МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ «АРХИТЕКТУРА И АЛГОРИТМИЧЕСКИЕ ОСНОВЫ СИСТЕМНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве практикума для обучающихся по основной образовательной программе высшего образования по направлению подготовки 03.03.01 Прикладные математика и физика

Составитель *А.Ю. Привалов*

САМАРА
Издательство Самарского университета
2022

ISBN 978-5-7883-1761-8

УДК 72(075)+004.9(075)
ББК 85.11я7+32.97я7
З-151

Составитель ***А.Ю. Привалов***

Рецензенты: д-р техн. наук, проф. А. В. И в а щ е н к о;
д-р техн. наук, проф. С. А. П р о х о р о в

З-151 Задания для лабораторных работ по дисциплине «Архитектура и алгоритмические основы системного программного обеспечения»: практикум / Составитель *А.Ю. Привалов*; Министерство науки и высшего образования Российской Федерации, Самарский университет. – Самара: Издательство Самарского университета, 2021. – 1 CD-ROM (1,6 Мб). – Загл. с титул. экрана. – Текст: электронный.

ISBN 978-5-7883-1761-8

Практикум содержит краткие теоретические сведения и задания для лабораторных работ по дисциплине «Архитектура и алгоритмические основы системного программного обеспечения» для обучающихся по направлению подготовки 03.03.01 Прикладные математика и физика.

Разработан на кафедре «Прикладных математики и физики» Самарского университета.

УДК 72(075)+004.9(075)
ББК 85.11я7+32.97я7

Минимальные системные требования:
PC, процессор Pentium, 160 МГц;
Microsoft Windows XP; мышь;
дисковод CD-ROM; Adobe Acrobat Reader.

Подписано для тиражирования 04.07.2022.
Объем издания 1,6 Мб.
Количество носителей 1 диск.
Тираж 10 дисков.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С. П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1

1. Начальные сведения о файловой системе Linux (часть 1).
2. Основы работы с файловой системой в командной оболочке.
3. Основы работы с компилятором gcc.
4. Системные вызовы и функции для работы с файловой системой.

Начальные сведения о файловой системе Linux (часть 1)

Вся информация, хранящаяся в компьютере, организована в **файлы** – *именованные области данных на носителе информации*. Также, во многих операционных системах (в том числе в Linux) в виде работы с файлами организован и доступ к другим ресурсам, например, устройствам ввода-вывода.

Все файлы, доступные в операционной системе Linux, объединяются в единую древовидную (имеющую вид направленного дерева – *связного направленного графа без циклов*) логическую структуру. Ветками этого дерева (*узлами, из которых могут исходить дуги*) являются специальные файлы, называемые **директориями** (**каталогами, папками**). Из них могут исходить дуги к другим файлам – как к другим директориям, так и к файлам, хранящим различные данные (**регулярным файлам).** Регулярные файлы являются листьями дерева – *узлами, из которых не исходят дуги*. Принято говорить, что **директория включает в себя** все файлы, к которым из неё исходят дуги (или, что все такие **файлы входят** в данную директорию, **находятся** в данной директории).

Каждому файлу (как регулярному, так и директории) должно быть присвоено имя. Оно не должно быть длиннее 255 символов (в других версиях 13 символов), не должно содержать символ с нулевым кодом (NUL) и символ ‘/’ (*слэш*). Также не рекомендуется использовать в именах файлов символы ‘*****’, ‘**?**’, ‘\’ (*обратный слэш*), кавычки и пробел. Файлы должны иметь уникальное имя в пределах той директории, в которую они входят. В отличие от Windows, Linux различает регистр символов, входящих в имя файла.

Особым случаем является корень дерева – единственная директория, не входящая ни в какую другую. Её имя всегда «/» (единственное исключение из правила именования файлов). Если имя файла начинается с символа ‘**.**’, такой файл считается **скрытым**.

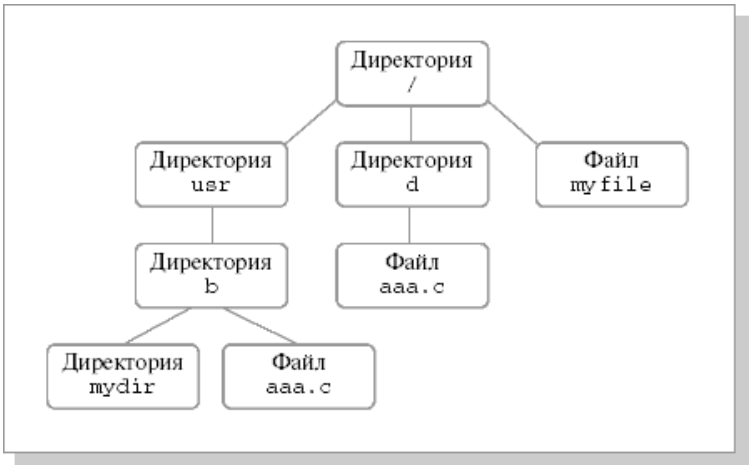

Рис. 1. Пример дерева файловой системы (https://www.intuit.ru/studies/courses/2249/52/lecture/1550?page=3)

Полное имя файла – это последовательность имён узлов на пути от корневой директории к этому файлу, с последним именем на этом пути – именем этого файла, при этом все имена некорневых директорий и имя файла разделяются символом ‘/’. Пример: если дерево файловой системы имеет вид, изображённый на рис.1, то полное имя файла myfile, находящегося в корневой директории – это /myfile, полное имя файла aaa.c, находящегося в директории b – это /usr/b/aaa.c, а полное имя файла aaa.c, находящегося в директории d – это /d/aaa.c.

В каждый момент работы пользователя в системе, одна из директорий считается для него **текущей** (**рабочей**). Относительно рабочей директории определяется **относительное имя файла**. Оно строится по последовательности узлов на пути от рабочей директории до этого файла по следующим правилам – начинается с «**./**» (но это необязательная часть, может быть опущена), а далее, если путь идёт по направлению к корню (против направления дуг), то вместо имени узла ставится «**../**», а когда путь пойдёт по направлению дуг (от корня), то ставится имя узла, отделённое от следующего имени символом ‘/’ (как в полном имени файла). Пример: если текущей директорией в файловой системе, изображённой на рис.1, является директория mydir, то относительное имя файла aaa.c из директории b будет **./..**/aaa.c (сокращённая форма **..**/aaa.c), относительное имя файла aaa.c из директории d будет **./../../..**/d/aaa.c (сокращённая форма **../../..**/d/aaa.c), а относительное имя файла myfile, лежащего в корневой директории будет **./../../..**/myfile (сокращённая форма **../../..**/myfile).

Кроме регулярных файлов, в директориях могут находиться и файлы других типов, они будут изучены позже, по мере знакомства с командами для работы с файловой системой в командной оболочке Linux.

У каждого файла в Linux, кроме имени, есть ряд атрибутов:

- тип файла (в отличие от Windows, в Linux тип определяется не по расширению, другим способом, будет изучен позже);
- идентификаторы пользователя-владельца и группы-владельца;
- наборы прав доступа к файлу для владельца, для членов группы-владельца и для всех остальных пользователей;
- размер файла (только для директорий, регулярных файлов и файлов типа “связь”, о которых позже);
- дата и время последнего доступа к файлу;
- дата и время последней модификации файла;
- дата и время модификации т.н. “индексного узла файла” (будет изучен позже);
- количество т.н. “жёстких связей файла” (будут изучены позже).

Все наборы прав доступа к файлу состоят из трёх элементов (битовых индикаторов):

- индикатор права на чтение (обозначается r – от слова read);
- индикатор права на модификацию (обозначается w – от слова write);
- индикатор права на исполнение (обозначается x – от слова execute).

Для регулярных файлов смысл этих прав совпадает с названием. При этом, право на исполнение имеет смысл только для регулярных файлов, содержащих программы.

Для директорий право на чтение означает право читать имена (и только имена) файлов, входящих в неё. Право записи позволяет изменять её содержимое, то есть создавать, удалять и переименовывать файлы в ней. Право исполнения для директории означает возможность

просматривать содержимое входящих в неё файлов, изменять это содержимое, запускать на исполнение, если это файлы программ (всё это при условии, что соответствующие действия позволяют права на сам файл). Также право на исполнение директории позволяет получать информацию об атрибутах входящих в неё файлов. Кроме того, чтобы сделать директорию рабочей, право на исполнение должно быть у пользователя и для неё, и для всех директорий на пути к ней.

Основы работы с файловой системой в командной оболочке

С появлением **графических оболочек** и **графических сред рабочего стола** всё основное взаимодействие с операционной системой в Linux стало очень просто производить с использованием мыши и почти без использования клавиатуры. Так, популярная среда рабочего стола Xfce включает в себя файловый менеджер, оконный менеджер, панель задач, текстовый редактор, менеджер настроек и другие компоненты, позволяющие работать практически со всеми механизмами операционной системы. Освоение работы с современными графическими средами несложно, и потому предлагается для самостоятельной работы. В цикле лабораторных работ внимание будет уделяться основам базового способа работы с операционной системой Linux – **командной оболочке** (**командной строке**).

Из графической среды рабочего стола (которая, обычно, загружается при старте операционной системы по умолчанию) командная оболочка запускается, как правило, выбором пункта меню “эмуляция терминала”, “терминал” или “консоль” (В Xfce – “эмуляция терминала”). При этом открывается текстовое окно с текущей строкой ввода, обозначаемой приглашением и стоящим за ним курсором. Например, оно может выглядеть так:

```
[student@stud2 ~]$_
```

Здесь в квадратных скобках сначала стоит имя пользователя (student), после символа ‘@‘ имя компьютера (stud2), тильда после пробела – знак, указывающий, что рабочей директорией в данный момент является т.н. **домашняя директория** данного пользователя, то есть директория (своя для каждого пользователя), куда он всегда попадает сразу после входа в систему. Доллар после закрывающей скобки перед курсором, стоящим на первом символе ввода, говорит о том, что это обычный пользователь. Если бы вместо ‘$’ стоял бы ‘#’, это означало бы, что этот пользователь обладает правами администратора (суперпользователя).

Работа с операционной системой с помощью командной оболочки производится путём подачи ей команд, представляющих собой текстовые строки следующего формата:

```
команда [-параметры] [аргументы]
```

Здесь и далее квадратные скобки не являются частью формата команды (если специально не оговорено обратное), а только обозначают, что параметры и аргументы могут отсутствовать. Обязательным является только имя команды. Параметры изменяют поведение команды, а аргументы задают объекты, на которые команда действует. Примеры простых команд без параметров и аргументов, приведены в табл. 1 (проверьте на практике как они работают), другие команды – в табл. 2-4.

