Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

# *СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ*

Лабораторный практикум
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

В 2-х частях

**Часть 1**

**В. А. Супонев, А. А. Уваров, В. А. Прытков**

## *Операционные системы*



Минск БГУИР 2009

УДК 004.451(075.8)
ББК 32.973.26-018.2 я73
С40

Рецензент
профессор кафедры информационно-вычислительных систем
учреждения образования «Военная академия Республики Беларусь»,
канд. техн. наук Д. Н. Одинец



**Системное** программное обеспечение ЭВМ : лаб. практикум для
С40 студ. спец. 1-40 02 01 «Вычислительные машины, системы и сети» всех форм обуч. В 2 ч. Ч. 1 : Операционные системы / В. А. Супонев, А. А. Уваров, В. А. Прытков. – Минск : БГУИР, 2009. – 36 с.
ISBN 978-985-488-365-6 (ч.1)

В лабораторном практикуме описаны лабораторные работы по курсу «Системное программное обеспечение ЭВМ».

Первая часть практикума посвящена изучению принципов организации операционных систем. В теоретической части каждой лабораторной работы приводятся базовые понятия и концепции, а также основные функции и структуры для их использования на примере операционных систем Windows XP и Linux. Практическая часть каждой лабораторной работы содержит задание для самостоятельного выполнения.

**УДК 004.451(075.8)**
**ББК 32.973.26-018.2 я73**

**ISBN 978-985-488-365-6 (ч.1)**

**ISBN 978-985-488-366-3**

**Содержание**

# ВВЕДЕНИЕ

Лабораторные занятия предполагают выполнение каждого задания в двух вариантах: под Linux и Windows. При этом будем считать, что студенты обладают базовыми навыками программирования под Windows, полученными при обучении на предшествующих курсах. Основные моменты, связанные с программированием под Unix-системы[1], будут рассмотрены в теоретической части каждой лабораторной работы данного практикума.

## Лабораторная работа №1
## ЗНАКОМСТВО С LINUX. ПОНЯТИЕ ПРОЦЕССОВ

*Цель работы*: ознакомиться с основами разработки программного обеспечения (ПО) под Linux; научиться создавать процессы под Unix и Windows, освоить базовые принципы работы с ними.

### 1.1. ЗНАКОМСТВО С LINUX

Для комфортной работы в среде Linux существует большое количество оконных менеджеров (KDE, Gnome, WindowMaker), интерактивных сред разработки (KDevelop, NetBeans, SlickEdit), текстовых редакторов (Kate, KWrite, OpenOffice Writer) и прочих графических средств, принцип работы которых уже стал более-менее привычен Windows-пользователю. Однако для понимания основ функционирования Linux и разработки приложений под Unix-подобные ОС необходимо иметь навыки работы с текстовой консолью и консольными приложениями, такими, как ViM, gcc, gdb и др. Выполнение задания лабораторной работы №1 предполагает использование исключительно консольных инструментов. В дальнейшем студент может выбрать наиболее удобную для него среду разработки.

Разработка приложений под Linux предполагает освоение следующих операций:

- работы с файловой системой;
- создания и редактирования текстовых файлов;
- компиляции и линковки программ;
- запуска исполняемых файлов;
- использования встроенных страниц справки (команды **man**).

Рассмотрим перечисленные операции подробнее.

[1] Строго говоря, понятия Unix и Linux не являются синонимичными. Linux – операционная система семейства Unix, которая имеет ряд отличий от классических Unix-систем. Однако темы, рассматриваемые в рамках данного практикума, как правило, имеют одинаковую реализацию как в Unix, так и в Linux.

***Работа с файловой системой***

Файловые системы, используемые в Linux, организованы по древовидному принципу. В отличие от FAT и NTFS здесь нет отдельных дисков, обозначаемых литерами. Файловая система имеет один общий корень, обозначаемый символом ‘/’, и иерархию каталогов, исходящую из него. Все дополнительные файловые системы, размещающиеся как на разделах жесткого диска, так и на CD, DVD, флэш-накопителях, дискетах и т.д., встраиваются в общее дерево в виде подкаталогов. Процесс связывания внешней файловой системы и локального каталога называется *монтированием* (см. команду **mount**).

Существуют два основных способа адресации файла или директории: относительный и абсолютный. В первом случае файл адресуется относительно текущей директории. При абсолютной же адресации указывается полный путь к файлу от корня файловой системы:

*/home/somefile.txt*

В данном случае файл somefile.txt находится в каталоге home, который, в свою очередь, расположен в корневой директории. Абсолютный адрес должен начинаться с символа ‘/’ – ссылки на корневой каталог. При указании как абсолютного, так и относительного адресов можно использовать метаобозначения ‘.’ (текущий каталог) и ‘..’ (родительский каталог).

Для работы с файлами и директориями в среде Linux используются следующие основные команды:

| | | |
|---|---|---|
| 1 | **pwd** | Получение абсолютного адреса текущей директории |
| 2 | **cd** | Смена текущей директории |
| 3 | **ls** | Получение списка файлов и каталогов в текущей директории |
| 4 | **cp** | Копирование файлов или каталогов |
| 5 | **mv** | Перемещение/переименование файлов или каталогов |
| 6 | **rm** | Удаление файлов |
| 7 | **mkdir** | Создание новой директории |
| 8 | **rmdir** | Удаление пустой директории |
| 9 | **cat** | Вывод на экран содержимого файла |
| 10 | **more (less)** | Страничный вывод содержимого файла |
| 11 | **tail** | Вывод на экран n последних строк файла |
| 12 | **chmod** | Смена прав доступа к файлу |
| 13 | **chown** | Смена владельца файла |

Подробное описание этих и других команд можно получить на страницах встроенной помощи (команда **man**).

***Создание и редактирование текстовых файлов***

Одним из наиболее мощных консольных текстовых редакторов в Linux является **ViM** (Vi iMproved – улучшенная версия редактора **Vi**). Для запуска ре-

дактора наберите “vim” в командной строке. Если в качестве параметра не было передано никакого имени файла, по умолчанию создается новый файл.

Принципы работы с **ViM** отличаются от привычного подхода к редактированию файлов. Полное описание возможностей данного редактора занимает несколько сотен man-страниц, поэтому здесь укажем только базовые его возможности, необходимые для работы.

**ViM** может находиться в одном из двух основных режимов работы: в режиме редактирования текста и режиме ввода команд. Индикатором режима редактирования является надпись INSERT в левом нижнем углу. Данный режим является стандартным при редактировании и трудностей не вызывает. Режим ввода команд активизируется нажатием клавиши Esc. Затем можно использовать основные команды управления файлом. Команда **ViM** обычно задается символом ‘:’, за которым следует обозначение команды и при необходимости ее параметры.

**ViM** насчитывает огромное количество команд, но сейчас укажем только команды выхода и сохранения файла (командам сохранения можно указывать имя файла в качестве параметра):

| | | |
|---|---|---|
| 1 | **:q** | Выход из редактора, если файл не был модифицирован |
| 2 | **:wq** | Выход из редактора с сохранением файла |
| 3 | **:q!** | Выход из редактора без сохранения изменений |
| 4 | **:w** | Сохранение изменений в файле без выхода из редактора |

***Компиляция и линковка программ***

Предположим, вы уже написали тестовую программу при помощи редактора ViM. Процесс трансляции исходного кода в бинарный исполняемый файл состоит из двух фаз: компиляции и линковки. Обе эти операции выполняются с помощью компилятора **gcc** (для C++ используется компилятор g++). Пример компиляции исходного кода:

*gcc –c somefile.o somefile.c*

В процессе компиляции из файла *somefile.c* будет создан объектный код *somefile.o*. Если производится компиляция нескольких исходных кодов в один объектный файл, то исходные файлы перечисляются через пробел.

Теперь скомпилированные объектные коды можно транслировать в исполняемый формат при помощи **gcc** с ключом –o:

*gcc –o somefile somefile.o*

В результате получим исполняемый файл с именем *somefile*. Как и в предыдущем случае, в конце можно указать список линкуемых объектных файлов.

Обратите внимание! Результирующий файл указан без расширения. В отличие от DOS/Windows, в Linux исполняемым является файл не со специфическим расширением (.exe), а с установленным соответствующим битом в правах доступа. В среде Linux-разработчиков принято, что исполняемые файлы, как правило, не имеют расширения. Это не означает, что все файлы без расширения – исполняемые. Многие конфигурационные файлы, например, также не

имеют расширения. Просто следует усвоить, что в Linux расширение файла не является существенным его атрибутом.

