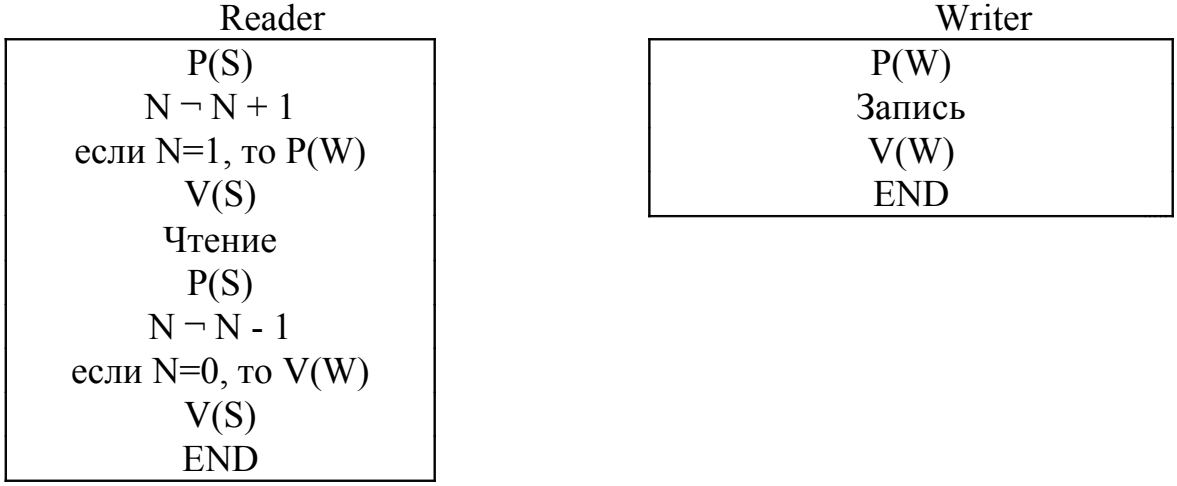

Рис. 1. Схемы программ «читателя» и «писателя»

Здесь с помощью обозначений P() и V() изображены специальные операции «проверки и захвата» и «освобождения» абстрактных семафоров S и W. Эти операции с точки зрения внутренних действий описаны в учебном пособии [1]. Перед началом работы программ для всех читателей и писателей должна быть выполнена инициализация

S = 1 , N = 0, W = 1.

Задание. Разработать многопоточную программу, отображающие на экране взаимодействие трех нитей «читателей» из общей области данных и трех «писателей», записывающих в этот буфер данные. Буфер предназначен для хранения 12 символов. Первая нить-писатель выводит в буфер данные в латинском алфавите строчными буквами, вторая нить-писатель выводит в буфер данные прописными латинскими буквами, а третья нить-писатель выводит в буфер специальные специальные символы – <+_!%^&*\|@~$>. Такой вывод эти три нити осуществляют в два приема, первый из которых записывает половину своего текста без завершающего этот промежуточный текст нуля. Между такими половинами вывода нити производят задержку на случайную величину миллисе-

кунд, но не более 1 сек. После вывода своего текста в буфер каждая нить-писатель переходит в ожидание порядка 2-3 сек до следующей попытки записи в буфер. Нити-читатели - через случайный интервал порядка 300 мсек - читают данные из буфера, если это позволяют средства синхронизации доступа между нитями, и вывод прочитанный текст на экран, каждая в свой столбец. Каждый вывод нити-читателя осуществляется в новую строку своего столбца, поэтому суммарные действия вывода в таких нитях предусмотреть только для 20 - 24 строк. Синхронизацию осуществить с помощью семафоров. (Возможны два вариант задания лабораторной работы - для Windows и для Linux). (Базовый вариант.)

Разработать такую же по поведению многопоточную программу, отображающие на экране взаимодействие трех нитей «читателей» из общей области данных и трех «писателей», записывающих в этот буфер данные, но все выводы нитей-читателей, предназначенные для наблюдения пользователем, должны отображаться не в текстовое, а в графическое окно. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Чем абстрактные семафоры отличаются от известных вам программных реализаций.
2. Укажите в каких ситуациях предпочтительней использование мьютексов вместо семафоров и в каких ситуациях целесообразно обратное предпочтение.

Лабораторная работа №10

Цель работы. Формирование умения использовать память, явно заказываемую в программе.

Краткие теоретические сведения. Основные материалом для данной работы служит глава 10 учебного пособия [1].

В императивном программировании оперативная память распределяется программному процессу по директивам описания данных – переменных или массивов. В них явно или неявно – размерами массивов или типом данных задается, сколько именно памяти нужно задействовать в оперативной памяти.