Таблица 1

Команда	Описание и комментарии
date	вывод текущих даты и времени
pwd	вывод полного имени текущего рабочего каталога (print working directory)
exit	окончание работы с командной оболочкой

Универсальный справочник по командам командной оболочки

Таблица 2

Команда	Описание и комментарии
man команда	вывод на экран информации о заданной команде. Если информация занимает несколько страниц, прокрутка страницы – пробел, прокрутка одной строки – enter, выход из режима просмотра – q

Основные команды навигации по файловой системе

Команда **cd** – смена рабочей директории (change directory)

Таблица 3

Команда	Описание и комментарии
Общий вид	
cd dirname	смена рабочей директории на директорию с указанным именем. Имя может быть полным или относительным
Примеры	
cd /usr	текущей станет директория usr, которая является дочерней к корневой (т.е. входит в корневую директорию файловой системы, здесь использовано полное имя)
cd tmp	если в текущую директорию входит директория tmp, то она станет текущей (здесь использовано относительное имя, необязательные начальные символы "**./**" опущены)
cd ..	рабочей станет директория, родительская по отношению к текущей (то есть та, в которую входит текущая; здесь тоже используется относительное имя)
Специальные случаи (полезные сокращения)	
cd	при использовании без аргументов, рабочая директория меняется на домашнюю директорию пользователя
cd -	если в качестве аргумента стоит минус, то рабочая директория меняется на предыдущую рабочую директорию
cd ~username	рабочей директорией становится домашняя директория пользователя с именем username

Команда **ls** – вывод содержимого директории (list)

Таблица 4

Команда	Описание и комментарии
Использование без параметров	
ls dir1 [dir2…]	вывод списка имён файлов, находящихся в директориях с заданными именами (полными или относительными)
ls	без аргументов, вывод списка имён файлов, входящих в текущую рабочую директорию
Команда с часто используемыми параметрами	
ls –a [dir1 dir2…]	при наличии параметра –a (all) выводятся имена всех файлов, включая скрытые; при отсутствии параметра имена скрытых файлов не выводятся
ls –l [dir1 dir2…]	при наличии параметра –l (long), кроме имён, для каждого файла выводятся все значения его атрибутов (см. рис.2)
Примечание: при наличии нескольких параметров в одной команде, их можно объединять для краткости в одну запись, так **-a -l** эквивалентно **-al** или **-la**.	

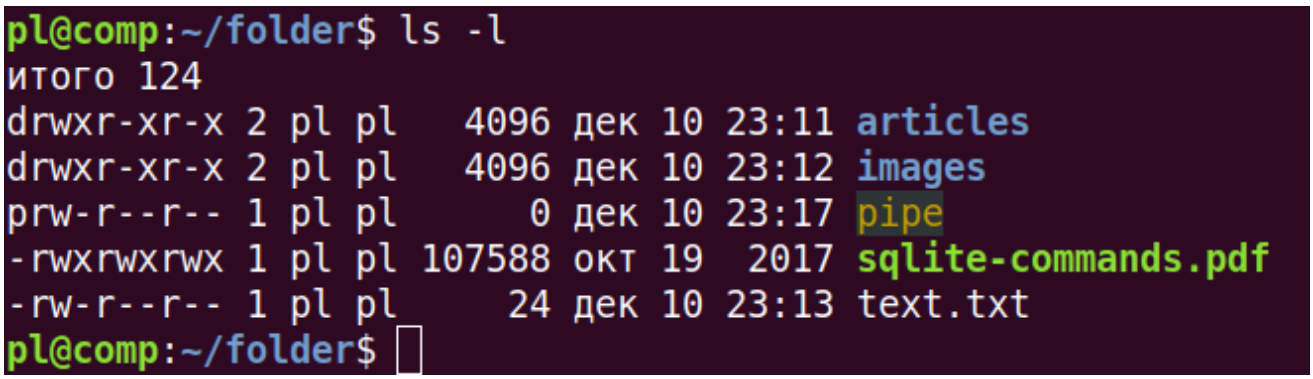

Рис. 2. Пример вывода информации с атрибутами файлов (https://younglinux.info/bash/ls)

Формат вывода информации об атрибутах файлов команды ls –l:

- первая строка – размер директории в дисковых блоках (будет рассмотрен позже, в примере на рис. 2 – размер текущей директории 124 дисковых блока); далее информация о каждом файле (по строке на файл), слева направо:

- первый символ – тип файла (регулярный файл **–**, директория **d**, остальные возможные типы будут рассмотрены позже);

- далее девять идущих подряд символов – права доступа для владельца файла, группы владельца файла и всех остальных пользователей;

- целое число – количество жёстких связей файла (будет рассмотрено позже);

- имя владельца файла;

- имя группы владельца файла;

- целое число – размер файла на диске в байтах;

- дата и время создания файла;

- имя файла.

Шаблоны имён файлов для команд манипуляций с файлами

Применяются для задания набора имён файлов во многих командах операционной системы. При использовании шаблона просматривается вся совокупность имён файлов, находящихся в операционной системе, и те имена, которые удовлетворяют шаблону, включаются в набор. Метасимволы шаблонов приведены в табл. 5.

Таблица 5

Метасимвол	Описание и комментарии
`*`	соответствует любой последовательности символов, включая пустую
`?`	соответствует любому одиночному символу
`[ ]`	соответствует любому символу, заключённому в скобки; пара символов, разделённых знаком минус, задаёт диапазон символов (в порядке их следования в таблице ASCII)
Примеры	
`*.c`	все имена в текущей директории, оканчивающиеся на .c ; для использования шаблона * в начале имени файла существует следующее исключение: имена файлов, начинающиеся с точки (т.е. скрытые), считаются не удовлетворяющими шаблону
`a??`	все имена в текущей директории, состоящие из трёх букв и начинающиеся на a
`[a-d]*`	все имена в текущей директории, начинающиеся с букв a, b, c, d

При задании имён файлов в приведённых далее командах можно использовать шаблоны. Имена файлов могут быть как относительными, так и полными.

Команды манипуляций с файлами

Основные команды манипуляций с файлами приведены в табл. 6.

Таблица 6

Команда	Описание и комментарии
Копирование (copy)	
cp file1 file2	Копирует содержимое файла с именем **file1** в файл **file2**
cp file1 [file2… fileN] dir	Копирует файлы с именами **file1, file2 … fileN** в существующую директорию **dir**
cp -r dir1 [dir2… dirN] dir	Копирует содержимое директорий **dir1, dir2 … dirN** в существующую директорию **dir** рекурсивно, т.е. со всеми их поддиректориями
Перемещение и переименование (move)	
mv file1 file2	Перемещает или переименовывает файл **file1** в **file2**
mv file1 [file2… fileN] dir	Перемещает файлы **file1, file2 … fileN** в существующую директорию **dir**
Удаление (remove)	
rm file1 [file2… fileN]	Удаление файлов **file1, file2 … fileN**
rm –r dir1 [dir2… dirN]	Рекурсивное удаление директорий **dir1, dir2 … dirN**
Создание директории (make directory)	
mkdir dir	Создание директории с именем **dir**

Команды манипуляций с атрибутами файла

Основные команды манипуляций с атрибутами файла приведены в табл. 7.

Таблица 7

Команда	Описание и комментарии
file file1 [file2 … fileN]	Определение типов файлов **file1 [file2 … fileN]**
chown owner file1 [file2 … fileN]	Смена владельца файлов **file1 [file2 … fileN]** с текущего на **owner.** Может быть произведена только текущим владельцем файлов или системным администратором. Новый владелец может быть задан как с помощью системного имени, так и с помощью системного идентификатора
chgrp group file1 [file2 … fileN]	Смена группы владельцев файлов **file1 [file2 … fileN]** с текущего на **group.** Может быть произведена только владельцем файлов или системным администратором. Новая группа может быть задана как с помощью имени группы, так и с помощью идентификатора группы
Изменение прав доступа (использование символьной нотации)	
chmod [whom] +[perm] file1 [file2… fileN]	Добавление прав **perm** категории **whom** для файлов **file1 [file2 … fileN].** Обозначение категорий (**whom**): **u** – владелец файла, **g** – группа владельцев файла, **o** – все остальные пользователи, **a** – все пользователи (владелец, группа и остальные), используется по умолчанию (при отсутствии указания категории). Обозначение прав (**perm**): **r –** право на чтение, **w –** право на запись,

Команда	Описание и комментарии
	x – право на исполнение. Отсутствие **perm** обозначает все права одновременно
chmod [whom] -[perm] file1 [file2… fileN]	Удаление прав **perm** категории **whom** для файлов **file1 [file2 … fileN].**
chmod [whom] =[perm] file1 [file2… fileN]	Замена прав доступа категории **whom** для файлов **file1 [file2 … fileN],** то есть, удаление всех существующих и добавление указанных в команде (**perm**)
Просмотр и установка маски прав доступа файла по умолчанию	
umask	Вывод в восьмеричном виде битовой маски прав доступа файла по умолчанию
umask value	Установка маски прав доступа файла по умолчанию, равной **value** (восьмеричное)

Маска прав доступа используется всякий раз, когда создаётся новый файл – в текстовом редакторе, в процессе компиляции программы и т.д. Маска представляет собой девятибитное двоичное число, в котором старшие три бита отведены для прав доступа владельца файла, средние три бита – для прав доступа группы владельцев, и младшие три бита – для прав всех остальных пользователей. **Установка в 1 соответствующего бита маски означает отсутствие соответствующего права.**

Для манипуляций с битами маски используются восьмеричные представления двоичных чисел. А именно:

0400 – запрет чтения для владельца (старший бит 1, остальные 0),

0200 – запрет записи для владельца (второй справа бит 1, остальные 0),

0100 – запрет исполнения для владельца (третий справа бит 1, остальные 0),

0040 – запрет чтения для группы владельцев,

0020 – запрет записи для группы владельцев,

0010 – запрет исполнения для группы владельцев,

0004 – запрет чтения для всех остальных пользователей,

0002 – запрет записи для всех остальных пользователей,

0001 – запрет исполнения для всех остальных пользователей.

Самый левый ноль – это знак, обозначающий восьмеричную систему счисления, сами же числа – трёхзначные. Значение маски равно сумме всех соответствующих установленным битам значений, указанных выше. Маска прав доступа не сохраняется между сеансами работы в системе, и если маска, устанавливаемая системой при загрузке, вас не устраивает, её надо устанавливать каждый раз вручную.

