Содержание

Лабораторная работа №1
Знакомство с Linux. Понятие процессов

Цель работы: Ознакомиться с основами разработки ПО под Linux. Научиться создавать процессы под Unix и Windows, освоить базовые принципы работы с ними.

Знакомство с Linux

Введение

Лабораторные занятия по курсу СПО ЭВМ предполагают решение каждой задачи 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 ниже.

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 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 использования консольных приложений, таких как vim, gcc, gdb и др. Выполнение первого задания предполагает использование исключительно консольных инструментов. В дальнейшем студент волен выбрать наиболее подходящую для него среду разработки.

Ознакомление с разработкой приложений под Linux подразумевает освоение следующих операций:

- работа с файловой системой;
- создание и редактирование текстовых файлов;
- компиляция и линковка программ;
- запуск исполняемых файлов;
- использование встроенных страниц справки (man).

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,

[1] Строго говоря, понятия Unix и Linux не являются синонимичными. Linux – операционная система, входящая в семейство Unix, но имеющая определенные отличия от классических Unix-систем. Однако темы, рассматриваемые в рамках данного курса, как правило, имеют одинаковую реализацию как в Unix, так и в Linux

таки и на CD и DVD диска, 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, находится в корневой директории. Абсолютный адрес должен начинаться с символа '/', ссылающегося на корневой каталог. При указании как абсолютного, так и относительного адресов можно использовать метаобозначения '.' (текущий каталог) и '..' (родительский каталог).

Для манипулирования файлами и директориями можно выделить 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 в страницах встроенной помощи (man).

Создание и редактирование текстовых файлов

Одним из наиболее мощных консольных текстовых редакторов является **ViM** (Vi iMproved – улучшенная версия редактора **Vi**). Для его запуска наберите "vim" в командной строке. Если не было передано никакого имени файла в качестве параметра, по умолчанию создается новый файл.

Принципы работы с **ViM** отличаются от привычного подхода к редактированию файлов. Полное описание возможностей редактора занимает несколько сот man-страниц, здесь же обрисуем базовые его возможности, необходимые для работы.

ViM может находиться в одном из двух основных режимов работы: режим редактирования текста и режим ввода команд. Индикатором режима редактирования является слово “INSERT” в левом нижнем углу. Данный режим является привычным и трудностей не представляет. Режим ввода команд активируется нажатием клавиши Esc. После этого можно использовать основные команды для управления файлом. Команда **ViM** обычно представляет собой символ ‘:’, после которого следует обозначение команды и при необходимости ее параметры.

ViM насчитывает огромное количество команд, но сейчас нас будут интересовать команды выхода и сохранения файла:

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.

Командам сохранения можно указывать имя файла в качестве параметра.

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** (g++ для C++). Пример компиляции исходного кода:

gcc –c somefile.o somefile.c

При этом 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.

Скомпилированные объектные коды можно теперь транслировать в исполняемый формат. Для этого используется тот же **gcc**, но с ключом –o:

gcc –o somefile somefile.o

В итоге на выходе получится исполняемый файл с именем *somefile*. Как и в предыдущем случае, в конце можно указывать список из линкуемых объектных файлов.

Стоит обратить внимание, что результирующий файл был указан без расширения. В отличие от Dos/Windows, исполняемым является файл не со специфическим расширением (.exe), а с установленным соответствующим битом в правах доступа. В среде Linux-разработчиков принято, что исполняемые файлы расширения, как правило, не имеют. Это не означает, что все файлы без расширения – исполняемые. Многие конфигурационные файлы, к примеру, также не имеют расширения. Просто следует усвоить, что в Linux расширение файла не является существенным его атрибутом.

Стоит отметить, что возможна и сокращенная форма трансляции:

gcc somefile.c

При этом будут выполнены фазы компиляции и линковки, и на выходе получится исполняемый файл с именем *a.out*. Это один из немногих случаев

нарушения правила об отсутствии расширения у исполняемых файлов. При кажущейся легкости и простоте использования данный способ крайне не рекомендуется, так как дальнейшее использование такого «дефолтного» имени может вызвать путаницу. Подобный прием может быть применен на этапе первоначальной отладки приложения, однако считается весьма дурным тоном программирования.