Но существуют задачи, для которых на стадии написания программы нельзя точно знать сколько конкретно памяти потребуется для ее выполнения. Такие ситуации встречаются очень часто при построении динамических структур данных: очередей, списков, структур типа «дерево» и т.п. Для решении принципиально непростой задачи – предоставить место в оперативной памяти данным, о появления которых становится известно только в ходе вычислений, но никак не раньше, приходится обращаться к помощи ОС.

Соответствующие средства API ОС называются средствами распределения и управления памятью. Простейшие из них только заказывают ОС, сколько до-

полнительной памяти нужно на текущий момент выполнения дополнительно программе. Выделяемую при этом ОС память нужно каким способом сделать доступной для обозначения и дальнейшего использования в программе. Простых типизованных обозначений переменных для этого недостаточно, просто потому, что такие переменных по смыслу их описания должны иметь фиксированный заранее известный размер.

Поэтому с указанной целью в системном языке программирования Си используются специализированный вид обозначения данных, называемый «указатель на данные». Практически этот вид данных очень близок по смыслу адресам объектов в реальном мире, где по общей форме задания адреса может находиться и небольшое строение и огромное здание или целое предприятие. Более того, на внутреннем архитектурно-системной уровне представления информации в машинных кодах, соответствующие средства так и называются – адресами данных, причем, в отличии от реального мира, адреса во внутреннем мире машинных кодов программ являются просто числовыми номерами ячеек памяти, каждая из которых величиной в один байт.

С точки зрения формализма синтаксиса языка Си указатель задается в виде

*тип *имя_указателя;*

где использование одного из стандартных для языка типов позволяет обращаться в дальнейшем только к одному или последовательности из элементов указанного типа. Подобное задание аналогично по смыслу формирования «визитной адресной карточки» для будущих реальных данных, в котором отсутствует пока информация, где территориально находится данный объект. Вместо стандартного (или производного от стандартного) типа можно использовать наиболее общее обозначение указателя, как указателя «на что угодно» или, иначе говоря, указателя без детализации типа, что записывается в виде

void имя_указателя;*

Чтобы таким указателем можно было непосредственно пользоваться, в него должна быть занесена адресная информации о конкретно существующих данных в доступной области памяти. Существует вспомогательная по существу возможность нацелить указатель на уже существующие данные, но эта возможность не основная и не ради нее созданы в языке указатели. Основное назначение их – попросить ОС выделить нужный объем дополнительной к текущей памяти и занести в такой указатель служебную информацию, где размещается эта память. Точнее указатель после выполнения таких действий будет содержать адрес начала размещения данных в оперативной памяти, как адрес большого комплекса зданий указывает в принятой системе адресования материальных строений только главную или начальную позиции их (главное здание, вход на территорию и т.п.)

Таким образом запрос на получение требуемого объема памяти известного размера *size* и запоминание, где начинается в памяти выделенный объем, в простейшим и очень перспективной форме требования запишется как

имя_указателя=имя_функции_запроса_памяти(size);

Именно такой синтаксис запроса использует ОС Linux, в которой унаследованное от ОС Unix функции запроса имеет имя malloc. Практически это сокращение наименования memory allocation.

Значение указателя на память, дающее по существу адрес начала участка на который он показывает, может быть получено с помощью формата %p в форматном списке функции printf или ей подобной. В частности может быть использован оператор вида

printf("Address места в памяти =%p\n", имя_указателя).

В ОС Windows получение произвольного блока дополнительной памяти в общем случае организуется более сложным образом. Для простейших ситуаций получения небольшого количества байт используется так называемый Heap (куча). Универсальной функцией для получения процессом любого дополнительного объема памяти, ограниченного только аппаратными возможностями и ограниченностью ресурсов, является API функция VirtualAlloc. Поэтому упрощенно изображая, можно считать, что такое получение дается вызовом

имя_указателя=VirtualAlloc(...,size, ...);

Сложности при этом заключается в том, что в отличии от malloc для Linux, требуется задать больше аргументов с правильно выбранными значениями. Для обеспечения особенно эффективной работы разработчики Windows вложили в эту функцию использование сложных внутренних особенностей системных архитектур современных процессоров, в первую очередь типа Intel. Эта сложность связана с тем, что внутреннее управление виртуальной памятью, реализуемое на стыке специального внутреннего программного обеспечения и аппаратуры, многоуровневое [1]. Для возможностей более полного его использования встроенная аппаратно-программная система управления памятью должна вначале распределить диапазон виртуальных адресов и подготовить на его основе набор «таблиц страниц». Только на втором этапе непосредственно заполняются эти таблицы или их часть и строится внутреннее отображение участка виртуальной памяти на реальную аппаратуру. Внешние функциональные возможности позволяют объединить оба этапа в один вызов функции VirtualAlloc, но это только частный случай, хотя в прикладных программах и наиболее употребительный. Соответственно этому соображению третий параметр указанной функции должен задавать первый или второй из описанных этапов (значения констант MEM_RESERVE или MEM_COMMIT), либо объединение обоих в одном вызове путем объединения таких констант в том же аргументе.