Основы работы с компилятором gcc

GCC – это свободно доступный оптимизирующий компилятор для языков C и C++. Программа **gcc**, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, **gcc** запускает необходимые препроцессоры, компиляторы, линкеры.

Файлы с расширением **.cc** или **.C** рассматриваются, как файлы на языке C++, файлы с расширением **.c** как программы на языке C, файлы с расширением .hpp как заголовочные для C++, с расширением .h как заголовочные для c, файлы с расширением **.o** считаются

объектными, с расширением **.a** – библиотеками компилятора (в отличие от операционной системы, gcc различает тип файла по расширению). Чтобы получить исполняемый файл из файла программы на языке С, можно выполнить команду

```
gcc file.c
```

Тогда в рабочей директории должен появиться файл **a.out** – это файл с исполняемым кодом программы (и именем по умолчанию, которое когда-то означало assembler output), который можно запускать на исполнение:

```
./a.out
```

В данном случае в относительном имени файла нельзя опускать имя рабочей директории **./** , как это можно было делать при работе с командами операционной системы (например, при запуске самого **gcc**). Из соображений безопасности, если путь к исполняемому файлу не задан явно, операционная система ищет его по имени в специальных директориях, где хранятся различные программы. Директории, в которых работают пользователи, в этот список, как правило, не входят. Если компилятор по каким-либо причинам не смог сделать исполняемый файл, он выдаст сообщения об этом, и исполняемый файл создан не будет.

У **gcc** большое количество опций, позволяющее управлять всеми аспектами компиляции. Ниже приведены некоторые из них, наиболее часто встречающиеся приведены в табл. 8.

Таблица 8

Опция	Описание и комментарии
-o	Задание имени выходного файла – имя должно стоять сразу после опции, отделенное пробелом. Пример: **gcc –o hello hello.c** - результат успешной компиляции файла hello.c будет записан в исполняемый файл hello.
-l	(“Строчное эль”) Использование указанной библиотеки в процессе компиляции. Пример**:** **gcc hello.c –o hello -lname** - использование библиотеки **libname** – обратите внимание: все имена файлов библиотек в gcc должны начинаться с **lib**, но этот префикс в опции указывать не надо. По умолчанию для программ на с включается только главная библиотека **libc**, содержащая самые часто используемые функции. Остальные библиотеки надо подключать опциями –l.
-Wall	Выводит предупреждения, вызванные потенциальными ошибками в коде, не препятствующими компиляции программы, но способными привести, по мнению компилятора, к тем или иным проблемам при её исполнении. Важная и полезная опция, разработчики **gcc** рекомендуют пользоваться ей всегда.

Некоторые системные вызовы для работы с файлами и файловой системой

Системными вызовами называются функции, предназначенные для взаимодействия программы с операционной системой. Познакомимся с некоторыми из них, касающимися работы с файлами и файловой системой.

Чаще всего, при работе с файлами, пользуются функциями стандартной библиотеки языка С, такими как fopen(), fclose(), fprintf(), fscanf(), fgets(), fputs(), fread(), fwrite() и т.д. Предполагается, что навык работы с ними уже есть, в частности, есть понимание того, что большинство функций предполагают некоторое знание о структуре информации, содержащейся в файле (например, о формате данных). Однако, эти функции представляют собой надстройку над системными вызовами операционной системы. Системные вызовы – это функции более низкого уровня, работающие с файлами, как с **потоками ввода-вывода**, то есть, как с последовательностью байт без какой-либо внутренней структуры.

Системный вызов open()

Прототип:

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

Описание: предназначен для выполнения операции открытия файла, то есть, подготовки файла для работы с ним программы. В случае успешного исполнения возвращает файловый дескриптор открытого файла (небольшое неотрицательное число, которое используется в дальнейшем для указания на этот файл в таблице открытых файлов программы).

path – указатель на строку, содержащую полное или относительное имя файла

flags – может принимать одно из трёх значений (целых именованных констант):

O_RDONLY – если над файлом будут осуществляться только операции чтения;

O_WRONLY – если над файлом будут осуществляться только операции записи;

O_RDWR – если над файлом будут осуществляться операции и чтения, и записи;

Каждое из этих значений может быть скомбинировано операцией “побитовое или” (|) со следующими именованными константами:

O_CREAT – если файла с указанным именем не существует, он будет создан;

O_EXCL – применяется совместно с O_CREAT – при совместном использовании в случае существования файла с указанным именем, открытия файла не производится, и констатируется ошибочная ситуация;

O_APPEND – при открытии файла и перед выполнением каждой операции записи (если она разрешена) указатель текущей позиции файла устанавливается на конец файла;

O_TRUNC – если файл существует, его размер уменьшается до нуля, все остальные атрибуты не изменяются, кроме времени последнего доступа и времени последней модификации. Есть и другие именованные константы, которые здесь можно использовать, но они за пределами нашего рассмотрения.

mode – задаёт атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Обязателен, если при формировании flags присутствует O_CREAT, в противном случае может быть опущен. Задаётся как сумма следующих восьмеричных значений:

0400 – разрешено чтение для владельца,

0200 – разрешена запись для владельца,

0100 – разрешено исполнение для владельца,

0040 – разрешено чтение для группы владельцев,

0020 – разрешена запись для группы владельцев,

0010 – разрешено исполнение для группы владельцев,

0004 – разрешено чтение для всех остальных пользователей,

0002 – разрешена запись для всех остальных пользователей,

0001 – разрешено исполнение для всех остальных пользователей.

Реально устанавливаемые права доступа к созданному файлу получаются из **mode** и маски прав доступа по умолчанию (**umask**) с помощью следующих побитовых операций:

(mode & ~umask) – то есть, каждый вид доступа будет разрешён, только если он не запрещён маской по умолчанию и разрешён в **mode**. В противном случае, этот вид доступа будет запрещён.

Возвращаемое значение: значение файлового дескриптора открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Системные вызовы read() и write()

Прототипы:

```
#include <sys/types.h>
#include <unistd.h>
size_t  read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Описание:

Предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации.

fd – дескриптор файла (канала потокового ввода-вывода), для которого проводятся операции;

addr – адрес начала области оперативной памяти, из которой будет браться или в которую будет помещаться информация;

nbytes – для **write()** – количество байт, которые требуется прочитать из оперативной памяти и записать в файл; для **read()** – количество байт, которые требуется прочитать из файла и записать в оперативную память.

Возвращаемое значение: при успешном завершении – количество байт, которые действительно были записаны или прочитаны. Может не совпадать с **nbytes**, например, если при чтении из файла он закончился раньше, чем это было задано; при записи в файл – если на диске исчерпалось свободное место. При попытке чтения из уже закончившегося файла возвращается **0**. При возникновении какой-либо ошибки возвращается отрицательное значение.

Системный вызов close()

Прототип:

```
#include <unistd.h>
int close(int fd);
```

Описание:

Корректно завершает работу с указанным потоком ввода-вывода, заданным файловым дескриптором.

fd – файловый дескриптор закрываемого потока;

Возвращаемое значение: **0** при успешном завершении, **-1** при возникновении ошибки.

Системный вызов lseek()

Прототип:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Описание:

Системный вызов lseek предназначен для изменения положения указателя текущей позиции в открытом регулярном файле. Параметр fd является дескриптором соответствующего

файла, т. е. значением, которое вернул системный вызов open). Параметр offset совместно с параметром whence определяют новое положение указателя текущей позиции следующим образом:

- Если значение параметра whence равно SEEK_SET, то новое значение указателя будет составлять offset байт от начала файла. Естественно, что *значение offset в этом случае должно быть неотрицательным*.

- Если значение параметра whence равно SEEK_CUR, то новое значение указателя будет составлять старое значение указателя + offset байт. При этом *новое значение указателя не должно стать отрицательным*.

- Если значение параметра whence равно SEEK_END, то новое значение указателя будет составлять длина файла + offset байт. При этом *новое значение указателя не должно стать отрицательным*.

Системный вызов lseek позволяет выставить текущее значение указателя за конец файла (т. е. сделать его превышающим размер файла). При любой последующей операции записи в этом положении указателя, файл будет выглядеть так, как будто возникший промежуток был заполнен нулевыми битами. Тип данных off_t обычно является синонимом типа long.

Возвращаемое значение: системный вызов возвращает новое положение указателя текущей позиции в байтах *от начала файла* при нормальном завершении и значение –1 при возникновении ошибки.

Системный вызов stat()

Прототип:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *fname, struct stat *data);
```

Возвращает информацию об указанном файле. Не требуется иметь права доступа к файлу, чтобы получить эту информацию, но требуются права поиска во всех директориях, которые находятся по пути к файлу. Информация помещается в структуру типа stat.

Возвращаемое значение: **0** при успешном завершении, **-1** при возникновении ошибки.

Функция opendir()

Прототип функции:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (char *name);
```

Описание функции:

Функция opendir служит для открытия потока информации для директории, имя которой расположено по указателю name. Тип данных DIR представляет собой некоторую структуру данных, описывающую такой поток. Функция opendir подготавливает почву для других функций, выполняющих операции над директорией, и позиционирует поток на первой записи директории.

Возвращаемое значение: при удачном завершении функция возвращает указатель на открытый поток директории, который будет в дальнейшем передаваться в качестве параметра всем другим функциям, работающим с этой директорией. При неудачном завершении возвращается значение NULL.

Функция readdir()

Прототип функции:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Описание функции:

Функция readdir служит для чтения очередной записи из потока информации для директории, параметр dir представляет собой указатель на структуру, описывающую поток директории, который вернула функция opendir( ). Тип данных struct dirent представляет собой структуру данных, описывающую одну запись в директории. Поля этой записи сильно варьируются от одной файловой системы к другой, но одно из полей, которое собственно и будет нас интересовать, всегда присутствует в ней. Это поле char d_name[ ] неопределенной длины, не превышающей значения NAME_MAX+1 (максимальный размер имени файла + 1), которое содержит символьное имя файла, завершающееся символом конца строки. Данные, возвращаемые функцией readdir, переписываются при очередном вызове этой функции для того же самого потока-директории.

Возвращаемое значение: при удачном завершении функция возвращает указатель на структуру, содержащую очередную запись директории. При неудачном завершении или при достижении конца директории возвращается значение NULL.

Функция rewinddir()

Прототип функции:

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir (DIR *dir);
```

Описание функции: Функция rewinddir служит для позиционирования указателя текущей позиции потока-директории, ассоциированного с указателем dir (т. е. с тем, который вернула функция opendir, на первую запись (т.е. на начало) директории.