Запуск исполняемых файлов

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

./somefile

Использование встроенных страниц справки

В Linux существует встроенная система справки, доступная из консоли. В ней содержатся 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 данном формате.

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, несколько разделов справки именуются одинаковым образом. К примеру, *time* – это и команда shell, и библиотечная функция языка Си. Для разрешения подобных конфликтов страницы man разделены на несколько разделов. В самих страницах можно встретить ссылки на другие команды и функции. Как правило, они оформляются в виде *команда(раздел)*. Например, ссылка на команду *time* будет выглядеть так: *time(1)*. Если подобная двусмысленность имеет место, необходимо явно указать команде *man* раздел, в котором следует искать справочную информацию:

man 1 time

Т.е. 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, данная страница документации и др.
8	Команды управления системой	Команды типа **mount**(8), которые может использовать только *root*.
9	Процедуры ядра	Это устаревший раздел руководства. В свое время появилась идея держать документацию о ядре Linux в этом разделе. Однако, в то время документация о ядре была неполной и, по большей части, устаревшей. Существуют значительно более удобные источники информации для разработчиков ядра.

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 т.н. псевдопараллельного режима, когда операционная система последовательно выделяет для каждого процессора определенное количество квантов процессорного времени, тем самым достигая иллюзии одновременной работы 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.

Linux

Новый процесс в Linux создается при помощи системного вызова fork(). Данный вызов создает дубликат процесса-родителя; выполнение процесса-потомка начинается со следующего после fork() оператора. Функция fork() возвращает родительскому процессу идентификатор созданного процесса-потомка, самому потомку – 0, а в случае ошибки – -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 произошла ошибка.

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

При выборе конкретной функции надо иметь в виду следующие моменты:

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 дочерний процесс завершен, но ро-

дитель не вызвал какую-либо из вышеназванных функций, дочерний становится т.н. «зомби»-процессом. Несмотря на то, что он ничего не выполняет, он явно присутствует в системе. Если родитель завершился, не вызвав wait(), новым родителем всем его потомкам назначается init, корневой процесс ОС Unix. Init автоматически производит очистку «зомби»-процессов. Те не менее, хорошим стилем программирования считается контролировать жизненный цикл дочерних процессов из родительского, не перекладывая эту задачу на систему.

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:

1) Входная функция первичного потока возвращает управление;

2) Один из потоков процесса вызывает ExitProcess();

3) Любой поток любого процесса вызывает TerminateProcess();

4) 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().

Задание

Разработать консольное приложение, в котором базовый процесс порождает дочерний. Для каждого процесса предусмотрена своя область вывода, в которой он выводит текущее системное время. Работа выполняется в двух вариантах: под Linux и Windows. 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. К примеру, существует понятие *состояния гонки (race condition)*, при котором один процесс должен дождаться, пока другой не выполнит определенную операцию. Это означает, что должна выполняться строгая последовательность выполнения операций. Такое возможно, когда, к примеру, один процесс должен подготовить ресурс к использованию другим процессом.

Бывают также ситуации, когда 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 процесс перед выполнением критической операции должен заблокировать доступ к ресурсу до ее окончания. Только после этого другой процесс имеет право произвести свои действия с этим ресурсом.

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

1) Проверить доступность ресурса;
2) Если ресурс доступен, заблокировать его. В противном случае подождать до разблокирования его другим процессом;
3) Выполнить необходимую операцию;
4) Разблокировать ресурс.

Как в Linux, так и в Windows существует большое количество способов синхронизации процессов. Мы рассмотрим два способа на каждую ОС.

Linux

В качестве инструментов синхронизации в Linux наиболее распространены 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 выражается в том, что в некоторых ситуациях процесс может не получить сгенерированный сигнал. Кроме этого, у процессов довольно ограничены возможности по управлению ненадежными сигналами.

Одна из проблем в ненадежных сигналах выражается 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, что любой сигнал будет доставлен процессу. При этом процесс имеет возможность указать, какие сигналы он будет перехватывать, а какие игнорировать, т.е. составить т.н. *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 параметру в функции *signal*.

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

Рассмотрим теперь, что представляет собой структура *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 управления обработчиком 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 позволяют реализовать произвольную логику взаимодействия процессов, инициируя срабатывание функций-обработчиков в нужное время.