Важно отметить, что возможна также сокращенная форма трансляции:

*gcc somefile.c*

При этом будут выполнены фазы компиляции и линковки, и на выходе получим исполняемый файл с именем ***a.out***. Это один из немногих случаев наличия расширения у исполняемых файлов. При кажущейся легкости и простоте использования данный способ не рекомендуется ни в коем случае. Подобный прием может быть применен на этапе первоначальной отладки приложения, однако считается «дурным тоном» в программировании.

***Запуск исполняемых файлов***

После компиляции и линковки исходного кода в текущей директории окажется исполняемый бинарный файл. Вопреки шаблонному представлению запустить его, набрав имя файла в командной строке, не получится. Для запуска исполняемого файла следует указать его абсолютный или относительный адрес. Обычно используется адресация относительно текущего каталога:

*./somefile*

***Использование встроенных страниц справки***

В Linux существует встроенная система справки, доступная из консоли. Справка содержит страницы руководства по консольным командам, функциям языков программирования, установленным в системе, основным программам и т.д. Как правило, любая программа, предполагающая работу с консолью, инсталлирует свои страницы помощи в формате этой справочной системы.

Получить доступ к справочной системе можно при помощи команды **man**. В качестве параметра **man** требует ключевое слово, для которого запрашивается справка, например:

*man gdb*

Данный запрос выведет на экран страницы помощи по консольному отладчику gdb. Многие man-руководства насчитывают сотни страниц с подробным описанием всех возможностей конкретных программ. Выход из справочной системы осуществляется посредством клавиши ‘q’.

Однако бывают случаи, когда несколько разделов справки именуются одинаковым образом. Например, *time* – это и команда shell, и библиотечная функция языка C. Для разрешения подобных конфликтов страницы man разбиты на несколько разделов. На самих man-страницах можно встретить ссылки на другие команды и функции. Как правило, ссылки оформляются в виде *команда(раздел)*. Например, ссылка на команду *time* будет выглядеть так: *time(1)*. При наличии нескольких одинаково именованных разделов справки необходимо явно указать команде *man* раздел, в котором следует искать справочную информацию:

*man 1 time*

Таким образом, первым параметром идет номер раздела, вторым – команда, для которой запрашивается помощь.

Страницы помощи в Linux обычно разбиваются на следующие разделы:

| | | |
|---|---|---|
| 1 | Команды | Команды, которые могут быть запущены пользователем из оболочки |
| 2 | Системные вызовы | Функции, исполняемые ядром |
| 3 | Библиотечные вызовы | Большинство функций *libc*, таких как **qsort**(3) |
| 4 | Специальные файлы | Файлы, находящиеся в */dev* |
| 5 | Форматы файлов | Формат файла */etc/passwd* и подобных ему легко читаемых файлов |
| 6 | Игры | |
| 7 | Макропакеты | Описание стандартной «раскладки» файловой системы, сетевых протоколов, кодов ASCII и других таблиц, данной страницы документации и др. |
| 8 | Команды управления системой | Команды типа **mount**(8), которые может использовать только *root* |
| 9 | Процедуры ядра | Это устаревший раздел руководства. В то время когда появилась идея держать документацию о ядре Linux в этом разделе, она была неполной и по большей части устаревшей. Существуют значительно более удобные источники информации для разработчиков ядра |

## 1.2. ПОНЯТИЕ ПРОЦЕССОВ

*Процесс* – одно из средств организации параллельных вычислений в современных ОС. Под процессом обычно понимают выполняющуюся программу и все ее элементы, включая адресное пространство, глобальные переменные, регистры, стек, открытые файлы и т.д.

Параллельные вычисления подразумевают одновременное исполнение нескольких процессов. Для однопроцессорных систем характерно применение так называемого псевдопараллельного режима, когда операционная система последовательно выделяет для каждого процессора определенное количество квантов процессорного времени, тем самым достигается тот же эффект, что и при одновременной работе процессов.

Для управления процессом, обеспечения синхронизации и взаимодействия нескольких параллельных процессов необходимо знать, как идентифицируется процесс в операционной системе. Идентификатором процесса служит уникальный номер (pid), определяющий его положение в системных таблицах.

Ниже рассмотрим процедуры создания и завершения новых процессов в Linux и Windows.

### 1.2.1. Linux

Новый процесс в Linux создается при помощи системного вызова *fork()*. Данный вызов создает дубликат процесса-родителя. Выполнение процесса-потомка начинается с оператора, следующего после *fork()*. В случае успешного выполнения операции функция *fork()* возвращает родительскому процессу идентификатор созданного процесса-потомка, а самому процессу-потомку – 0; в случае ошибки функция возвращает -1.

Классический пример создания процесса:

```
pid = fork();
switch( pid ) {
        -1:    …;    // Ошибка
        0:     …;    // Дочерний процесс
default: …;          // Родительский процесс
}
```

Распараллеливание вычислений при помощи дублирования текущего процесса не всегда эффективно с точки зрения программирования. Для запуска произвольных исполняемых файлов существует семейство функций *exec*. Все функции этого семейства (*execv*, *execve*, *execl* и т.д.) замещают текущий процесс новым образом, загружаемым из указанного исполняемого файла. По этой причине exec-функции не возвращают никакого значения, кроме случая, когда при их запуске происходит ошибка.

Наиболее распространенная схема создания нового процесса в Unix – совместное использование *fork()*- и *exec*-функций. При этом дубликат родительского процесса, создаваемый *fork()*, замещается новым модулем.

При выборе конкретной функции необходимо учитывать следующее:

1. Функции, содержащие в названии литеру ‘p’ (*execvp* и *execlp*), принимают в качестве аргумента имя запускаемого файла и ищут его в прописанном в окружении процесса пути. Функции без этой литеры нуждаются в указании полного пути к исполняемому файлу.

2. Функции с литерой ‘v’ (*execv*, *execve* и *execvp*) принимают список аргументов как null-терминированный список указателей на строки. Функции с литерой ‘l’ (*execl*, *execle* и *execlp*) принимают этот список, используя механизм указания произвольного числа переменных языка C.

3. Функции с литерой ‘e’ (*execve* и *execle*) принимают дополнительный аргумент, массив переменных окружения. Он представляет собой null-терминированный массив указателей на строки, каждая из которых должна представлять собой запись вида “VARIABLE=value”.

Процесс завершается по окончании работы основного потока (main) либо после системного вызова *exit()*.

Получить идентификатор текущего процесса можно при помощи функции *getpid()*. Идентификатор родительского процесса возвращается функцией *getppid()*.

Родительский процесс должен явно дождаться завершения дочернего при помощи функций *wait()* или *waitpid()*. Если родительский процесс завершился раньше дочернего, не вызвав *wait()*, новым родителем всем его потомкам назначается init, корневой процесс ОС Unix. Если же дочерний процесс завершился, а процесс-родитель не вызвал какую-либо из вышеназванных функций, дочерний процесс становится так называемым «зомби»-процессом. По завершении родительского процесса процессы-«зомби» не могут быть унаследованы, т.к. уже завершены. Несмотря на то что они ничего не выполняют, они явно присутствуют в системе. Поэтому «хорошим тоном» в программировании считается контроль жизненного цикла дочерних процессов из родительского, когда разработчик не перекладывает эту задачу на систему.

### 1.2.2. Windows

В Windows создание процессов любого типа реализуется при помощи системной функции *CreateProcess()*. Функция принимает следующие аргументы:

*pszApplicationName* – имя исполняемого файла, который должен быть запущен;

*pszCommandLine* – командная строка, передаваемая новому процессу;

*psaProcess* – атрибуты защиты процесса;

*psaThread* – атрибуты защиты потока;

*bInheritHandles* – признак наследования дескрипторов;

*fdwCreate* – флаг, определяющий тип создаваемого процесса;

*pvEnvironment* – указатель на блок памяти, содержащий переменные окружения;

*pszCurDir* – рабочий каталог нового процесса;

*psiStartupInfo* – параметры окна нового процесса. Элементы этой структуры должны быть обнулены перед вызовом *CreateProcess()*, если им не присваивается специальное значение. Поле *cb* должно быть проинициализировано размером структуры в байтах;

*ppiProcInfo* – указатель на структуру PROCESS_INFORMATION, в которую будут записаны идентификаторы и дескрипторы нового процесса и основного его потока.