Последний аргумент функции дает для пользователей дополнительные возможности, присутствующие в самой внутренней виртуальной памяти, по ограничению операций с ней для повышения надежности функционирования сложных программ. Эти ограничения задают права либо только чтения из запрошенной области, либо чтения и записи. Кроме функции VirtualAlloc существует еще функция VirtualFree, задающая освобождение части или всей ранее запро-

шенной памяти, и VirtualProtect, которая дает возможность изменять вид доступа какой-то части ранее полученной через VirtualAlloc памяти.

Очень полезная функция VirtualQuery дает возможность узнать во время выполнения программы детальную информацию о любом месте раннее полученной памяти: с какого места начинается участок, в котором находится это место (адрес памяти), какова его выделенная длина, какой вид доступа к нему возможен и т.д.

Управление памятью со стороны ОС позволяет двум различным процессам использовать общий для них участок памяти, организуя таким образом предельно быстрое средство для взаимодействия по общим данным.

Если многопоточные программы могут легко использовать общие данные, которые описаны вне их действующих процедур, то для различных процессов такой простой путь невозможен. В конечном счете, эта невозможность играет положительную роль в многопрограммных ОС, так как даже ошибки поведения, не говоря уже о целенаправленных попытках вмешательства в данные другого процесса (в частности, другой программы), не могут помещать извне процесса его функционированию. Но в мнопрограммных приложений нередко, хотя и не часто, возникает потребность взаимодействия через общие данные. Простейший способ этого – через общие в использовании файлы, проблемы использования которых рассматривались в 3 и 4 лабораторных работах. Но этот способ очень медлителен с точки затрат реального времени, так как время доступа к файлам, размещенных на внешних носителях информации на много порядков (едва ли не в миллионы раз) медленнее, чем доступ к оперативной памяти.

Эти причины вынудили создателей ОС включить в их состав специальные средства организации общий, так называемой *разделяемой* между процессами памяти.

В конечном счете эти средства основаны на виртуальной памяти. В ОС Linux и Unix доступ к разделяемой памяти осуществляется в два этапа. На первом этапе через внутренний инструментарий виртуальной памяти создается специальный системный объект разделяемой памяти (share memory) или осуществляется доступ к уже существующей в ОС такой памяти. С учетом последнего требовалось как-то программно обозначать этот объект. Для этого разработчики использовали аналогию с доступом к файлам.

Функция shmget, описываемая прототипом

int shmget(key_t key, int size, int flag),

позволяет получить в качестве ее значение идентификатор разделяемой памяти. Первый ее аргумент задает пользовательское обозначение такого объекта, по смыслу аналогичного имени файла, но задаваемого специальным числовым кодом. В простейшем случае это может быть любое число, ранее не задействованное в ОС для указанных целей. Если же указанное число уже использовано для тех же целей, то делается попытка получить доступ к уже имеющемуся

объекту, причем для создания нового объекта в ОС необходимо использовать в последнем аргументе специальную константу IPC_CREAT вместе с указанием конкретных прав доступа, задаваемых совершенно аналогично правам доступа к создаваемым файлам.

Второй этап позволяет получить в процессе выполнения программы указатель (адрес) на разделяемый объект. Для этого служит функция

void* shmat(int shmid, void* addr, int flag),

в которую полученный ранее идентификатор передается как значение ее первого аргумента. Второй аргумент может задавать дополнительное пожелание и в простейших случаях записывается нулевым указателем NULL. Для отсоединения от виртуального пространства данного процесса разделяемой памяти служит обратная функция, задаваемая прототипом

int shmdt(void* addr).

Для ОС Windows средства использования разделяемой памяти несколько более сложны и основаны на не совсем профильном применении функций «отображения файла в память». Эти функции также реализуют два этапа: создания специализированного объекта виртуальной памяти (в этих функциях – объекта «отображения файла» – FileMapping) и собственно отображения его на виртуальную память. Первая из API функций задается прототипом

```
HANDLE CreateFileMapping(HANDLE hFile,
//хэндл файла или условное значение
   SECURITY_ATTRIBUTES *pFileMappingAttributes,
   DWORD protect, // protection for mapping object
   DWORD MaxSizeHigh,  // high-order 32 bits of object size
   DWORD MaxSizeLow,  // low-order 32 bits of object size
 CTSTR *pName);
```

Вторая из указанных функций дается прототипом

```
void* MapViewOfFile(HANDLE hFileMappingObject,
                    DWORD DesiredAccess,
                    DWORD OffsetHigh, DWORD OffsetLow,  DWORD size),
```

который и возвращает указатель на начало области разделяемой памяти.