Функция closedir()

Прототип функции:

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

Описание функции: Функция closedir служит для закрытия потока информации для директории, ассоциированного с указателем dir (т. е. с тем, который вернула функция **o**pendir). После закрытия поток директории становится недоступным для дальнейшего использования.

Возвращаемое значение: при успешном завершении функция возвращает значение 0, при неудачном завершении – значение -1.

Пример программы, использующей средства из fcntl.h и unistd.h для работы с файлами

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
```

```
int main()
{
int in, out;
int nread;
char block[1024];

in   = open("file.in", O_RDONLY);
out = open("file.out", O_WRONLY|O_CREAT);
while((nread = read(in, block, sizeof(block))) > 0)
      write(out, block, nread);

exit(0);
}
```

Пример программы работы с директорией

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

void printdir(char *dir)
{
DIR *dp;
struct dirent *entry;

if ((dp = opendir(dir)) == NULL) {
      fprintf(stderr, "cannot open directory: %s\n", dir);
      return;
      }
while((entry = readdir(dp)) != NULL)
      if(strcmp(".",entry->d_name)==0 || strcmp("..",entry-
>d_name)==0);
      else printf("%s\n", entry->d_name);

closedir(dp);
}

int main()
{
/* Обзор каталога /home */
printf("Directory scan of /home:\n");
printdir("/home");
printf("done.\n");

exit(0);
}
```

Замечание: обратите внимание на ветвь if в цикле while в функции printdir – смысл этого оператора в игнорировании имён “.” и “..” – в любой директории эти имена обязательно содержатся (кроме корневой, в корневой есть только “.”) – каждая директория обязательно хранит ссылку на саму себя – это “.” и на родительскую директорию – это “..”. Но в данном случае они не нужны.

Задание к лабораторной работе

1. Изучите исходные тексты приведённых выше программ, определите, что они делают.

2. В домашней директории создайте директорию, которая будет называться Вашим именем, в ней создайте директорию lab1, а в ней две директории – input и output. В директории input создайте не менее двух-трёх текстовых файлов, в каждом из которых записан текст латинскими буквами, не более 1 кбайта. Других файлов там быть не должно.

3. В директории lab1 создайте файл с исходным кодом, и с помощью gcc создайте исполняемый файл программы. Программа должна прочитать информацию из каждого файла в директории input, произвести с ними действия согласно полученному варианту, результаты записать в файл, находящийся в директории output, вывести на экран список имён прочитанных файлов, и имя файла с результатами.

Обратите внимание: имя файла, содержащееся в строке d_name структуры dirent, с которой работает функция readdir – это локальное имя файла внутри директории. Так как программа должна запускаться не из директории, где эти файлы находятся, то системные вызовы работы с файлами (open, например) должны получать в качестве параметров либо полное, либо относительное имя файла, включающее имя директории.

Для работы с директориями и регулярными файлами разрешается пользоваться только системными вызовами и функциями Linux, но при формировании относительных имен файлов и при обработке прочитанной из файлов информации разрешается (и рекомендуется) пользоваться средствами из стандартной библиотеки языка C (в частности, из ctype.h, stdio.h и stdlib.h).

В программе не должна использоваться априорная информация о количестве файлов в директории input, их названиях и количестве текста в них, кроме условия, что количество текста в одном файле не может превышать 1 килобайта.

Обратите внимание: для правильной работы программы у файлов в директориях input и output должны быть правильно установлены права доступа. Для простоты можно предоставить все права всем.

Варианты задания на обработку данных:

1. Посчитать общее количество символов, являющихся буквами во всех входных файлах. Полученный результат вывести в файл.

2. Посчитать количество букв в нижнем регистре (строчных) во всех входных файлах, вывести полученные результаты в файл.

3. Посчитать количество букв в верхнем регистре (заглавных), вывести полученные результаты в файл.

4. Посчитать количество пробельных символов (пробелов и табуляций) во всех входных файлах. Результат вывести в файл.

5. Посчитать количество символов, являющихся цифрами во всех входных файлах. Результат вывести в файл.

6. Посчитать количество знаков препинания во всех входных файлах. Результат вывести в файл.

Во время отчёта по лабораторной работе будет проверяться знание основных команд для работы с файловой системой командной оболочки, знание системных вызовов и функций для работы с регулярными файлами и директориями, правильность работы написанной программы и понимание её исходного кода.

ЛАБОРАТОРНАЯ РАБОТА №2

1. Начальные сведения о процессах в Linux.
2. Работа с процессами в командной оболочке.
3. Системные вызовы для управления процессами.

Начальные сведения о процессах в Linux

Процесс – это программный код в стадии исполнения. Чаще всего это программный код, являющийся какой-то программой (и для простоты можно себе так и представлять), но бывают случаи, когда для выполнения одной программы необходимы несколько процессов, а бывает и так, что один процесс выполняет последовательно несколько программ.

Управление процессами (в частности, выделение для них необходимой оперативной памяти, времени процессора, других ресурсов, а также загрузка программного кода в оперативную память и запуск) – задача ядра операционной системы. В ОС Linux любой процесс может быть порождён только другим процессом, с единственным исключением – самым первым процессом, называемым **kernel,** который запускается при старте операционной системы. Процессы, порождённый каким-то процессом, называются **процессами-потомками** этого процесса, а процесс, породивший другой процесс, по отношению к процессу потомку называется **родительским процессом**. Процесс kernel порождает процесс **init**, который производит инициализацию различных структур операционной системы, а также порождает другие процессы, те, в свою очередь, могут порождать другие и т.д. В частности, потомками процесса init являются процессы, реализующие т.н. **системные сервисы**, то есть, некоторые действия операционной системы, происходящие без интерактивной обработки данных пользователей, и потому не подключённые ни к одному терминалу. Такие процессы принято называть **процессами-демонами**. Примерами могут служить сервера протоколов HTTP и FTP httpd и ftpd, входящие в состав операционной системы, или сервер системного журнала syslogd (по традиции имена файлов программ-демонов оканчиваются на d, хотя это не обязательно).

Для хранения информации о существующих в данный момент процессах используется т.н. **таблица процессов**. Каждому процессу соответствует запись в этой таблице, она идентифицируется уникальным номером, называемым **идентификатором процесса** или **PID** (process identificator). Процесс kernel всегда имеет PID = 0, init – PID = 1**,** другие процессы нумеруются в порядке появления. При переполнении поля номера, то есть, превышении значением PID числа 2^{31}-1, используется самый младший из свободных номеров (так как некоторые компьютеры, например, сетевые серверы под управлением Linux, могут без перезагрузки интенсивно работать годами, такая ситуация иногда действительно происходит).

За время своего существования, процесс может находиться в разных **состояниях**. Упрощённая диаграмма состояний для пользовательского процесса ОС Linux изображена на рис. 3.

Процесс–родитель порождает процесс-потомок при помощи специального системного вызова, во время которого порождённый процесс находится в состоянии “Рождение”. В этом состоянии операционная система выделяет процессу необходимые для рождения ресурсы и создает у себя структуры данных, необходимые для корректного управления всем жизненным

циклом процесса. Переводится процесс из состояния в состояние также операционной системой.

Рис. 3. Диаграмма состояний пользовательского процесса (https://www.intuit.ru/studies/courses/2249/52/lecture/1552)

Во время рождения процесса операционная система создаёт для управления им некоторую совокупность данных, которую, для простоты, будем считать единой структурой и называть её **блоком управления процес**сом или **PCB** – Process Control Block (на самом деле, это несколько взаимосвязанных структур, одна из которых – это запись в таблице процессов, но детали состава и размещения этой информации для нас здесь не важны).

PCB включает в себя **системный контекст процесса** и **регистровый контекст процесса.** В состав системного контекста процесса входят:

- учётные данные (**PID**, идентификатор родительского процесса **PPID** – parent process identificator, идентификатор пользователя, инициировавшего создание процесса **UID** – user identificator, общее время использования процессора данным процессом и т.д.);

- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение выделенного процессу адресного пространства и т.д.);

- текущее состояние процесса;

- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых процессом файлов и т.д.);

Регистровый контекст процесса – это содержимое регистров процессора, включая программный счётчик (то есть, адрес команды в оперативной памяти, которую процессор будет выполнять следующей) в момент, когда процесс последний раз перешёл из состояния “исполнение в режиме пользователя” в состояние “исполнение в режиме ядра”.

Информации PCB достаточно, чтобы корректно переводить процесс из одного состояния в другое, но полностью его не характеризует. У процесса есть ещё **пользовательский контекст** – то есть, содержимое адресного пространства процесса. Его начальное состояние тоже создаётся при рождении процесса путём загрузки программного кода в выделенную для

процесса область оперативной памяти. Эти три контекста полностью характеризуют процесс в любой момент времени (см. рис. 4).

Рис. 4. Контексты процесса (https://www.intuit.ru/studies/courses/2249/52/lecture/1552)

Из состояния готовности в состояние “исполнения в режиме ядра” и далее в состояние “исполнения в режиме пользователя” процесс переводится механизмом операционной системы, называемым **планировщиком**, когда до данного процесса дойдёт очередь на выделение процессорного времени. Из состояния “исполнение в режиме пользователя” в состояние “исполнения в режиме ядра” процесс переходит либо когда выполняет системный вызов, либо по **прерыванию**, когда происходит какое-то событие, при котором необходимо либо временно, либо вообще прекратить работу (например, началась операция ввода-вывода или в программе произошло деление на ноль). В состоянии ожидания процесс ждёт наступления какого-либо события, имеющего вид сообщения или **сигнала** (будут рассмотрены позже) от другого процесса (например, сообщение от процесса, управляющего устройством ввода-вывода о том, что запрашиваемый ввод или вывод закончен).

Попав в состояние “закончил исполнение”, процесс не исчезает из таблицы процессов, но освобождает все выделенные ему ресурсы. Такие процессы называются **процессами-зомби**. Они ожидают, пока либо родительский процесс не считает их код завершения с помощью специального системного вызова, либо сам не перейдёт в состояние “закончил исполнение”, и только после этого исчезают полностью, освобождая PID и запись в таблице процессов.

Возможны ситуации, когда родительский процесс завершается раньше процесса-потомка. В этом случае операционная система назначает новым родительским процессом для **процессов-сирот** процесс init (эта операция называется **переподчинением** (**reparenting**)).