Существует четыре способа явно завершить процесс:

- входная функция первичного потока возвращает управление;
- один из потоков процесса вызывает *ExitProcess()*;
- любой поток любого процесса вызывает *TerminateProcess()*;
- все потоки процесса завершаются.

В Windows нет понятия «зомби»-процесса. Однако в ряде случаев необходимо дождаться окончания выполнения процесса. Делается это при помощи функции *WaitForSingleObject()* либо же *WaitForMultipleObjects()*.

### 1.3. ЗАДАНИЕ

Задание выполняется в двух вариантах: под Linux и Windows. Необходимо разработать консольное приложение, в котором базовый процесс порождает дочерний. Следует предусмотреть для каждого процесса свою область вывода, в которую выводится текущее системное время. Под Linux использовать библиотеку *ncurses*.

## Лабораторная работа №2
## СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

*Цель работы*: научиться организовывать синхронизацию нескольких параллельно выполняющихся процессов.

При разработке программ, использующих несколько параллельно выполняющихся процессов, могут возникать ситуации, требующие синхронизации вычислений. Например, существует понятие *состояния гонки (race condition)* – в случае ее возникновения корректность работы системы зависит от того, в каком порядке выполняются процессы при доступе в критическую область. Это означает, что должна соблюдаться строгая последовательность выполнения операций. Такое возможно, когда, например, один процесс должен подготовить ресурс к использованию другим процессом.

Иногда процессы одновременно используют некий общий ресурс, доступ к которому не должен перекрываться (например, два процесса выполняют посимвольный вывод текстовых строк в общий файл). В таком случае перед выполнением критической операции процесс должен заблокировать доступ к ресурсу до ее окончания. Только после этого другой процесс может получить доступ к этому ресурсу.

Как правило, для синхронизации процессов используют различные индикаторы занятости ресурса. В общем случае процедура синхронизации состоит из следующих шагов:

- проверить доступность ресурса;
- если ресурс доступен, заблокировать его. В противном случае подождать до разблокирования его другим процессом;
- выполнить необходимую операцию;
- разблокировать ресурс.

Как в Linux, так и в Windows существует большое количество способов синхронизации процессов. Мы рассмотрим по два способа для каждой из ОС.

## 2.1. LINUX

Наиболее распространенными инструментами синхронизации в Linux являются сигналы и семафоры.

***Сигналы***

*Сигналы* можно интерпретировать как программные прерывания. При получении сигнала выполнение текущей операции прекращается и вызывается функция-обработчик данного сигнала. Сигналы традиционно делятся на ненадежные и надежные. «Ненадежность» сигнала заключается в том, что в некоторых ситуациях процесс может не получить сгенерированный сигнал. Кроме этого, возможности процессов по управлению ненадежными сигналами довольно ограничены.

Одна из проблем ненадежных сигналов заключается в том, что обработчик сигнала сбрасывается в значение по умолчанию всякий раз, как срабатывает сигнал. Пример использования ненадежных сигналов:

```
 int    sig_int();      /* Прототип функции-обработчика сигнала */
   ...
signal(SIGINT, sig_int);  /* установка обработчика на сигнал */
   ...
sig_int()
{
   signal(SIGINT, sig_int);  /* восстановить обработчик */
   ...                  /* обработать сигнал... */
}
```

Данный фрагмент кода будет работать корректно в большинстве случаев, однако возможны ситуации, когда второй сигнал придет в промежуток времени между срабатыванием функции-обработчика и восстановлением обработчика посредством вызова *signal*. В таком случае будет вызван обработчик по умолчанию, что для SIGINT означает завершение процесса.

Для установки обработчика сигнала в ненадежной модели используется следующая функция:

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

В качестве параметров функция *signal* принимает номер сигнала (допустимы мнемонические макроопределения SIGINT, SIGUSR1 и т.д.) и указатель на функцию-обработчик, которая должна принимать значение типа *int* (номер сигнала) и не возвращать ничего. Функция *signal* возвращает указатель на предыдущий обработчик данного сигнала. В качестве второго параметра *signal*

можно указать константы SIG_IGN (игнорировать данный сигнал) или SIG_DFL (установить значение по умолчанию).

Надежная семантика сигналов подразумевает, что ядро ОС гарантирует доставку любого сигнала процессу. При этом процесс имеет возможность указать, какие сигналы он будет перехватывать, а какие игнорировать, т.е. составить так называемую *маску сигналов*.

Установка обработчика в надежной модели производится следующей функцией:

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act, struct sigaction *restrict oact);
```

Данная функция может как устанавливать новый обработчик, так и возвращать старый. Если параметр ***act*** не равен *NULL*, устанавливается новая реакция на сигнал. Если параметр ***oact*** не равен *NULL*, в этот указатель возвращается старый обработчик. Параметр ***signo*** – номер сигнала – аналогичен таковому в функции ***signal***.

В случае неудачной операции ***sigaction*** возвращает -1, в противном случае – 0.

Рассмотрим теперь структуру ***sigaction:***

```
struct sigaction {
  void     (*sa_handler)(int);
  sigset_t sa_mask;
  int      sa_flags;
  void     (*sa_sigaction)(int, siginfo_t *, void *);
};
```

Эта структура содержит следующие поля:

- ***sa_handler*** – указатель на функцию-обработчик. Может принимать значения SIG_IGN и SIG_DFL;
- ***sa_mask*** – дополнительная маска игнорируемых сигналов, которая добавляется к существующей, прежде чем будет вызвана функция-обработчик сигнала. После возврата обработчиком управления маска возвращается в предыдущее состояние;
- ***sa_flags*** – флаги, отвечающие за обработку сигнала. Обычно этот параметр устанавливается в 0;
- ***sa_sigaction*** – альтернативный обработчик сигналов. Используется, если выставлен флаг SA_SIGINFO.

Для генерации сигналов используются две функции: *kill()* и *raise()*. Первая служит для отправки сигнала произвольному процессу, вторая – текущему, т.е. самому себе. Синтаксис их вызова таков:

```
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
```

Здесь *pid* – идентификатор процесса, которому отправляется сигнал, а *signo* – номер сигнала.

***Семафоры***

Сигналы инициируют срабатывание функций-обработчиков в нужное время, что позволяет реализовать произвольную логику взаимодействия процессов. *Семафоры* – инструмент, разработанный исключительно для реализации механизма критических секций в межпроцессном взаимодействии (Inter Process Communication – IPC).

Семафор представляет собой глобальный системный счетчик, теоретически доступный всем процессам. При этом операции выполняются не над одним семафором, а над набором семафоров (*semaphore set*).

Создать набор семафоров можно следующей функцией:

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

Данная функция принимает следующие параметры:

- *key* – уникальный ключ, идентифицирующий набор семафоров. Имеет тип «длинное целое» (*long int*) и обычно генерируется при помощью функции *ftok()*;
- *nsems* – количество семафоров в наборе;
- *flag* – флаг, устанавливающий права доступа к набору семафоров.

Функция возвращает идентификатор набора семафоров в случае успешной операции и –1 при ошибке. При этом данная функция может как создавать новый набор семафоров (*flag* должен содержать значение *IPC_CREAT*), так и получать доступ к уже существующему. Для открытия существующего набора семафоров необходимо указать ключ *key*, идентифицирующий созданный ранее набор, при этом параметр *nsems* обычно устанавливается в 0.

Операции с семафорами производятся при помощи двух основных функций: *semop()* и *semctl()*.

Функция *semop()* работает следующим образом:

```
#include <sys/sem.h>

  int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Здесь *semid* – идентификатор набора семафоров, над которым производится операция; *semoparray* – массив структур типа *sembuf*, описывающих конкретные операции (если операция одна, массив представляет собой указатель на структуру); *nops* – количество операций (элементов в массиве).

Рассмотрим структуру *sembuf*:

```
struct sembuf {
  unsigned short  sem_num;
  short         sem_op;
  short         sem_flg;
 };
```

Здесь *sem_num* – номер семафора в наборе, над которым производится операция (начиная с нуля), s*em_op* – сама операция. При этом действие этого параметра зависит от принимаемого им значения:

- sem_op > 0: значение параметра прибавляется к текущему значению счетчика семафора;
- sem_op = 0: операция «дождаться нуля». Процесс переводится в состояние ожидания, пока значение семафора не станет равным нулю;
- sem_op < 0: модуль значения параметра отнимается от текущего значения счетчика семафора, если при этом результат не становится меньше нуля. В противном случае процесс переводится в состояние ожидания, пока такая операция не станет возможна, т.е. пока значение счетчика не станет больше либо равным модулю sem_op.