Хотя первая функция предназначена в первую очередь для отображения файлов, ее использование для разделяемой памяти достигается заданием в качестве первого аргумента специального константного значения вместо хэндла файла. Это значение записывается как (HANDLE)-1.

Задание. Разработать программы в ОС Windows для двух отдельных процессов, использующих общую память. Второй процесс запускается как дочерний для выполняемой первой программы. В первой программе должна создаваться разделяемая память и семафор для взаимодействия между процессами. Во второй открывается доступ к этой же разделяемой памяти и семафору. Первая программа, запустив второй процесс, должна после задержки в 10 - 15 секунд запи-

сать какой-то текст в разделяемую память и указать его готовность с помощью семафора, а вторая программа должна вначале прочитать текстовую информацию из того места разделяемой памяти, где она должна появиться и выдать полученную информацию на экран с примечанием о существе действия, затем перейти к ожиданию разрешающего значения семафора и только после его завершения выдать на экран содержимое из разделяемой памяти с соответствующим примечанием. Вместо семафора в Windows можно использовать событие или мьютекс. Обе программы должны после получения действующего указателя на область общей памяти вывести адрес этой области в консольное окно с соответствующими пояснениями.

Первая программа после формирования данных для второго процесса, закрывает для себя доступ к разделяемой памяти. Далее она должна запросить 1000 байтов дополнительной памяти, никак не связанной с бывшей разделяемой, и используемой ею через другой или другие указатели. Дополнительно запрашиваемая память создается с помощью универсальной функции VirtualAlloc. Программа (первая) должна записать в эту область памяти, начиная с ее начала, сколько возможно символов латинского алфавита, записывая эти символы, пропуская по 399 свободных байтов (т.е. записывая 'a' в байт с нулевым смещением этой области памяти, затем 'b' в байт со смещением 400, далее 'c' в байт со смещением 800 и т.д., выполняя такое продвижение на 400 байтов по крайнем мере 15 раз). Каждую такую запись байта сопровождать детальным сообщением о действии и числовым значением адреса, по которому размещается этот байт в памяти. Наблюдаемые результаты объяснить. (Базовый вариант)

Первую часть задания с двумя процессами, обеспечивающими взаимодействие через разделяемую память разработать для Linux. (Дополнительное задание для максимального рейтинга).

Контрольные вопросы.

1. Чем абстрактные семафоры отличаются от известных вам программных реализаций.
2. Какие программные средства для решения проблема тупиков при взаимодействии программных единиц в Windows вы знаете?
3. Укажите в каких ситуациях предпочтительней использование мьютексов вместо семафоров и в каких ситуациях целесообразно обратное предпочтение.
4. Объясните, с чем связано большая сложность использования функций явного распределения памяти в Windows по сравнению с аналогичными функциями в Unix.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

[1] Флоренсов А.Н. Операционные системы для программиста. Учеб. пос. – Омск, Изд-во ОмГТУ, 2005. – 240 с.
[2] Буленок В.Г., Пьяных Е.Г. Основы работы с ОС Linux. Учеб. пос. – Томск: Изд-во ТГПУ, 2008. – 69 с.
[3] Побегайло А.П. Системное программирование в Windows. СПб.: БХВ-Петербург, 2006. – 1056 с.
[4] Мэтью, Н. Основы программирования в Linux. СПб.: БХВ-Петербург, 2009. – 896 с.
[5] Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. – СПб:. Питер; М.: Изд.-торг. дом "Русская редакция", 2001. – 752 с.
[6] Хэвиленд К., Дайна Г., Салама Б. Системное программирование в Unix. Руководство программиста по разработке ПО. – М.: ДМК Пресс, 2000. - 368 с.
[7] Митчел М., Оулдем Дж., Самьюэл А. Программирование для Linux. Профессиональный подход. – М.: Изд. дом "Вильямс", 2002. - 288 с.
[8] Иванов Н.Н. Программирование в Linux. Самоучитель. –СПб.: БХВ-Петербург, 2007. – 416 с.
[9] Флоренсов А.Н. Программирование для графического интерфейса. Семантический подход. – Омск: Изд-во ОмГТУ, 2002. - 128 с.
[10] Флоренсов А.Н. Программирование в графических оболочках операционных систем. - Омск: Изд-во ОмГТУ, 2007. - 104 с.