Семафоры – инструмент, разработанный исключительно для реализации механизма критических секций в межпроцессном взаимодействии (IPC).

Семафор представляет собой глобальный системный счетчик, теоретически доступный всем процессам. При этом оперирует не одним семафором, а 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. В этом случае обычно *nsems* 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;
 };
```

Sem_num – номер семафора в наборе, над которым производится операция (начиная с нуля). *Sem_op* – сама операция. При этом действие этого параметра отличается в каждом из нижеперечисленных случаев:

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

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

- удаление набора семафоров, вызвавших блокировку процесса;
- получение 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.

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: свободно или занято (сигнальное и несигнальное). Событие может быть одного 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;
- *flnitialState* – начальное состояние события (*TRUE* – сигнальное, *FALSE* - несигнальное);
- *pszName* – уникальное имя-идентификатор события.

Получить доступ к событию из другого процесса можно следующими способами:

- вызвав функцию CreateEvent() с тем же именем, которое было присвоено событию первым процессом;
- унаследовав дескриптор;
- применив функцию DuplicateHandle();
- вызвав функцию OpenEvent() с указанием имени существующего события.

Управлять состоянием события позволяют две функции: SetEvent() переводит его в свободное состояние, ResetEvent() – в занятое. Для ожидания события используется функция WaitForSingleObject(). Если событие создано с флагом автосброса, то при успешном окончании ожидания событие автоматически переведется в занятое 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 ими.

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 OpenSemaphore().

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

Дождаться освобождения ресурса (ненулевого состояния семафора) можно при помощи функции WaitForSingleObject(). При этом счетчик ресурсов автоматически будет уменьшен на единицу.

Задание

Начальный процесс является управляющим. Он принимает поток ввода с клавиатуры и контролирует дочерние процессы. По нажатию 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. Выполняется в двух вариантах: под Linux и Windows. В качества метода синхронизации использовать сигналы/события.

Лабораторная работа №3
Взаимодействие процессов

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

При взаимодействии нескольких параллельных процессов часто возникает необходимость обмена данными между ними. При этом встает следующая проблема: каждый процесс имеет собственное адресное пространство, и 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. Под каждую из рассматриваемых операционных систем существует большое количество методов и способов решения данной проблемы, поэтому подробно остановимся на наиболее часто используемых.

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 двусторонней связи приходилось организовывать два канала. Несмотря на то, что практически все современные системы такую возможность предоставляют, необходимо быть осторожным при разработке приложений и 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 - -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 каналы можно использовать только с процессами, имеющими общего предка. Обычно канал создается до вызова *fork()* и используется затем в обоих процессах-клонах.

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

Сегменты разделяемой памяти

Каналы позволяют связать два процесса. Если взаимодействующих процессов больше, либо по каким-то причинам использование каналов исключается, можно использовать сегменты разделяемой памяти. При этом выделяется область памяти, идентифицируемая уникальным целочисленным ключом и доступная всем процессам, которые пожелают обратиться 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 функция возвращает идентификатор созданного сегмента, в противном случае - -1. Так же, как и с семафорами, можно получить доступ к созданной ранее области разделяемой памяти, через ключ *key*.

Для непосредственного использования разделяемой памяти необходимо получить адрес сегмента 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. Все данные, записанные через этот указатель, будут доступны процессам, использующим данный сегмент. После завершения работы необходимо вызвать функцию *shmdt()* для отсоединения сегмента.

Для расширенных операций над сегментом разделяемой памяти используется функция *shmctl()*. В спектре ее возможностей получение и установка параметров сегмента, его удаление, а также блокирование/разблокирование. Операции по блокировке сегмента предоставляют не все Unix-системы; на данный момент эта возможность присутствует только в Linux и Solaris.

Удаление сегмента производится посредством вызова функции *shmctl()* с параметром *IPC_RMID*. При этом сегмент удаляется, когда все процессы, его использующие, отсоединяют его. Во избежание накапливания неиспользуемых сегментов разделяемой памяти, каждый сегмент должен быть вручную удален после завершения работы с ним. Использование функций *exit()* и *exec()* отсоединяет сегмент, но не удаляет его.