При этом, если параметр *sem_flg* установлен в значение IPC_NOWAIT, процесс в состояние ожидания не переводится, а во всех соответствующих случаях функция *semop()* возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN.

Состояние ожидания процесса прерывается не только установкой семафора в соответствующее состояние, но также в следующих случаях:

- при удалении набора семафоров, вызвавших блокировку процесса;
- при получении процессом сигнала, на который установлен обработчик.

Функция *semctl()* предоставляет расширенный спектр операций над семафорами, позволяя получать статус семафора, получать и устанавливать значение счетчика семафора, удалять набор семафоров и т.д. В общем случае для решения задач межпроцессного взаимодействия достаточно функции *semop()*, а *semctl()* обычно используется для удаления набора семафоров, когда в них отпадает надобность. Делать это необходимо, потому что семафоры являются

глобальными объектами ядра и сохраняются в системе и после завершения процесса.

## 2.2. WINDOWS

Для синхронизации процессов в Windows обычно используются события и семафоры.

***События***

Один из самых распространенных механизмов межпроцессной синхронизации в Windows – *события*. Они не являются полным аналогом Unix-сигналов, представляя собой бинарные флаги, имеющие два потенциальных состояния: сигнальное и несигнальное. Событие может быть одним из двух типов: со сбросом вручную и с автосбросом.

Для создания события используется функция *CreateEvent()*:

*HANDLE CreateEvent(PSECURITY_ATTRIBUTES psa, BOOL fManualReset, BOOL fInitialState, PCTSTR pszName);*

Функция принимает следующие параметры:

- *psa* – атрибуты защиты;
- *fManualReset – TRUE,* если событие сбрасывается вручную, *FALSE* в противном случае;
- *fInitialState* – начальное состояние события (*TRUE* – сигнальное, *FALSE* – несигнальное);
- *pszName* – уникальное имя-идентификатор события.

Получить доступ к событию из другого процесса можно одним из следующих способов:

- вызвать функцию *CreateEvent()* с тем же именем, которое было присвоено событию первым процессом;
- унаследовать дескриптор;
- применить функцию *DuplicateHandle()*;
- вызвать функцию *OpenEvent()* с указанием имени существующего события.

Управлять состоянием события позволяют функции *SetEvent()* – переводит событие в сигнальное состояние, и *ResetEvent()* – в несигнальное. Для ожидания события используется функция *WaitForSingleObject()*. Если событие создано с флагом автосброса, то при успешном окончании ожидания событие автоматически переведется в несигнальное состояние.

***Семафоры***

Второй рассматриваемый тип синхронизирующих объектов в Windows – *семафоры*. Принципиальных различий между семафорами в Unix и Windows не так много. Основное различие для программиста заключается в инструментальных средствах управления семафорами.

В Windows семафор создается функцией *CreateSemaphore()*:

*HANDLE CreateSemaphore( PSECURITY_ATTRIBUTE psa, LONG lInitial-Count, LONG lMaximumCount, PCTRTR pszName);*

Функция принимает следующие параметры:

- *psa* – атрибуты защиты;
- *fManualReset – TRUE,* если событие сбрасывается вручную, *FALSE* – в противном случае;
- *lInitialCount* – начальное значение счетчика семафора;
- *lMaxCount* – максимальное значение счетчика семафора;
- *pszName* – уникальное имя-идентификатор события.

Очевидно, что в отличие от Linux, Windows оперирует не наборами, а отдельными семафорами. Унаследовать созданный семафор можно теми же методами, что и событие. Для открытия семафора можно использовать функцию *OpenSemaphore().*

При вызове функции *ReleaseSemaphore()* значение счетчика ресурсов увеличивается. При этом его можно изменить не только на 1, как это обычно делается, но и на любое другое значение.

Дождаться освобождения ресурса (ненулевого состояния семафора) можно при помощи функций *WaitForSingleObject(), WaitForMultipleObjects(), MsgWaitForMultipleObjects().* При этом счетчик ресурсов автоматически будет уменьшен на единицу.

## 2.3. ЗАДАНИЕ

Задание выполняется в двух вариантах: под Linux и Windows.

Необходимо разработать многопроцессное приложение. Исходный процесс является управляющим, принимает поток ввода с клавиатуры и контролирует дочерние процессы. По нажатии клавиши ‘+’ добавляется новый процесс, ‘-’ – удаляется последний добавленный, ‘q’ – программа завершается. Каждый дочерний процесс посимвольно выводит на экран в вечном цикле свою уникальную строку. При этом операция вывода строки должна быть атомарной, т.е. процесс вывода должен быть синхронизирован таким образом, чтобы строки на экране не перемешивались. В качестве метода синхронизации следует использовать сигналы/события.

# Лабораторная работа №3
# ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ

*Цель работы:* научиться осуществлять передачу произвольных данных между параллельно выполняющимися процессами.

Взаимодействие нескольких параллельных процессов часто вызывает необходимость обмена данными между ними. При этом возникает следующая проблема: каждый процесс имеет собственное адресное пространство, при этом переменные одного процесса недоступны другим процессам. Обычно для передачи данных между процессами используют глобальные объекты ядра, выделение общих сегментов памяти, сокеты, каналы и т.д. Для каждой из рассматриваемых операционных систем существует большое количество методов и способов решения данной проблемы, поэтому подробно остановимся на наиболее часто используемых.

## 3.1. LINUX

Чаще всего для передачи данных между процессами в Linux используются каналы (pipes) и сегменты разделяемой памяти.

***Каналы***

*Канал* представляет собой специальный файл, связывающий два процесса. Таким образом, передача информации через канал сводится к записи в него данных в одном процессе и чтению из него в другом. Ранние Unix-системы не поддерживали полнодуплексную передачу данных, т.е. чтение и запись одним процессом через один канал; для обеспечения двусторонней связи приходилось создавать два канала. Несмотря на то, что практически все современные системы такую возможность предоставляют, при разработке приложений необходимо предусматривать наличие только полудуплексной передачи данных.

Канал создается системной функцией ***pipe()***:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Функции надо передать массив *filedes* из двух целочисленных элементов, первый из которых будет проинициализирован файловым дескриптором для чтения, а второй – для записи.

Если канал создан успешно, функция ***pipe()*** возвратит 0. В противном случае получим –1 .

Дескрипторы можно использовать напрямую функциями ***read()*** и ***write()***, а можно преобразовать к стандартному файловому потоку ***FILE**** функцией ***fdopen()***. Второй вариант позволяет использовать высокоуровневые функции форматированного ввода-вывода, такие как ***fprintf()*** и ***fgets()***.

Необходимо учитывать также, что передача данных через каналы возможна только для процессов, имеющих общего предка. Обычно канал создается до вызова *fork()* и затем используется в обоих процессах-клонах.

Существует правило, согласно которому неиспользуемый конец канала должен быть закрыт. Так, процесс-писатель закрывает дескриптор чтения (и наоборот).

***Сегменты разделяемой памяти***

Каналы обычно применяются для связи двух процессов. В случае, если взаимодействующих процессов больше, либо по каким-то причинам применение каналов исключается, для передачи данных можно использовать сегменты разделяемой памяти. При создании сегмента выделяется область памяти, идентифицируемая уникальным целочисленным ключом и доступная всем процессам, которые обратятся к ней по этому ключу.

Сегмент разделяемой памяти выделяется следующей функцией:

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Функции необходимо передать следующие параметры:

- *key* – уникальный ключ. Правила формирования ключа аналогичны случаю с семафорами;
- *tsize* – размер выделяемой области, в байтах;
- *flag* – флаг доступа, аналогичный случаю с семафорами.

При успешном завершении операции функция возвращает идентификатор созданного сегмента, в противном случае получим -1. Доступ к созданной ранее области разделяемой памяти, как и в случае с семафорами, можно получить через ключ *key*.