В Linux существует несколько способов взаимодействия между процессами. Один из основных – сигналы. **Сигнал** – это асинхронное уведомление процесса о каком-либо событии. Послать его процессу может либо операционная система, либо администратор, либо процесс

пользователя с тем же UID (то есть, с тем же владельцем). Когда сигнал послан процессу, операционная система прерывает выполнение процесса (переводит его из состояния “исполнение в режиме пользователя” в состояние “исполнения в режиме ядра”). При этом, если процесс установил собственный **обработчик сигнала** (специальную функцию в программе, особым образом написанную), операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Названия сигналов «SIG…» являются числовыми константами со значениями, определяемыми в заголовочном файле signal.h. Числовые значения сигналов могут меняться от системы к системе (семейства Linux), хотя основная их часть имеет в разных системах одни и те же значения. В Linux используется более трёх десятков сигналов, мы познакомимся с несколькими наиболее важными.

Работа с процессами в командной оболочке

ps – вывод списка текущих процессов (опции приведены в табл. 9).

Таблица 9

Опция	Описание и комментарии
-a	Вывод информации о процессах всех пользователей (при отсутствии - только о процессах данного пользователя)
-U usermane	Вывод информации о процессах пользователя **username**.
-p PID1,[PID2,…]	Вывод информации только о процессах с заданными PID (идентификаторы в списке разделяются запятыми, без пробелов)
-x	Вывод информации о процессах, не подключённых к терминалам (например, о процессах-демонах)
-m	Сортировка вывода по объёму используемой процессом оперативной памяти
-r	Сортировка вывода по используемому процессом процессорному времени
-u	Вывод информации о ресурсах, используемых процессами
Пример	
ps -u -x -r	Вывод списка процессов текущего пользователя, в том числе не подключённых к терминалу, с информацией об используемых ими ресурсах, отсортированный по величине используемого процессорного времени

Формат вывода информации о процессах

Выводимый список включает несколько столбцов; эти столбцы содержат следующую информацию:

USER – имя пользователя, запустившего процесс;

PID – идентификатор процесса;

%CPU – процент процессорного времени, потребляемый процессом;

%MEM – процент объема памяти, занятой процессом;

COMMAND – команда, использованная для запуска процесса;

TT – терминальное устройство, к которому подключен процесс;

VSZ – объем виртуальной памяти, затребованный процессом, в блоках по 1024 байта;

RSZ – реальный объем памяти, используемый процессом, в блоках по 1024 байта;

START – время запуска процесса.

STAT – информация о состоянии процесса; первая буква означает состояние процесса:

S – процесс находится в состоянии ожидания менее 20 секунд;

I – процесс находится в состоянии ожидания 20 секунд или более;

К – процесс находится в состоянии готовности;

Т – процесс остановлен;

Z – процесс-зомби.

За первой буквой может также быть указана дополнительная информация (здесь не рассматривается).

top – программа наблюдения за процессами в системе

Выводит и периодически обновляет таблицу с информацией о текущих процессах (каждому процессу отводится 1 строка). После запуска и вывода информации не возвращает управление пользователю, выход их программы – нажатием клавиши **Q**, опции приведены в табл. 10.

Таблица 10

Опция	Описание и комментарии
-o par **-o+par**	Задаёт параметр **par** для сортировки строк таблицы и направление сортировки. По умолчанию – сортировка по убыванию, знак + перед именем параметра (без пробелов перед ним и после него) – сортировка по возрастанию; Возможные значения par: **pid** – идентификатор процесса (также используется по умолчанию) **cpu** – доля используемого процессорного времени **time** – время выполнения процесса **rsize** – реальный используемый процессом объём памяти **vsize** – объём виртуальной памяти, используемой процессом **command** – команда, которой был запущен процесс
-s num	Задаёт интервал времени в секундах между обновлением информации в таблице (по умолчанию 1 секунда)
-n num	Задаёт число процессов, о которых выводится информация
Пример	
top –n 10 -s 3 –o cpu	Выводится таблица не более, чем из 10 строк, строки отсортированы по убыванию доли используемого процессорного времени, обновление каждые 3 секунды

Формат таблицы программы top

Столбцы таблицы содержат следующую информацию:

PID – идентификатор процесса;

TIME – процессорное время, потребленное процессом с момента запуска;

%CPU – процент процессорного времени, потребляемого процессом;

USERNAME – пользователь, от имени которого выполняется процесс.

kill – посылка процессу сигнала из командной оболочки

Опции команды см. в табл. 11.

Таблица 11

Команда, имя сигнала	Описание и комментарии
kill –s sig PID1 [PID2…]	Посылает сигнал **sig** (может быть задан именем или значением) процессам с заданными идентификаторами. Опция –s может отсутствовать, тогда по умолчанию посылается сигнал **SIGTERM**
Примеры сигналов (значения sig)	
SIGKILL	Безусловное завершение процесса. Не может быть проигнорирован или обработан обработчиком сигналов, установленным процессом
SIGSTOP	Безусловная остановка процесса (перевод в состояние ожидания). Не

	может быть проигнорирован или обработан обработчиком сигналов, установленным процессом
SIGCONT	Возобновление работы после SIGSTOP
SIGTERM	Завершение процесса. В отличие от SIGKILL этот сигнал может быть обработан или проигнорирован программой. Как правило, процесс-родитель посылает процессу-потомку этот сигнал, чтобы тот завершил работу
SIGCHLD	Сигнал, уведомляющий родительский процесс об изменении состояния процесса-потомка (при остановке, возобновлении или завершении)
SIGALARM	Сигнал, посылаемый процессу по истечении времени, предварительно заданного функцией alarm().Программы обычно используют SIGALRM при реализации тайм-аута для долговременной операции, или выполнения операции через определённые интервалы
kill -l	Вывод списка всех сигналов, поддерживаемых системой

Системные вызовы для управления процессами

Системные вызовы getpid() и getppid()

Прототип:

```
#include <sys/types.h>
#include <unistd.h>
pid_t  getpid(void);
pid_t  getppid(void);
```

Описание:

getpid() возвращает идентификатор текущего процесса, getppid() возвращает идентификатор процесса-родителя для текущего процесса. Тип pid_t является синонимом одного из целочисленных типов языка С.

Системный вызов fork()

Прототип:

```
#include <sys/types.h>
#include <unistd.h>
pid_t  fork(void);
```

Описание:

Создаёт новый процесс. Это единственный в ОС Linux способ создать новый процесс после загрузки системы. Процесс, который произвёл системный вызов fork() становится родительским процессом, а порождённый этим системным вызовом процесс – процессом-потомком. Процесс-потомок является почти точной копией родительского процесса, кроме:

- идентификатора процесса;
- идентификатора родительского процесса;
- времени, оставшегося до получения сигнала SIGALARM.
- сигналы, ожидавшие доставки родительскому процессу, процессу потомку доставляться не будут.

При успешном исполнении системного вызова fork(), возврат из него происходит дважды – в родительский процесс возвращается значение PID процесса-потомка, а в порождённый процесс-потомок – ноль. Это единственный системный вызов в Linux,

обладающий двойным возвратом. В случае неуспеха процесс-потомок не создаётся, и в процесс-родитель возвращается отрицательное число.

После выхода из системного вызова оба процесса продолжают работу с одного и того же места кода – сразу после этого системного вызова.

Завершение процесса. Функция exit()

Прототип:

```
#include <stdlib.h>
void exit(int status);
```

Описание:

Существует два корректных способа завершения процесса в языке С. Первый – по достижении конца функции main(), или при выполнении оператора return в функции main(). Второй – вызов функции exit() в каком-либо другом месте программы (на самом деле, при завершении функции main() функция exit() тоже неявно вызывается – это делает операционная система). При выполнении этой функции происходит корректное закрытие всех открытых потоков (файлов) ввода-вывода (с сохранением на диске всех частично заполненных буферов ввода-вывода), после чего производится системный вызов прекращения работы процесса и перевода его в состояние “закончил исполнение”. Возврата из функции в текущий процесс не происходит, и никакого значения не возвращается. Родительскому процессу операционная система посылает сигнал SIGCLD. Значение параметра status – кода завершения процесса – передаётся ядру операционной системы и может быть передано в процесс-родитель завершённого процесса (при этом используются только 8 младших бит целого числа, поэтому для кода завершения имеют смысл значения от 0 до 255). По соглашению, код безошибочного завершения процесса 0.

Если процесс родитель явно не указал, что не хочет получать информацию о статусе завершившихся процессов-потомков, то завершившийся процесс не исчезнет из системы, а будет находиться в состоянии “завершил исполнение” (станет процессом-зомби) либо до того момента, когда процесс-родитель получит его код завершения, либо до того момента, когда процесс-родитель закончит исполнение.

Процесс-родитель может приостановить своё выполнение до окончания процесса потомка и получить его код завершения с помощью функции wait.

Ожидание родительским процессом завершения процесса-потомка. Функция wait()

Прототип:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Описание: Приостанавливает выполнение текущего процесса до тех пор, пока процесс-потомок не прекратит выполнение вызовом функции exit(status) с некоторым статусом завершения (или до появления сигнала, который либо завершает сам родительский процесс, либо требует вызвать функцию-обработчик, но эта возможность здесь не рассматривается). Если процесс-потомок к моменту вызова функции wait в процессе-родителе уже завершился (т.е. потомок – "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с процессом-потомком, освобождаются и он исчезает из системы. Процесс-родитель после этого продолжает исполнение.

В переменную целого типа, адрес которой передаётся аргументом **status,** помещается значение кода завершения из функции exit процесса-потомка.

Возвращаемое значение: PID завершившегося процесса-потомка.

Таким образом, процесс-родитель получает информацию и о том, какой именно процесс-потомок завершился, и о том, каков код его завершения.

Параметры функции main() в языке С. Переменные среды и аргументы командной строки

Кратко повторим здесь стандартный механизм взаимодействия с операционной системой, предусмотренный в языке С для функции main, так как похожий механизм используется в Linux при запуске исполняемых программ.

У функции main() в языке программирования С существует три параметра, которые могут быть переданы ей операционной системой. Полный прототип функции main() выглядит следующим образом:

```
int main(int argc, char *argv[], char *envp[]);
```

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки. Вся командная строка рассматривается как набор слов, разделенных пробелами. Через параметр argc передается количество слов в командной строке, которой была запущена программа. Параметр argv является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

```
a.out 12 abcd
```

то значение параметра argc будет равно 3, argv[0] будет указывать на имя программы – первое слово – "a.out", argv[1] - на слово "12", argv[2] - на слово "abcd". Так как имя программы всегда присутствует на первом месте в командной строке, то argc всегда больше 0, а argv[0] всегда указывает на имя запущенной программы.