Windows

Среди наиболее часто употребляемых механизмов передачи данных между процессами в Windows можно выделить каналы и файловые проекции

Каналы

Подобно каналам в Linux, в Windows каналы также связывают два процесса. Процесс, создающий канал, обычно называют *сервером*, а процесс, использующий этот канал – *клиентом.* Каналы в Windows бывают двух типов: анонимные и именованные. Первые универсальны и быстры, однако процессу-клиенту сложнее получить дескриптор канала, они не поддерживают дуплексную передачу данных и не работают в сетях. Именованные каналы свободны от

этих 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.

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

Функция создает анонимный канал с параметрами:

- *hReadPipe* – дескриптор чтения;
- *hWritePipe* – дескрпитор записи;
- *lpPipeAttributes* – атрибуты защиты;
- *nSize* – количество байт, резервируемых для 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, оно представляет собой примерно следующую строку: «*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* – развер входного буфера;
- *nDefaultTimeOut* – время ожидания в миллисекундах;
- *lpSecurityAttributes* – атрибуты защиты.

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

Отображаемая память

Еще один способ передать данные между процессами – использование файлов, отображаемых в память. При этом один процесс создает специальный объект, «файловую проекцию», 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 некоем специальном файле, проек-

ция 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. Это производится следующей функцией:

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 выделенной другим процессом памяти. Здесь нужно учитывать, что файловая проекция уникально идентифицируется именем, указанным в функции *CreateFileMapping()*. Для получения дескриптора проекции необходимо воспользоваться функцией *OpenFileMapping()* и по полученному дескриптору при помощи функции *MapViewOfFile()* получить указатель на искомую область памяти.

Задание

Создаются два процесса: клиентский и серверный. Серверный процесс ждет ввода пользователем текстовой строки и по нажатию клавиши Enter производит следующие действия:

- клиентский процесс уведомляется 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 выводит полученную строку на экран и уведомляет серверный процесс об успешном получении строки;
- серверный процесс ожидает ввода следующей строки 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.

Работа выполняется в двух вариантах: под Linux и Windows.

Лабораторная работа №4
Работа с потоками

Цель работы: Научиться создавать, уничтожать и управлять вычислительными потоками.

Понятие потока несколько различается в системах Linux и Windows. Рассмотрим эти концепции подробнее.

Linux

В Linux поток представляет собой облегченную версию процесса (*lightweight process*). Потоки можно представить как особые процессы, принадлежащие родительскому процессу и разделяющие с ним адресное пространство, файловые дескрипторы и обработчики сигналов. Такие потоки являются гораздо более легковесными для системы, и как следствие быстрее выполняется переключение между ними, увеличивается быстродействие. Однако использова-

ние общего адресного пространства порождает ряд проблем синхронизации и некоторые трудности отладки.

У любого потока имеется так называемся потоковая функция, иногда именуемая телом потока. Это функция, указываемая потоку при создании и содержащая код, который поток должен выполнить. При выходе из этой функции поток завершается.

В Linux существует множество библиотек, предоставляющих интерфейс к системным вызовам, управляющим потоками. Стандартом де-факто считается 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* – по этому указателю 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 необходимо передать в поток какие-то данные при его создании, их можно записать в отдельную область памяти и передать указатель на нее этим параметром, который является аргументом потоковой функции.

Для завершения потока используйте функцию *pthread_exit()*.

Поскольку потоки выполняются параллельно, открытыми остаются вопросы синхронизации. Первым средством решения этой проблемы является использование функции *pthread_join()*. Данная функция принимает в качестве первого аргумента идентификатор потока и приостанавливает выполнение текущего потока, пока не будет завершен поток с указанным идентификатором.

Использование этой функции – инструмент эффективный, но грубый и не всегда подходящий для тонкой синхронизации, когда оба потока должны, не завершаясь, синхронизировать свои действия. Для «умной» синхронизации используются специальные объекты, *мьютексы*.

Мьютекс представляет собой бинарный флаг, который имеет два состояния: «свободно» и «занято». Общая процедура синхронизации, описанной в л.р. №2, остается актуальной и для мьютексов. 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 и после этого занимает его;
- *pthread_mutex_unlock* – установка мьютекса в положение «свободно»;

- *pthread_mutex_trylock* – функция аналогична *pthread_mutex_lock*, только при занятости мьютекса не ждет его освобождения, а тут же возвращает значение *EBUSY*;
- *pthread_mutex_destroy* – уничтожение мьютекса.