Для непосредственного использования разделяемой памяти необходимо получить адрес созданного сегмента разделяемой памяти с помощью функции *shmat()*:

```
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);
```

Функция принимает идентификатор существующего сегмента, указатель, в который будет записан адрес сегмента, и флаг доступа. После вызова функции с памятью можно работать через прямой указатель. При этом данные, записанные через этот указатель, будут доступны процессам, использующим созданный сегмент. После завершения работы необходимо вызвать функцию *shmdt()* для отсоединения сегмента.

Для выполнения расширенных операций над сегментом разделяемой памяти используется функция *shmctl()*. В спектре ее возможностей – получение и установка параметров сегмента, его удаление, а также блокирова-

ние/разблокирование. Возможность выполнения операции блокировки сегмента предоставляют не все Unix-системы; на данный момент эта возможность присутствует только в Linux и Solaris.

Чтобы избежать накопления неиспользуемых сегментов разделяемой памяти, каждый из них должен быть вручную удален после завершения работы с ним. Использование функций *exit()* и *exec()* отсоединяет сегмент, но не удаляет его. Удаление сегмента разделяемой памяти производится посредством вызова функции *shmctl()* с параметром IPC_RMID. В действительности удаление происходит только тогда, когда все процессы, использующие сегмент, отсоединяют его.

## 3.2. WINDOWS

Среди механизмов передачи данных между процессами в Windows наиболее часто используются каналы и разделяемая память.

***Каналы***

Как и в Linux, каналы в Windows также применяются для связи двух процессов. Процесс, создающий канал, называют *сервером*, а процесс, использующий этот канал, – *клиентом*. Каналы в Windows бывают двух типов: анонимные и именованные. Анонимные каналы быстры и универсальны, однако процессу-клиенту сложнее получить дескриптор такого канала, кроме того, они не поддерживают дуплексную передачу данных и не работают в сетях. Именованные каналы не имеют подобных недостатков, однако немного более тяжеловесны для операционной системы.

Использование каналов в Windows во многом подобно таковому в Linux. Рассмотрим основные функции для работы с каналами. Создание анонимного канала:

*BOOL CreatePipe( PHANDLE hReadPipe, PHANDLE hWritePipe, LPSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize);*

Функция создает анонимный канал, используя следующие параметры:

- *hReadPipe* – дескриптор чтения;
- *hWritePipe* – дескриптор записи;
- *lpPipeAttributes* – атрибуты защиты;
- *nSize* – количество байт, резервируемых для канала.

Создание именованного канала:

*HANDLE CreateNamedPipe( LPCTSTR lpName, DWORD dwOpenMode, DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOutBufferSize, DWORD nInBufferSize, DWORD nDefaultTimeOut, LPSECURITY_ATTRIBUTES lpSecurityAttributes);*

Функция принимает следующие параметры:

- *lpName* – имя канала. Как правило, представляет собой строку вида: «*Имя сервера*\pipe*Имя канала*». При использовании именованных каналов на локальной машине данная строка сокращается до «\\.\pipe\Имя канала»;
- *dwOpenMode* – режим открытия канала;
- *nMaxInstances* – максимальное количество реализаций канала;
- *nOutBufferSize* – размер выходного буфера;
- *nInBufferSize* – размер входного буфера;
- *nDefaultTimeOut* – время ожидания, в миллисекундах;
- *lpSecurityAttributes* – атрибуты защиты.

С целью дальнейшего использования канала с серверной и клиентской сторон применяют следующие функции: *ConnectNamedPipe()* (для именованных каналов), *CreateFile(), ReadFile(), WriteFile().* После установления соединения через канал с ним можно работать как с обычным файлом.

***Разделяемая память***

Еще один способ передать данные между процессами – использование файлов, отображаемых в память. При этом один процесс создает специальный объект, «файловую проекцию» (File Mapping), выделяя область памяти, которая связывается с определенным файлом и в дальнейшем может быть доступна глобально из других процессов.

Для передачи данных между процессами такое решение может показаться избыточным. Однако нет необходимости в специальном файле, проекция может использоваться исключительно для выделения виртуальной памяти без привязки к конкретному файлу.

Последовательность действий при этом такова. Для создания файловой проекции используется следующая функция:

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

Данная функция принимает параметры:

- *hFile* – дескриптор файла, связываемого с выделяемой областью памяти. Если связывать файл не нужно, а функция используется только для выделения общей области памяти, данный параметр устанавливается в *INVALID_HANDLE_VALUE.* При этом в качестве проецируемого файла будет использоваться системный файл подкачки. В этом случае также необходимо явно указать размер выделяемой памяти в параметрах *dwMaximumSizeHigh* и *dwMaximumSizeLow*;
- *lpAttributes* – атрибуты безопасности;
- *flProtect* – флаги защиты выделенной области;

- *dwMaximumSizeHigh* – старшее слово размера выделяемой памяти, в байтах;
- *dwMaximumSizeLow* – младшее слово размера выделяемой памяти, в байтах;
- *lpName* – имя объекта «файловая проекция».

После создания файлового отображения необходимо получить его адрес в памяти для того, чтобы записать туда передаваемые данные. Получить адрес можно при помощи следующей функции:

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

Принимаемые параметры:

- *hFileMappingObject* – дескриптор файловой проекции, полученный от предыдущей функции;
- *dwDesiredAccess* – режим доступа к области памяти;
- *dwFileOffsetHigh* – старшее слово смещения файла, с которого начинается отображение;
- *dwFileOffsetLow* – младшее слово смещения;
- *dwNumberOfBytesToMap* – число отображаемых байт. Если параметр равен нулю, отображается весь файл.

Данная функция возвращает указатель на спроецированную область памяти. После его получения можно записывать в полученную общую память необходимые данные.

Процесс-клиент должен получить доступ к памяти, выделенной другим процессом. Здесь необходимо учитывать, что файловая проекция уникально идентифицируется именем, указанным в функции *CreateFileMapping()*. Для получения дескриптора проекции следует воспользоваться функцией *OpenFileMapping()*. Далее по полученному дескриптору при помощи функции *MapViewOfFile()* можно получить указатель на искомую область памяти.

## 3.3. ЗАДАНИЕ

Задание выполняется в двух вариантах: под Linux и Windows.

Необходимо создать два процесса: клиентский и серверный. Серверный процесс ждет ввода пользователем текстовой строки и по нажатии клавиши Enter инициируются следующие действия:

- клиентский процесс ожидает уведомления о том, что серверный процесс готов начать передачу данных (синхронизация);
- серверный процесс передает полученную от пользователя строку клиентскому процессу, используя либо каналы, либо сегменты разделяемой памяти / файловые проекции;
- клиентский процесс принимает строку и выводит ее на экран;

- серверный процесс ожидает уведомления от клиентского процесса об успешном получении строки;
- серверный процесс вновь ожидает ввода строки пользователем и т.д.

В данной работе продолжается освоение синхронизации процессов. Уведомление процессов должно производиться посредством семафоров. Реализация механизма непосредственной передачи данных остается на выбор студента, однако в теории освоены должны быть все варианты.

# Лабораторная работа №4
# РАБОТА С ПОТОКАМИ

*Цель работы*: научиться создавать и уничтожать вычислительные потоки, а также управлять ими.

Понятие *потока* несколько различается в системах Linux и Windows. Более того, диспетчеризация потоков может быть реализована как на уровне пользователя (облегченные потоки Windows, fibers), так и на уровне ядра. В дальнейшем будем рассматривать исключительно потоки на уровне ядра.

## 4.1. LINUX

В Linux поток представляет собой облегченную версию процесса (*light-weight process, LWP*). Потоки можно представить как особые процессы, принадлежащие родительскому процессу и разделяющие с ним адресное пространство, файловые дескрипторы и обработчики сигналов. В отличие от процессов потоки более легковесны: переключение между ними выполняется быстрее, производительность системы увеличивается. Однако использование общего адресного пространства порождает ряд проблем, связанных с синхронизацией, и некоторые трудности при отладке.

При создании любого потока ему назначается так называемая *потоковая функция*, иногда именуемая телом потока. Эта функция содержит код, который поток должен выполнить. При выходе из потоковой функции поток завершается.