Анализируя в программе содержимое командной строки, мы можем предусмотреть ее различное поведение в зависимости от слов, следующих за именем программы. Таким образом, не внося изменений в текст программы, мы можем заставить ее работать по-разному от запуска к запуску. Именно так команды операционной системы распознают свои параметры и аргументы. Например, компилятор gcc, вызванный командой gcc 1.c будет генерировать исполняемый файл с именем a.out, а при вызове командой gcc 1.c –o 1.exe – файл с именем 1.exe.

Третий параметр – envp – является массивом указателей на параметры окружающей среды процесса. Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В дальнейшем они могут быть изменены с помощью специальных команд операционной системы Linux. Каждый параметр имеет вид: переменная=строка. Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки. Например, задание параметра TERM=vt100 может говорить процессам, осуществляющим вывод на экран дисплея, что работать им придется с терминалом vt100. Меняя значение переменной среды TERM, например на TERM=console, мы сообщаем таким процессам, что они должны изменить свое поведение и осуществлять вывод для системной консоли.

Размер массива аргументов командной строки в функции main() мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра

нет, то его размер определяется другим способом – последний элемент этого массива содержит указатель NULL.

Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Прототипы:

```
#include <unistd.h>

int execv(const char *path, char *argv[]);
int execvp(const char *file, char *argv[]);
int execve(const char *path, char *argv[], char *envp[]);
int execl(const char *path, const char *arg0, . . . /* const char *argN,
(char *)NULL */ );
int execlp(const char *file, const char *arg0, . . .  /* const char *argN,
(char *)NULL */ );
int execle(const char *path,  const char *arg0, . . . /* const char *argN,
(char *)NULL, char *enva */ );
```

Описание: Любая функция семейства загружает новый пользовательский контекст из заданного файла в контекст текущего процесса (то есть, запускает на исполнение заданный исполняемый файл). Функции отличаются друг от друга формой представления параметров:

- аргумент **file** является указателем на имя файла, который должен быть загружен (если в имени не указан полный или относительный путь, то сначала исполняемый файл будет искаться в системных директориях, задаваемых переменными среды, и только потом в текущей директории пользователя);

- аргумент **path** – это указатель на полный путь к файлу, который должен быть загружен;

- аргумент **argv** представляет собой массив из указателей на аргументы командной строки, начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель NULL (аналог argv в прототипе функции main, но без явного размера массива);

- аргумент **envp** представляет собой массив из указателей на переменные среды, заданные в виде строк "переменная=строка", а заканчиваться массив должен элементом, содержащим указатель NULL (также, как envp в прототипе функции main);

- аргументы arg0, ..., argN представляют собой указатели на соответствующие аргументы командной строки (все вместе они заменяют argv). Заметим, что аргумент arg0 должен указывать на имя загружаемого файла. Поскольку функции, использующие arg0, ..., argN - это функции с переменным числом аргументов, то после последнего указателя на аргумент командной строки (после argN) при вызове функции должен идти специальный аргумент (char *)NULL, по которому эти варианты функции exec распознают окончание последовательности аргументов arg0, ..., argN**;**

- аргумент enva – указатель на строку переменных среды, должен стоять сразу после

(char *)NULL, заканчивающего последовательность указателей на аргументы командной строки.

Поскольку вызов функции семейства exec не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;

- идентификатор родительского процесса;
- идентификатор группы процессов;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала SIGALRM;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- идентификатор группы пользователя;
- явное игнорирование сигналов (если задано);
- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак "закрыть файл при выполнении exec()").

В случае успешного выполнения, возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Пример программы с порождением процессов

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
pid_t child, pid, ppid;
int a = 0;

child= fork();
if (child < 0) {                        /* ошибка создания процесса */
    printf("error\n");  exit(1);
} else if (child == 0) {     /* код для процесса-потомка */
    a = a+l;
} else {               /* код для процесса-родителя */
    a = a+l;
}

pid   = getpid();
ppid = getppid();
printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid,
(int)ppid, a);

return 0;
}
```

Задание на лабораторную работу

1. Создайте в вашей директории директорию lab2, в ней откомпилируйте и запустите приведённую выше программу. Запомните результаты её работы (значение переменной **a** в

обоих процессах - это пригодится в следующей лабораторной работе). Остальное выполнение данной лабораторной работы также производите в директории lab2.

2. Скопируйте из директории lab1 в директорию lab2 директории input и output и исполняемый файл первой лабораторной работы. Из директории output удалите файл с результатом. Напишите программу, аналогичную программе из первой лабораторной работы, но производящую обработку информации из файлов директории input согласно полученному варианту (см. ниже).

Обратите внимание: название выходного файла в директории output у этой программы должно отличаться от имени выходного файла программы из лабораторной работы №1, чтобы в директории output сохранялись файлы с результатами обоих программ.

Убедитесь в правильности работы написанной программы.

3. Напишите программу, в которой родительский процесс создаёт процесс-потомок и ждёт окончания его работы. А процесс-потомок загружает на исполнение программу из лабораторной работы №1. Потом процесс-родитель создаёт другой процесс-потомок, который загружает на исполнение программу из пункта 2 данной лабораторной работы. **Обратите внимание:** перед вызовом функции семейства exec (рекомендуется использовать здесь функцию execvp), необходимо сформировать правильные структуры данных, передаваемые ей в качестве аргументов (в частности, структуру argv).

Убедитесь в правильности работы данной программы.

Варианты задания на обработку данных:

1. Найти среди входных файлов файл, содержащий максимальное количество символов являющихся буквами, записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.
2. Найти среди входных файлов файл, содержащий минимальное количество символов являющихся буквами, записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.
3. Найти среди входных файлов файл, содержащий максимальное количество символов являющихся цифрами, записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.
4. Найти среди входных файлов файл, содержащий минимальное количество символов являющихся цифрами, записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.
5. Найти среди входных файлов файл, содержащий максимальное количество символов являющихся буквами верхнего регистра (заглавными), записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.
6. Найти среди входных файлов файл, содержащий минимальное количество символов являющихся буквами верхнего регистра (заглавными), записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.
7. Найти среди входных файлов файл, содержащий максимальное количество символов являющихся буквами нижнего регистра (строчными), записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.

8. Найти среди входных файлов файл, содержащий минимальное количество символов являющихся буквами нижнего регистра (строчными), записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.

9. Найти среди входных файлов файл, содержащий максимальное количество пробельных символов (пробелов и табуляций), записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.

10. Найти среди входных файлов файл, содержащий минимальное количество пробельных символов (пробелов и табуляций), записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.

11. Найти среди входных файлов файл, содержащий максимальное количество знаков препинания, записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.

12. Найти среди входных файлов файл, содержащий минимальное количество знаков препинания, записать в выходной файл это количество и название этого файла. Если таких файлов несколько, записать все их имена.

Во время отчёта по лабораторной работе будет проверяться знание основных команд командной оболочки по мониторингу процессов, знание системных вызовов и функций для работы с процессами, правильность работы написанных программ и понимание их исходного кода.

ЛАБОРАТОРНАЯ РАБОТА №3

1. Взаимодействие процессов. Управление взаимодействием с помощью командной оболочки. Переключение стандартного ввода-вывода и конвейеры. Фильтры.
2. Программирование взаимодействия процессов. Системные вызовы и функции для организации pipe и FIFO в программах.
3. Нити исполнения, как реализация взаимодействия процессов через общую память.
4. Функции для организации нитей исполнения.

Взаимодействие процессов. Управление взаимодействием с помощью командной оболочки. Переключение стандартного ввода-вывода и конвейеры. Фильтры

Кроме сигналов, в Linux предусмотрено ещё несколько механизмов для взаимодействия процессов. Сигналы – самый первый и важный способ взаимодействия, но и самый ограниченный по объёму информации, которую один процесс может таким образом передать другому. При использовании сигналов процессы могут передавать друг другу только информацию о том, что какое-то событие произошло.

Существует ряд способов взаимодействия, которые позволяют передавать гораздо больший объём разнообразной информации.

В данной работе будут изучаться: один из способов, позволяющих процессам общаться посредством организации специальных потоков ввода-вывода – т.н. конвейер, и один из способов взаимодействия процессов посредством общей оперативной памяти – т.н. нити исполнения (threads).

В командной оболочке существуют возможности организовать взаимодействие между процессами, выполняющими команды оболочки. **Они основаны на механизме переключения ввода-вывода.** У каждой программы в операционной системе Linux есть три стандартных потока ввода-вывода:

- стандартный ввод – по умолчанию это ввод с клавиатуры;
- стандартный вывод – по умолчанию это вывод на экран;
- стандартный вывод сообщений об ошибках – по умолчанию это вывод на экран.

Но любой из них можно перенаправить, например, в файл, при запуске программы из командной оболочки, используя особые параметры командной строки:

< - для переключения стандартного ввода;

> - для переключения стандартного вывода, при этом, если файл существует, он будет стёрт и начат заново;

>> - для переключения стандартного вывода, при этом, если файл уже существует, выводимая информация будет дописываться в конец файла.

>& - для переключения и стандартного вывода и стандартного вывода ошибок (в один и тот же файл), при этом, если файл существует, он будет стёрт и начат заново;

>>& - для переключения и стандартного вывода и стандартного вывода ошибок (в один и тот же файл), при этом, если файл уже существует, выводимая информация будет дописываться в конец файла.

Можно переключать только некоторые стандартные каналы, а можно все. Общий вид команды с переключением стандартных каналов:

```
command [< infile] [> outfile]
command [< infile] [>& outfile]
```

где infile – имя файла, откуда команда будет получать ввод, как будто он набран на клавиатуре, outfile – имя файла, куда будет помещён вывод, который обычно идёт на экран (во втором случае туде же пойдёт и вывод ошибок). Пример:

```
ls –l > ls-output.txt
```

вместо вывода на экран списка файлов с их атрибутами, находящихся в текущей рабочей директории, этот список будет помещён в файл ls-output.txt; так как вывод ошибок в этом примере не переключен, то если какая-то ошибка при выполнении команды произойдёт, информация о ней будет выведена на экран.

Данный механизм переключения стандартных каналов используется для организации т.н. **конвейера**, когда стандартный вывод одной команды переключается на стандартный ввод следующей с использованием параметра │ (вертикальная черта):

```
command1 | command2
```

Пример 1:

```
ls /bin /usr/bin | less
```

команда less производит постраничный вывод на экран с возможностью просмотра предыдущих страниц-экранов. При обычном выводе результатов команды ls, если список файлов не умещается на экран, то будет виден только его конец (последняя страница), без возможности просмотреть ушедшую за верхнюю границу экрана часть. Так как в директориях /bin и /usr/bin размещаются файлы установленных на данном компьютере программ, то списки их (для каждой директории свой), следующие при выводе друг за другом, как правило, весьма велики, и указанное выше “переполнение экрана” происходит часто.

Команд в конвейере может быть произвольное число. Пример:

```
ls /bin /usr/bin | sort | less
```

команда sort производит сортировку строк, поступающих к ней на стандартный ввод и выводит их на стандартный вывод. По умолчанию строки сортируются в лексикографическом порядке, другой порядок сортировки может быть задан параметрами команды. В данном примере, список из каждой указанной в команде ls директории будет отсортирован “по алфавиту”, и все списки друг за другом будут постранично выведены на экран с возможностью просмотра всех выведенных страниц.

Программы обработки текста, которые информацию принимают со стандартного ввода, и результаты выдают на стандартный вывод в Linux принято называть **фильтрами**. В Linux их весьма много. Кроме **sort** и **less**, это, например, **unic**, **wc**, **grep**, **head**, **tail**, **tee** (предлагается познакомиться с ними самостоятельно).

Программирование взаимодействия процессов. Системные вызовы и функции для организации конвейеров типа pipe и FIFO в программах

В программах, наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами является т.н. pipe (канал, труба, конвейер), родственный конвейеру командной строки. Поскольку после того, как pipe создан, он для

программы будет представляется обычными потоками ввода-вывода, то и работа с ним осуществляется с помощью системных вызовов, реализующих потоковый ввод-вывод.

Важное отличие конвейера от файла заключается в том, что прочитанная получателем информация немедленно удаляется из него, и не может быть прочитана снова. Как правило, конвейер реализуется в виде кольцевого буфера в оперативной памяти (в адресном пространстве операционной системы), по которому при операциях записи и чтения перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может обогнать входной и наоборот.

Особенностью простого конвейера является то, что при его создании в файловой системе не происходит никаких изменений. Как будет показано ниже, это приводит к тому, что пользоваться таким конвейером могут только родственные процессы – либо процесс-родитель и процесс-потомок, либо процессы, имеющие одного процесса-родителя.

Неродственные процессы полностью изолированы друг от друга операционной системой, поэтому, всё, что делает один процесс, совершенно недоступно другому, и организация взаимодействия неродственных процессов несколько сложнее. Для того, чтобы конвейером могли пользоваться неродственные процессы, служит т.н. **именованный конвейер,** или, как его принято называть, **FIFO.** От неименованного конвейера он отличается лишь тем, что при его создании в файловой системе создаётся особый файл-метка, по которому неродственные процессы могут найти конвейер и начать его использовать.

Системные вызовы для работы с pipe и FIFO

Системный вызов pipe()

Прототип:

```
#include <unistd.h>
int  pipe(int *fd);
```

Описание:

Создаёт конвейер (pipe) в оперативной памяти. Параметр fd должен быть указателем на массив из двух целых переменных. При нормальном завершении вызова в fd[0] будет записан файловый дескриптор, соответствующий выходному потоку данных конвейера, и позволяющий выполнять только операцию чтения, а в fd[1] – файловый дескриптор, соответствующий входному потоку данных конвейера, и позволяющий выполнять только операции записи. Возвращаемое значение – 0 при нормальном завершении, -1 при ошибке.

При создании процесса-потомка, он, среди прочего, получает от процесса-родителя копию таблицы открытых процессом-родителем потоков ввода-вывода, в том числе в этой таблице будут и потоки, образующие конвейер. Процесс-потомок, таким образом, будет иметь к ним доступ по тем же самым файловым дескрипторам, и никаких других объектов для этого не требуется.

Функция mkfifo().

Прототип:

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

Описание:

Функция mkfifo предназначена для создания FIFO в операционной системе.

Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO, файла с таким именем перед вызовом функции не должно существовать.

Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

- 0400 – разрешено чтение для пользователя, создавшего FIFO;
- 0200 – разрешена запись для пользователя, создавшего FIFO;
- 0040 – разрешено чтение для группы пользователя, создавшего FIFO;
- 0020 – разрешена запись для группы пользователя, создавшего FIFO;
- 0004 – разрешено чтение для всех остальных пользователей;
- 0002 – разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно – они равны (0777 & mode) & ~umask.

Возвращаемые значения: при успешном создании FIFO функция возвращает значение 0, при неуспешном – отрицательное значение.

Следует отметить, что при этом не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам системного вызова open().

После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip’ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем (системным вызовом open()).

Работа системных вызовов read() и write() в случае конвейера

Системные вызовы read() и write() имеют определенные особенности поведения при работе с pipe, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования операционной системой (то есть, переводом процесса в состояния ожидания) обменивающихся информацией процессов Организация запрета блокирования этих вызовов для pipe выходит за рамки нашего курса.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. Помните, что за один раз из pipe может прочитаться меньше информации, чем вы запрашивали, и за один раз в pipe может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!

Одна из особенностей поведения блокирующегося системного вызова read() связана с попыткой чтения из пустого pipe. Если есть процессы, у которых этот pipe открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости

закрытия файлового дескриптора, ассоциированного с входным концом pipe, в процессе, который будет использовать pipe для чтения (см. пример программы ниже).

Аналогичной особенностью поведения при отсутствии процессов, у которых pipe открыт для чтения, обладает и системный вызов write(), с чем связана необходимость закрытия файловою дескриптора, ассоциированного с выходным концом pipe, в процессе, который будет использовать pipe для записи (см. пример программы ниже).

Пример программы, использующей pipe

```
/* Программа, осуществляющая однонаправленную связь через pipe между процессом–родителем и процессом–потомком */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {

int fd[2}, result;
size_t size;
char resstring[14];

/* Попытаемся создать pipe */
if(pipe(fd) < 0){       /* Если создать pipe не удалось */
    printf("Can\'t create pipe\n"); exit(-1);
}

/* Порождаем новый процесс */
result = fork();
if(result < 0) {        /* Если создать процесс не удалось */
      printf(”Can\'t fork child\n"); exit(-1);
} else if (result > 0) {
   /* мы находимся в родительском процессе, который будет передавать информацию процессу–ребенку. В этом процессе выходной поток данных нам не понадобится, поэтому закрываем его.*/
        close(fd[0]);

    /* Пробуем записать в pipe 14 байт, т.е. всю строку "Hello, world" вместе   с признаком конца строки */
        size = write(fd[l], "Hello, world!", 14);
        if(size != l4) {   /* Если записалось меньшее количество байт */
               printf("can\'t write all string\n"); exit(-l);
        }

    /* Закрываем входной поток, на этом родитель прекращает работу */
      close(fd[l]);
      printf("Parent exit\n");
} else {
```

```
    /* Мы находимся в порожденном процессе, который будет получать
информацию от процесса–родителя. Он унаследовал от родителя таблицу
открытых файлов и, зная файловые дескрипторы, соответствующие pip'у,
может его использовать. В этом процессе входной поток данных нам не
понадобится, поэтому закрываем его.*/
        close(fd[1]);

/* Пробуем прочитать из pip`а 14 байт в массив, т.е.всю записанную строку */
        size = read(fd[0], resstring, 14);
        if(size < 0) {  /* Если прочитать не смогли */
                printf("Can\'t read string\n"); exit(-1);
        }

        printf("%s\n",resstring);
        close(fd[0]);
}
return 0;
}
```

Нити исполнения, как реализация взаимодействия процессов через общую память

Нити исполнения – это “облегчённые” процессы. Нити исполнения процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются общими, они могут использовать их, как элементы разделяемой памяти, не прибегая к более сложным механизмам.

Основной причиной создания механизма нитей исполнения является выполнение большинством приложений существенного числа действий, некоторые из которых могут время от времени блокироваться. Схему программы можно существенно упростить, если разбить приложение на несколько последовательных нитей, запущенных в квазипараллельном режиме.

Еще одним аргументом в пользу нитей является легкость их создания и уничтожения (поскольку с нитью не связаны никакие ресурсы). В большинстве операционных систем (в том числе в Linux) на создание нити уходит примерно в 100 раз меньше времени, чем на создание процесса. Это свойство особенно полезно, если необходимо динамическое и быстрое изменение числа нитей.

Третьим аргументом является производительность. Концепция нитей не дает увеличения производительности, если все они ограничены возможностями процессора. Но когда имеется одновременная потребность в выполнении большого объема вычислений и операций ввода-вывода, наличие нитей позволяет совмещать эти виды деятельности во времени, тем самым увеличивая общую скорость работы приложения.

И наконец, концепция нитей полезна в системах с несколькими процессорами, где возможен настоящий параллелизм.

Системные вызовы для организации нитей исполнения

Каждая нить исполнения (**thread**), как и процесс, имеет в системе уникальный номер – идентификатор thread'a. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать

идентификатор этой нити и для любого обычного процесса. Для этого используется функция pthread_self(). Нить исполнения, создаваемую при рождении нового процесса, принято называть **начальной** или **главной** нитью исполнения этого процесса.

Функция pthread_self() для определения идентификатора нити исполнения

Прототип функции:

```
#include  <pthread.h>
pthread_t  pthread_self(void);
```

Описание функции:

Функция pthread_self возвращает идентификатор текущей нити исполнения. Тип данных pthread_t является синонимом для одного из целочисленных типов языка С.

Создание и завершение thread'a. Функции pthread_create(), pthread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом

```
void *thread_routing(void *arg);
```

Параметр arg может, до некоторой степени, рассматриваться как аналог параметров функции main(). Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-потомка. Для создания новой нити исполнения применяется функция pthread_create(), которой в качестве аргументов передаются указатель на функцию нити, и аргумент для неё.

Функция для создания нити исполнения

Прототип функции:

```
#include <pthread.h>
int pthread_create(pthread_t   *thread,
                   pthread_attr_t  *attr,
                   void* (*thread_routing)(void*),
                   void *arg );
```

Описание функции:

Функция pthread_create служит для создания новой нити исполнения внутри текущего процесса. Новый thread будет выполнять функцию thread_routing, передавая ей в качестве аргумента параметр arg. Если требуется передать более одного параметра, они собираются в структуру, и в arg передается адрес этой структуры. Значение, возвращаемое функцией thread_routing не должно указывать на динамический объект данного thread'a (который исчезнет после завершения нити).

Параметр attr служит для задания различных атрибутов создаваемого thread'a. Их описание выходит за рамки нашего курса, и мы всегда будем считать их заданными по умолчанию, подставляя в качестве аргумента значение NULL.

Возвращаемые значения: При удачном завершении функция pthread_create возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается **положительное значение** (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код

ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается. Остальные детали работы этой функции – за пределами нашего курса.

Созданный thread может завершить свою деятельность тремя способами:

- С помощью возврата из функции, ассоциированной с нитью исполнения (из функции, на которую указывал параметр thread_routing при вызове функции pthread_create). Объект, на который указывает адрес, возвращаемый функцией thread_routing, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread, и должен указывать на объект, не являющийся локальным для завершившегося thread'a (то есть, который не исчезнет вместе с завершившейся нитью), например, на статическую переменную;
- С помощью выполнения функции pthread_exit(). Функция никогда не возвращается в вызвавшую ее нить исполнения (так как эта нить исчезает). Объект, на который указывает параметр этой функции (описание ниже), как и в предыдущем случае, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося thread'a, например, на статическую переменную;
- Если в процессе выполняется возврат из функции main() или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции exit(), это приводит к завершению всех thread'ов процесса.

Функция для завершения нити исполнения

Прототип функции