Windows

В Windows ситуация несколько отлична от Linux. Здесь процесс на самом деле не выполняет никакого кода, являясь лишь контейнером для вычислительных потоков. Потоки - 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. В процессе должен существовать 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 работает со следующими параметрами:

- *lpThreadAttributes* – атрибуты безопасности потока;
- *dwStackSize* – размер стека, выделяемого под поток;
- *lpStartAddress* – адрес потоковой функции. Она должна следовать прототипу *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 четырех способов:

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

Стоит отметить, что использование функции *CreateThread()* может привести к некорректной работе приложения, если используются глобальные функции и переменные стандартной библиотеки C/C++ (*errno, strok, strerror* и т.д.). Если их использование необходимо, то рекомендуется прибегнуть к функциям _beginthreadex и _endthreadex, которые реализуют аналогичную функциональность, но безопасны в работе.

Для решения проблемы синхронизации в Windows также существуют мьютексы. Однако они представляют собой, по сути, бинарные семафоры и предназначены в первую очередь для взаимодействия процессов. Использование подобных глобальных объектов в многопоточном приложении не всегда оправдано.[2]

Более подходящим механизмом для синхронизации потоков являются критические секции. Логика их использования на самом деле очень похожа на работу мьютексов в pthreads. Существуют пять основных функций для работы с критическими секциями:

- *InitializeCriticalSection* – создание и инициализация объекта «критическая секция»;
- *EnterCriticalSection* – вход в критическую секцию. Это действие аналогично блокировке мьютекса. Если блокируемая критическая секция уже кем-то используется, функция ждет ее освобождения;
- *LeaveCriticalSection* – покинуть критическую секцию;
- *TryEnterCriticalSection* – попробовать войти в критическую секцию. Функция аналогична *EnterCriticalSection()*. Если критическая секция занята, возвращается ноль, в противном случае – ненулевое значение;
- *DeleteCriticalSection* – удалить критическую секцию.

Задание

Задание аналогично лабораторной работе №2, но с реализацией с помощью потоков. Работа выполняется в двух вариантах: под Linux и Windows.

Лабораторная работа №5
Асинхронные файловые операции. Динамические библиотеки

[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 они используются только для ожидания освобождения секции, а в большинстве случаев работа функций *EnterCriticalSection()* и *LeaveCriticalSection()* сводится к простому приращению счетчиков.

Цель работы: Научиться осуществлять асинхронные файловые операции. Разобраться с понятием «динамическая библиотека», научиться создавать их и динамически использовать в приложениях.

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 окончанию 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 запускается в фоновом режиме. Это означает, что функция чтения/записи возвращает управление немедленно, в то время как сама операция может выполняться еще некоторое время после этого в фоне. Асинхронные файловые операции, как правило, ускоряют работу приложения, однако порождают следующую проблему: в большинстве случаев необходимо убедиться, что операция завершена и все данные переданы. Поскольку реализация фонового режима программе 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.

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 осуществлена 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 чтения/записи данных используются функции *aio_read()* и *aio_write()*. Обе они принимают адрес сформированной структуры *aiocb* в качестве единственного аргумента. В нормальной ситуации функции возвращают 0, если запрос был принят и поставлен в очередь. 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 она все еще продолжает выполнение, возвращаемое значение будет равно *EINPROGRESS*.

Windows

Использование асинхронных файловых операций в Windows схоже с библиотекой *aio*. Аналогом структуры *aiocb* 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 чтения/записи могут использоваться два семейства функций: стандартные и расширенные.

Во-первых, классические функции *ReadFile()* и *WriteFile()* замечательно умеют 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. Перед началом операции функции *ReadFile()/WriteFile()* переводят его в несигнальное состояние. Переход же события в сигнальное состояние свидетельствует о том, что операция была завершена.

Иной случай – 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.

Динамические библиотеки

В программировании весьма часто возникает ситуация, когда запущенные приложения используют одни и те же функции или участки кода. Многократно продублированные таким образом, они 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 *дина-*

мически подключаемыми. Рассмотрим детали реализации в различных операционных системах.

Linux

В Linux динамическая библиотека создается полностью аналогично обычному бинарному исполняемому файлу:

1) компиляция исходного кода в объектный файл;

2) линковка объектного файла в результирующий формат.

Указать компилятору, что на выходе он должен породить динамическую библиотеку, можно при помощи ключа –shared:

gcc –shared –o mylibrary.so mylibrary.o

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

Подключение динамической библиотеки к приложению можно осуществить двумя способами. Во-первых, ее можно указать при линковке приложения с помощью ключа gcc –l. В этом случае система при загрузке приложения будет автоматически подгружать указанную библиотеку. Однако нередки случаи, когда для работы приложения нет необходимости все время иметь в памяти загруженную библиотеку либо из библиотеки используется лишь небольшое количество функций, в то время как остальные только занимают место в памяти. 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. Он 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()*.

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 необходи-

мости 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 *FreeLibrary()*.

Задание

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

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

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

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

4) поток-писатель уведомляет читателя 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.

При программировании самых 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 возни-

кает ситуация, когда стандартного функционала менеджера памяти недостаточно. К примеру, может потребоваться индексирование выделенных участков памяти, их дефрагментация, сборка мусора и т.д. В этих случаях наиболее логично написание собственного менеджера памяти.

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.

В качестве составных компонент менеджера памяти традиционно выделяют следующее:

- 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 (автоматическая сборка мусора, например).

Для выполнения задания необходимо рассмотреть следующие понятия:

1) *Сборка мусора* – автоматическое освобождение менеджером памяти неиспользуемых участков памяти принудительно или в фоновом режиме. Подразумевает отсутствие функции *free()* (оператора *delete*) или совместную с ней работу. Как правило, неиспользуемым считается участок памяти, на который отсутствуют ссылки.

2) *Дефрагментация* – процесс упорядочивания выделенных областей памяти и устранение пустого неиспользуемого пространства между ними. Фрагментация оперативной памяти возможна при последовательном выделении и освобождении памяти. Это означает, что теоретически возможна ситуация, когда общего объема свободной памяти достаточно для выделения, однако вся она сосредоточена в небольших пустых областях между выделенными участками. В такой ситуации невозможно выделить непрерывный участок памяти требуемого размера, несмотря на то, что в принципе нужный объем памяти помечен как свободный.

3) *Свопинг* – процесс сброса на жесткий диск участков памяти, используемых менее всего, 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 считать его с жесткого диска и вновь выделить для него память нужного объема.

Задание

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.

Задание выполняется в одном варианте под любую операционную систему на выбор студента.

Лабораторная работа №7
Эмулятор файловой системы

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

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 встают следующие проблемы:

- необходимость поддержки иерархии размещения файлов и каталогов. Как правило, файлы хранятся в древовидной системе каталогов, которую надо спроецировать в физическое представление на конкретном носителе;
- алгоритмы добавления, удаления, модификации файлов;
- быстрый доступ 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:

- Для носителей с произвольным доступом (жестким дискам). Примеры: ext2, ext3, ReiserFS, FAT32, NTFS, XFS... В Unix-системах обычно применяются первые три, использование FAT32 и NTFS характерно для ОС семейства Windows.
- Для носителей с последовательным доступом (магнитные ленты): QIC и др.
- 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 ФС;
- *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, количество 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. Эти блоки 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 представляет физическое представление файла на жестком диске с учетом того, что файл может занимать довольно большой объем, т.е. дробиться на небольшие блоки данных.

Каждый дескриптор содержит 13 указателей. Первые 10 указателей непосредственно ссылаются на блоки данных файла. Если файл большего размера - 11-ый указатель ссылается на первый косвенный блок (*indirection block*) из 128 (256) ссылок на блоки данных. Если и этого недостаточно, 12-ый указатель ссылается на дважды косвенный блок, содержащий 128 (256) ссылок на косвенные блоки. Наконец последний, 13-ый 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 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 полной картины о внутреннем устройстве файловых систем рекомендуется также ознакомиться с системами FAT2 и NTFS.

Задание

Разработать собственную файловую систему. Физический носитель в данном случае эмулируется файлом фиксированного размера. Архитектура файловой системы остается на усмотрение студента. В конечном результате 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.