В Linux существует множество библиотек, предоставляющих интерфейс к системным вызовам, управляющим потоками. Стандартом де-факто является библиотека *pthreads* (POSIX threads), основные функции которой рассмотрим ниже:

*int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);*

Функция создает новый поток и принимает следующие параметры:

- *thread* – указатель, по которому будет записан идентификатор созданного потока;

- *attr* – атрибуты потока;
- *start_routine* – указатель на функцию потока. Она должна иметь прототип *void * start_routine(void *)*. Имя функции произвольно;
- *arg* – аргументы, передаваемые в поток. Если при создании потока необходимо передать в него какие-то данные, их можно записать в отдельную область памяти и данным параметром передать указатель на нее. Этот указатель будет являться и аргументом потоковой функции.

Для завершения потока используется функция *pthread_exit()*.

Поскольку потоки выполняются параллельно, вопрос о синхронизации остается открытым. Для решения этой проблемы главным образом используется функция *pthread_join()*. Данная функция принимает в качестве первого аргумента идентификатор потока и приостанавливает выполнение текущего потока, пока не будет завершен поток с указанным идентификатором.

Применение функции *pthread_join()* эффективно, но не всегда подходит для «тонкой» синхронизации, когда оба потока должны, не завершаясь, синхронизировать свои действия. В этом случае используются специальные объекты – *мьютексы*.

Мьютекс представляет собой бинарный флаг, имеющий два состояния: «свободно» и «занято». Для мьютексов применима общая процедура синхронизации, описанная в лабораторной работе №2. При этом используются следующие функции:

- *pthread_mutex_init* – создание мьютекса;
- *pthread_mutex_lock* – установка мьютекса в положение «занято». Если мьютекс уже занят, функция ждет его освобождения, а потом занимает его;
- *pthread_mutex_unlock* – установка мьютекса в положение «свободно»;
- *pthread_mutex_trylock* – функция, которая отличается от аналогичной функции *pthread_mutex_lock* тем, что при занятости мьютекса не ждет его освобождения, а немедленно возвращает значение EBUSY;
- *pthread_mutex_destroy* – уничтожение мьютекса.

## 4.2. WINDOWS

В Windows ситуация несколько отлична от Linux. Здесь процессы на самом деле не выполняют никакого кода, а являются лишь контейнерами для вычислительных потоков. Потоки представляют собой отдельные объекты ядра, коренным образом отличающиеся от процессов. Любой процесс может иметь несколько параллельно выполняющихся внутри него потоков, объединенных общим адресным пространством, общими таблицами дескрипторов процесса и т.д. Процесс должен содержать как минимум один поток, иначе он завершается. При создании процесса создается первичный поток, который вызывает входную функцию *main()*, *WinMain()* и т.д. При завершении входной функции управление вновь передается стартовому коду, и тот вызывает функцию *ExitProcess()*, завершая текущий процесс.

Работа с потоками в Windows, несмотря на основополагающие архитектурные отличия, во многом аналогична работе в Linux. В потоках Windows тоже используется понятие потоковой функции. Поток создается следующей функцией WinAPI:

*HANDLE CreateThread( PSECURITY_ATTRIBUTES lpThreadAttributes, DWORD dwStackSize, PTHREAD_START_ROUTINE lpStartAddress, PVOID lpParameter, DWORD dwCreationFlags, PDWORD lpThreadId);*

Функция принимает следующие параметры:

- *lpThreadAttributes* – атрибуты безопасности потока;
- *dwStackSize* – размер стека, выделяемого под поток;
- *lpStartAddress* – адрес потоковой функции, созданной в соответствии с прототипом *DWORD WINAPI ThreadFunc(PVOID pvParam)*;
- *lpParameter* – указатель на параметры, передаваемые функции потока при его создании;
- *dwCreationFlags* – флаг создания потока (поток запускается немедленно или создается в остановленном состоянии);
- *lpThreadId* – указатель, по которому будет записан идентификатор созданного потока.

Завершается поток одним из следующих способов:

- функция потока возвращает управление;
- поток вызывает функцию *ExitThread()*;
- текущий (или другой) поток вызывает функцию *TerminateThread()*;
- процесс, в котором выполняется поток, завершается.

Важно отметить, что создание потока при помощи функции *CreateThread()* может привести к некорректной работе приложения, если одновременно используются глобальные функции и переменные стандартной библиотеки C/C++ (*errno, strok, strerror* и т.д.). В таких случаях рекомендуется прибегнуть к функциям *_beginthreadex* и *_endthreadex*, которые реализуют аналогичную функциональность, но безопасны в работе.

Для решения проблемы синхронизации в Windows также применяются мьютексы, которые здесь, по сути, представляют собой бинарные семафоры и предназначены в первую очередь для взаимодействия процессов. Тем не менее использование подобных глобальных объектов в многопоточном приложении не всегда оправдано.

Более подходящим механизмом для синхронизации потоков являются критические секции,[2] логика работы которых во многом сходна с работой мью-

[2] На самом деле внутри критических секций используются семафоры, однако критические секции специально разработаны для синхронизации именно потоков. Они, по сути, не являются объектами ядра и не могут быть использованы для взаимодействия процессов. Наличие семафора внутри критической секции не приводит к снижению быстродействия, так как семафоры используются только для ожидания освобождения секции, а в большинстве случаев работа функций *EnterCriticalSection()* и *LeaveCriticalSection()* сводится к простому приращению счетчиков.

тексов в pthreads. Для работы с критическими секциями существует пять основных функций:

- *InitializeCriticalSection* – создание и инициализация объекта «критическая секция»;
- *EnterCriticalSection* – вход в критическую секцию. Это действие аналогично блокировке мьютекса. Если блокируемая критическая секция уже используется каким-либо потоком, функция ждет ее освобождения;
- *LeaveCriticalSection* – выход из критической секции;
- *TryEnterCriticalSection* – попытка входа в критическую секцию. Функция аналогична *EnterCriticalSection()*. Если критическая секция занята, возвращается 0, в противном случае – ненулевое значение;
- *DeleteCriticalSection* – удаление критической секции.

### 4.3. ЗАДАНИЕ

Задание аналогично заданию 2.3 из лабораторной работы №2, но с реализацией с помощью потоков. Выполняется в двух вариантах: под Linux и Windows.

## Лабораторная работа №5
## АСИНХРОННЫЕ ФАЙЛОВЫЕ ОПЕРАЦИИ. ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

*Цель работы*: научиться осуществлять асинхронные файловые операции; получить представление о динамических библиотеках, научиться создавать их и динамически использовать в приложениях.

### 5.1. АСИНХРОННЫЕ ФАЙЛОВЫЕ ОПЕРАЦИИ

Классическая схема чтения/записи файла выглядит следующим образом: вызвать системную функцию чтения/записи, указать ей параметры доступа к файлу, дождаться конца ее выполнения. В данном случае по окончании работы функции гарантируется, что операция чтения/записи окончена. Это так называемый случай *синхронной* файловой операции.

*Асинхронные* файловые операции реализованы по другому принципу. В этом случае функция чтения/записи лишь инициирует соответствующую процедуру, которая ставит задачу чтения/записи в очередь операций ввода/вывода, которой в дальнейшем занимается ядро ОС. Это означает, что функция чтения/записи возвращает управление немедленно, в то время как сама операция может выполняться еще некоторое время после этого в отдельном потоке. Асинхронные файловые операции, как правило, ускоряют работу приложения, однако в большинстве случаев необходимо убедиться, что операция завершена и все данные переданы. Поскольку реализация инициированной операции про-

грамме не видна, нужны средства, позволяющие определить текущий статус операции.

Реализация самих асинхронных операций, а также механизмы отслеживания результата их выполнения различны для разных операционных систем.

### 5.1.1. Linux

Асинхронные файловые операции в Linux реализованы в библиотеке *aio* (*asynchronous input-output*).

Все функции, реализующие асинхронный ввод-вывод, используют специальную структуру *aiocb*. Она имеет следующие поля:

- *int aio_fildes* – дескриптор файла;
- *off_t aio_offset* – смещение, по которому будет осуществлено чтение/запись;
- *volatile void *aio_buf* – адрес буфера в памяти;
- *size_t aio_nbytes* – число передаваемых байт;
- *int aio_reqprio* – величина понижения приоритета;
- *struct sigevent aio_sigevent* – сигнал, отвечающий за синхронизацию операции, т.е. за оповещение о ее окончании;
- *int aio_lio_opcode* – запрошенная операция.

Для асинхронных операций чтения/записи данных используются функции *aio_read()* и *aio_write()*, которые принимают адрес сформированной структуры *aiocb* в качестве единственного аргумента. Если запрос был принят и поставлен в очередь, функции возвращают 0, в противном случае возвращается -1.

Определить статус выполняемой операции можно либо при помощи поля *aio_sigevent* структуры *aiocb*, либо при помощи функции *aio_error()*. Если операция была завершена успешно, функция вернет нулевой результат. Если выполнение операции все еще продолжается, возвращаемое значение будет равно EINPROGRESS.

### 5.1.2. Windows

Выполнение асинхронных файловых операций в Windows схоже с библиотекой *aio* в Linux. Аналогом структуры *aiocb* в Windows является структура *OVERLAPPED*, имеющая следующие поля:

- *Internal* – код ошибки для операции;
- *InternalHigh* – количество переданных байт. Устанавливается после успешного завершения операции;
- *Offset* – младшее слово смещения в файле, с которого производится операция;
- *OffsetHigh* – старшее слово смещения;
- *hEvent* – дескриптор синхронизирующего события.

Для асинхронных файловых операций чтения/записи могут использоваться два семейства функций: стандартные и расширенные.

Во-первых, стандартные функции *ReadFile()* и *WriteFile()* могут работать в асинхронном режиме. Для этого надо передать им последним параметром указатель на сформированную структуру *OVERLAPPED*. Основным способом узнать, закончилась ли операция асинхронного ввода-вывода, в данном случае является использование поля *hEvent*. Его необходимо проинициализировать дескриптором существующего события, которое перед началом операции функции *ReadFile()/WriteFile()* переводят в несигнальное состояние. Переход же события в сигнальное состояние свидетельствует о том, что операция была завершена.

Во-вторых, возможно также использование расширенного семейства функций *ReadFileEx()/WriteFileEx()*. Этим функциям также надо передавать указатель на структуру *OVERLAPPED*, однако поле *hEvent* ими игнорируется. Вместо этого последним параметром эти функции принимают указатель на процедуру обратного вызова. Иными словами, можно сообщить *Ex*-функциям, какую функцию следует вызвать по завершении асинхронной операции.

## 5.2. ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

В программировании часто возникает ситуация, когда запущенные приложения используют одни и те же функции или участки кода. Многократно продублированные таким образом фрагменты кода приводят к избыточному использованию памяти. Очевидное решение: выделить общие функции в отдельные библиотеки, загружая в память лишь одну их копию, которая используется всеми приложениями. Такие библиотеки называются *разделяемыми*, или *динамически подключаемыми*. Рассмотрим детали реализации механизма динамических библиотек в различных операционных системах.

### 5.2.1. Linux

Создание динамической библиотеки в Linux полностью аналогично обычному бинарному исполняемому файлу и происходит по следующей схеме:

- компиляция исходного кода в объектный файл;
- линковка объектного файла в результирующий формат.

При помощи ключа *–shared* можно указать компилятору, что на выходе он должен породить динамическую библиотеку:

*gcc –shared –o mylibrary.so mylibrary.o*

Имя результирующей библиотеки в данном случае выглядит как *mylibrary.so*. Расширение *.so* (*shared object*) является обязательным.

Подключение динамической библиотеки к приложению можно осуществить двумя способами. Во-первых, библиотеку можно указать при линковке приложения с помощью ключа *gcc –l*, в этом случае при загрузке приложения система будет автоматически подгружать указанную библиотеку. Однако нередки случаи, когда для работы приложения нет необходимости постоянно иметь в памяти загруженную библиотеку либо когда из библиотеки использу-

ется лишь небольшое количество функций. В такой ситуации используется механизм динамического подключения библиотеки.

Для загрузки библиотеки в произвольный момент времени служит функция *dlopen()*. Первым параметром она принимает имя загружаемой библиотеки, а вторым – флаг загрузки, который может иметь одно из следующих значений:

- *RTLD_LAZY* – разрешение всех неопределенных ссылок в коде будет произведено при непосредственном обращении к загружаемым функциям;
- *RTLD_NOW* – разрешение всех символов производится сразу при загрузке библиотеки;
- *RTLD_GLOBAL* – внешние ссылки, определенные в библиотеке, будут доступны загруженным после библиотекам. Флаг может указываться через OR с перечисленными выше значениями.

Функция *dlopen()* возвращает дескриптор загруженной библиотеки. Для непосредственной работы с функциями и переменными, определенными в библиотеке, необходимо получить их адрес в памяти. Это можно сделать с помощью функции *dlsym()*, передав ей дескриптор подключенной библиотеки и имя соответствующей функции.

После завершения работы с библиотекой ее можно выгрузить из памяти функцией *dlclose()*.

### 5.2.2. Windows

Создание динамически подключаемых библиотек (DLL – Dynamically Linked Library) практически в любой среде разработки Windows сводится к выбору соответствующего шаблона при создании проекта, так что нет необходимости останавливаться на этом пункте подробнее. Гораздо больший интерес представляет динамическое подключение библиотеки.

Загрузка библиотеки производится функцией *LoadLibrary()*, которая принимает строку с именем dll-файла и возвращает дескриптор загруженной библиотеки. Получить адрес переменной (или функции) из библиотеки можно функцией *GetProcAddress()*, сообщив ей дескриптор библиотеки и имя соответствующего символа. Выгрузка библиотеки осуществляется при помощи функции *FreeLibrary()*.

## 5.3. ЗАДАНИЕ

Задание выполняется в двух вариантах: под Linux и Windows.

В каталоге имеется набор текстовых файлов. Необходимо разработать приложение из двух потоков, которые работают по следующей схеме:

1) первый поток (поток-читатель) асинхронным образом считывает содержимое одного файла;

2) поток-читатель уведомляет второй поток (поток-писатель) о том, что содержимое файла прочитано и может быть передано писателю;

3) поток-писатель получает от потока-читателя содержимое файла и асинхронным образом записывает полученную строку в конец выходного файла;

4) поток-читатель получает уведомление от потока-писателя о том, что строка записана в выходной файл и можно приступать к чтению следующего файла;

5) процедура повторяется с п.1, пока не закончится список файлов.

В результате должна быть произведена конкатенация (объединение) входных текстовых файлов в один результирующий.

Функции чтения – записи должны быть выделены в динамическую библиотеку, подключены на этапе выполнения программы и выгружены после отработки основного цикла.

# Лабораторная работа №6
# РАЗРАБОТКА МЕНЕДЖЕРА ПАМЯТИ

*Цель работы*: ознакомиться с основами функционирования менеджеров памяти, реализовать собственный алгоритм учета памяти.

## 6.1. ОБЩИЕ СВЕДЕНИЯ

В отличие от предыдущих лабораторных работ, которые сводились к изучению соответствующих библиотечных функций, данная работа посвящена в первую очередь архитектурному проектированию собственной системы, т.е. собственной реализации стандартных функций.

При программировании различных приложений часто возникает необходимость динамически выделить участок памяти заданного размера, получить указатель на него, изменить размер этого участка, освободить выделенную память и т.д. Для программиста эти операции сводятся к вызову функций *malloc(), realloc(), free()*, использованию операторов *new* и *delete* и т.д. Непосредственным же манипулированием участками памяти занимается так называемый *менеджер памяти*.

В стандартных библиотеках C/C++ менеджер памяти представляет собой реализацию перечисленных выше функций и операторов. Однако часто возникает ситуация, когда стандартного функционала менеджера памяти недостаточно. Например, может потребоваться индексирование выделенных участков памяти, их дефрагментация, сборка мусора и т.д. В этих случаях наиболее логично написание собственного менеджера памяти.

Традиционно выделяют следующие составные компоненты менеджера памяти:

- управляющие структуры, описывающие размещение выделенных областей памяти;
- набор функций, позволяющих оперировать выделением памяти (аналог *malloc(), free()* и т.д.);

- дополнительные внутренние функции и компоненты, осуществляющие сервисные операции (например, автоматическая сборка мусора).

Для выполнения задания необходимо также рассмотреть следующие сервисные операции:

- *Сборка мусора* – автоматическое освобождение менеджером памяти неиспользуемых участков памяти принудительно или в фоновом режиме. При этом подразумевается отсутствие функции *free()* (оператора *delete*) или совместная с ней работа. Как правило, неиспользуемым считается участок памяти, на который отсутствуют ссылки.
- *Дефрагментация* – процесс упорядочивания выделенных областей памяти и устранения пустого неиспользуемого пространства между ними. Фрагментация оперативной памяти возникает при последовательном выделении и освобождении памяти. Это означает, что возможна ситуация, когда общего объема свободной памяти достаточно для выделения, однако вся память сосредоточена в небольших пустых областях между выделенными участками. В такой ситуации невозможно выделить непрерывный участок памяти требуемого размера, несмотря на то, что нужный объем памяти в принципе помечен как свободный.
- *Свопинг* – процесс сброса на жесткий диск наименее используемых участков памяти для их освобождения под другие нужды. Это происходит, когда общего объема памяти становится недостаточно, и ведет к замедлению работы программы. При последующем обращении к освобожденному таким образом участку менеджер памяти должен считывать его с жесткого диска и вновь выделять для него память нужного объема.

## 6.2. ЗАДАНИЕ

Задание выполняется в одном варианте под любую операционную систему (по выбору студента).

Разработать собственный менеджер памяти, реализующий аналоги функций *malloc()* и *free()*. Архитектура менеджера и детали реализации остаются на усмотрение студента. Предусмотреть дополнительную функциональность в одном из следующих вариантов:

- динамическое изменение размеров выделенной области (*realloc()*);
- автоматическая сборка мусора;
- дефрагментация;
- механизм свопинга при превышении максимально доступной памяти.

# Лабораторная работа №7
# ЭМУЛЯТОР ФАЙЛОВОЙ СИСТЕМЫ

*Цель работы*: ознакомиться с основами функционирования и проектирования файловых систем, разработать собственную файловую систему.

## 7.1. ОБЩИЕ СВЕДЕНИЯ

Под *файловой системой* традиционно понимается способ хранения данных в виде файлов на жестком диске, внутренняя архитектура распределения данных, а также алгоритмы манипулирования файлами и их составными компонентами.

При проектировании файловой системы перед программистом обычно встает необходимость поддержки:

– иерархии размещения файлов и каталогов. Как правило, файлы хранятся в древовидной системе каталогов, которую надо спроецировать в физическое представление на конкретном носителе;

– алгоритмов добавления, удаления, модификации файлов;

– быстрого доступа как к описанию файлов (имя, атрибуты доступа и т.д.), так и к произвольным их участкам.

На сегодняшний момент существует большое количество файловых систем как общего назначения, так и специализированных. Обычно их классифицируют следующим образом:

- файловые системы для носителей с произвольным доступом (жестких дисков), например: ext2, ext3, ReiserFS, FAT32, NTFS, XFS и др. В Unix-системах обычно применяются первые три, использование FAT32 и NTFS характерно для ОС семейства Windows;
- файловые системы для носителей с последовательным доступом (магнитных лент): QIC и др.;
- файловые системы для оптических носителей: ISO9660, ISO9690, HFS, UDF и др.;
- виртуальные файловые системы: AEFS и др.;
- сетевые файловые системы: NFS, SMBFS, SSHFS и др.

В качестве примера рассмотрим основные принципы организации файловых систем Unix (ext2, ext3, ReiserFS и др.).

Основные компоненты физического представления файловой системы Unix:

- *суперблок* – область на жестком диске, содержащая общую информацию о файловой системе;
- *массив индексных дескрипторов* – содержит метаданные всех файлов файловой системы. Каждый индексный дескриптор (*inode*) содержит информацию о статусе файла и его размещении. Один дескриптор является корневым, и

через него производится доступ ко всей структуре файловой системы. Размер массива дескрипторов фиксирован и задается при создании файловой системы;

- *блоки хранения данных* – блоки, в которых непосредственно хранится содержимое файлов. Ссылки на блоки хранятся в индексном дескрипторе файла.

В суперблоке хранится большое количество служебной информации. Особый интерес для нас представляет количество свободных блоков и свободных индексных дескрипторов, размер логического блока файловой системы, список номеров свободных индексных дескрипторов и список адресов свободных блоков.

Два последних списка по понятным причинам могут занимать довольно большое пространство, поэтому их хранение непосредственно в суперблоке непрактично. Эти списки содержатся в отдельных блоках данных, на первый из которых имеется ссылка в суперблоке. Блоки данных организованы в виде списка; каждый блок, входящий в его состав, первым своим элементом указывает на следующий блок.

Индексный дескриптор ассоциирован с одним файлом и содержит его метаданные, т.е. информацию, которая может потребоваться для доступа к нему. Основной интерес для нас составляет физическое представление файла на жестком диске, учитывая то, что занимая довольно большой объем, файл может дробиться на небольшие блоки данных.

Каждый дескриптор содержит 13 указателей. Первые 10 указателей ссылаются непосредственно на блоки данных файла. Если файл большего размера, то 11-й указатель ссылается на первый косвенный блок (*indirection block*) из 128 (256) ссылок на блоки данных. В случае если и этого недостаточно, 12-й указатель, в свою очередь, ссылается на дважды косвенный блок, содержащий 128 (256) ссылок на косвенные блоки. Наконец последний, 13-й указатель ссылается на трижды косвенный блок из 128 (256) ссылок на дважды косвенные блоки. Количество элементов в косвенном блоке зависит от его размера.

Следует отметить, что в Unix нет четкого разделения на файлы и директории. Индексный дескриптор файла содержит поле *тип файла*, в котором указывается, что именно представляет собой данный файл. В числе возможных вариантов этого поля – обычный файл, директория, специальный файл устройства, канал (*pipe*), связь (*link*) или сокет.

Подобная архитектура файловой системы позволяет оптимальным образом разрешить перечисленные выше проблемы и получить быстрый и удобный доступ к файлам и директориям, а также их метаинформации.

Для получения более подробного представления о внутреннем устройстве файловых систем рекомендуется также ознакомиться с системами FAT32 и NTFS.

## 7.2. ЗАДАНИЕ

Задание выполняется в одном варианте под любую операционную систему (по выбору студента) на протяжении четырех академических часов (двух аудиторных занятий).

Необходимо разработать собственную модель файловой системы, в которой физический носитель будет эмулироваться файлом фиксированного размера. Архитектура файловой системы остается на усмотрение студента. В результате выполнения задания должны быть реализованы следующие компоненты:

- библиотека функций по добавлению, удалению и модификации файлов;
- простой файловый менеджер, основанный на данной библиотеке.

Все изменения, внесенные в файловую систему (иерархия директорий, файлы, их атрибуты), должны сохраняться в эмулирующем файле и быть доступными при последующем запуске приложения.

# Литература

1. Вахалия, Ю. UNIX изнутри / Ю. Вахалия. – СПб. : Питер, 2003.

2. Гордеев, А. В. Операционные системы: учеб. для вузов / А. В. Гордеев. – СПб. : Питер, 2005.

3. Карпов, В. Е. Основы операционных систем. Курс лекций: учеб. пособие / В. Е. Карпов, К. А. Коньков. – М. : Интернет-университет информационных технологий, 2004.

4. Робачевский, А. М. Операционная система UNIX / А. М. Робачевский, С. А. Немнюгин, О. Л. Стесик. – СПб. : БХВ-Петербург, 2005.

5. Рочкинд, М. Программирование для UNIX / М. Рочкинд. – СПб. : БХВ-Петербург, 2005.

6. Руссинович, М. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000 / М. Руссинович, Д. Соломон. – М. : ИТД «Русская редакция»; СПб. : Питер, 2005.

7. Столлингс, В. Операционные системы / В. Столлингс. – М. : ИД «Вильямс», 2004.

8. Таненбаум, Э. Современные операционные системы / Э. Таненбаум. – СПб. : Питер, 2004.

9. Харт, Д. Системное программирование в среде Windows / Д. Харт. – М. : ИД «Вильямс», 2005.

Св. план 2009, поз. 63

Учебное издание

**Супонев** Виктор Алексеевич
**Уваров** Андрей Александрович
**Прытков** Валерий Александрович

# СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Лабораторный практикум
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

В 2-х частях

**Часть 1**

***Операционные системы***



Редактор Г. С. Корбут
Корректор Е. Н. Батурчик

| | | |
|---|---|---|
| Подписано в печать 9.01.2009. | Формат 60×84 1/16. | Бумага офсетная. |
| Гарнитура «Таймс». | Печать ризографическая. | Усл. печ. л. 2,21 |
| Уч.-изд. л. 1,9 | Тираж 150 экз. | Заказ 548. |

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6