```
#include <pthread.h>
void pthread_exit(void *status);
```

Описание функции:

Функция pthread_exit служит для завершения нити исполнения (thread) текущего процесса. Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр status, может быть впоследствии изучен в другой нити исполнения, например в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'a.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции pthread_join() (аналог функции wait для процессов). Нить исполнения, вызвавшая эту функцию, переходит в состояние **ожидание** до завершения заданного thread'a. Функция позволяет также получить указатель, который вернул завершившийся thread в операционную систему.

Функция pthread_join()

Прототип функции

```
#include <pthread.h>
int pthread_join (pthread_t  thread,  void  **status_addr);
```

Описание функции

Функция pthread_join блокирует работу вызвавшей ее нити исполнения до завершения thread'a с идентификатором thread. После разблокирования в указатель, расположенный по адресу status_addr, заносится адрес, который вернул завершившийся thread либо при выходе из

ассоциированной с ним функции, либо при выполнении функции pthread_exit(). Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение NULL.

Возвращаемые значения: функция возвращает значение 0 при успешном завершении. В случае ошибки возвращается **положительное значение** (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается.

Пример программы с использованием двух нитей исполнения

```
/* Каждая нить исполнения просто увеличивает на 1 разделяемую
переменную a. */
#include <pthread.h>
#include <stdio.h>

int a = 0;  /* глобальная статическая переменная, поэтому она будет
          разделяться всеми нитями исполнения.*/

/* функция, ассоциированная со 2-ой нитью */
void *mythread(void *dummy)
{
pthread_t mythid;

/* Запрашиваем идентификатор нити */
mythid = pthread_self();

a = a+1;
printf("Thread %d, Calculation result = %d\n", mythid, a);
return NULL;
}

/* Функция main() – она же ассоциированная функция главной нити */
int main(){

pthread_t thid, mythid;
int result;

/* создаём вторую нить, её идентификатор помещаем в thid */
result = pthread_create( &thid,(pthread_attr_t *)NULL, mythread,
NULL);

if(result != 0){       /* если не получилось */
    printf ("Error on thread create, return value = %d\n", result);
    exit(-1);
}

printf("Thread created, thid = %d\n", thid);

/* Запрашиваем идентификатор главного thread'a */
```

```
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n", mythid, a);

    /* Ожидаем завершения порожденного thread'a. Если не выполнить вызов
этой функции, то возможна ситуация, когда мы завершим функцию main() до
того, как выполнится второй thread, что автоматически повлечет за собой
его завершение, исказив результаты его работы. */
    pthread_join(thid, (void **)NULL);
    return 0;
    }
```

Для сборки исполняемого файла при работе редактора связей в процессе компиляции этой программы необходимо явно подключить библиотеку функций для работы с pthread'ами, которая не подключается автоматически. Это делается с помощью добавления к команде компиляции параметра – lpthread – подключение библиотеки libpthread .

Задание на лабораторную работу

1. Выполнение лабораторной работы производить в директории Lab3 (подготовка директории аналогична предыдущей лабораторной работе).

2. Напишите программу, в которой два процесса общаются через pipe, выполняя совместно обработку данных из лабораторной работы №1. Родительский процесс сначала читает информацию из входных файлов, производит необходимые вычисления, потом создаёт процесс потомок, передаёт ему через pipe результаты и ждёт окончания процесса-потомка. А процесс-потомок записывает результат в выходной файл.

Обратите внимание: для того, чтобы избежать многих проблем при использовании pipe, и процесс-передатчик, и процесс-приёмник информации должны знать точное количество символов, пересылаемых через pipe. Так как в данном случае должно быть передано одно целое число, то количество символов в его символьном представлении можно задать с помощью формата вывода с задаваемой минимальной шириной **%*d**.

3. Откомпилируйте и запустите приведённую выше программу для двух нитей. Объясните отличие результатов относительно значения переменной **a** по сравнению с результатами программы с двумя процессами из лабораторной работы №2.

4. Напишите программу, которая выполняет обработку данных согласно варианту из лабораторной работы №2, но делает это, используя две нити – одна читает информацию из входных файлов и обрабатывает её, а другая записывает результат в файл.

Во время отчёта по лабораторной работе будет проверяться знание переключения ввода-вывода, конвейеры и фильтры в командной оболочке, знание системных вызовов и функций для работы с процессами и нитями, правильность работы написанных программ и понимание их исходного кода.

ЛАБОРАТОРНАЯ РАБОТА №4

1. Начальные сведения о файловой системе Linux (часть 2). Устройства ввода-вывода и драйверы.
2. Физическое размещение файловой системы на жёстких дисках.
3. Жесткие и мягкие связи.
4. Команды оболочки для организации файловой системы и работы с разными типами файлов.
5. Системные вызовы для работы с разными типами файлов.

Начальные сведения о файловой системе Linux (часть 2). Устройства ввода-вывода и драйверы

В операционной системе Linux существуют файлы нескольких типов, а именно:

- обычные или регулярные файлы;
- директории или каталоги;
- файлы типа FIFO или именованные pip'ы;
- специальные файлы устройств;
- специальные файлы связи (символьные ссылки) (1ink, symlink);
- сокеты (sockets).

Три первых типа файлов рассматривались ранее, в данной работе познакомимся со специальными файлами устройств и специальными файлами связи. Файлы типа сокет используются при организации сетевого взаимодействия и в данных лабораторных работах рассматриваться не будут.

Для единообразия, в операционной системе Linux работа с различными устройствами ввода-вывода организована, как работа с файлами – со **специальными файлами устройств**. Для этого архитектура операционной системы (не только Linux, а вообще большинства современных ОС) представляет собой “уровневую” структуру (см. рис. 5), в которой “нижний уровень” составляют собственно говоря, сами устройства ввода-вывода вместе со своими контроллерами, то есть блоками управления. На более высоком уровне находятся **драйверы устройств ввода-вывода** – то есть программы, входящие в операционную систему и предназначенные для управления некоторым устройством ввода-вывода посредством взаимодействия с его контроллером. Над драйверами находится т.н. **базовая система ввода-вывода**, которая работает с драйверами, как с файлами.

Драйвер получает от базовой системы ввода вывода команды для выполнения файловых операций, преобразует их в команды, специфичные для данного устройства, которые передаёт контроллеру устойства. И получая от контроллера результаты выполнения этих команд, он преобразует их в результаты выполнения файловых операций. Таким образом, вся специфика работы с конкретным устройством сосредоточена в драйвере, а базовая система ввода-вывода является независимой от конкретных особенностей устройств, с которыми работает. Такой подход является самым логичным из-за большого разнообразия устройств ввода-вывода и постоянного появления их новых видов.

Вообще, устройства обычно принято разделять по преобладающему типу интерфейса на следующие виды:

- символьные (клавиатура, модем, терминал и т. п.);

- блочные (магнитные и оптические диски и ленты, и т. д.);
- сетевые (сетевые карты);
- все остальные (таймеры, графические дисплеи, видеокамеры и т. п.);

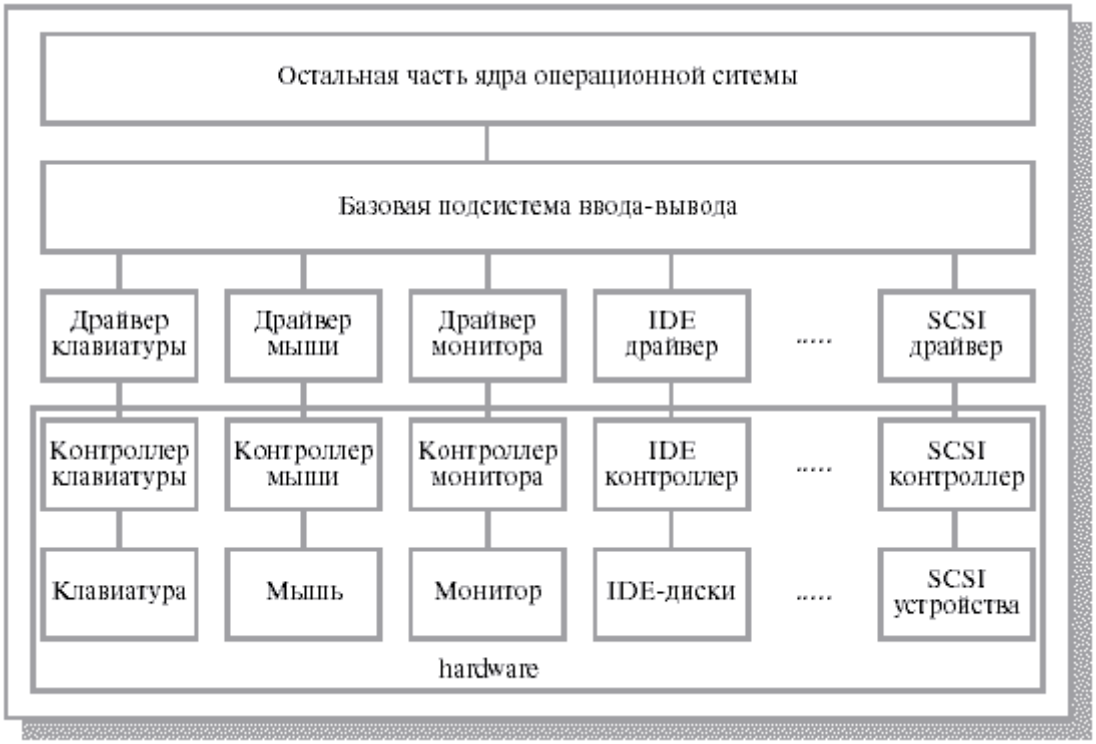

Рис. 5. Структура системы ввода-вывода (https://poznayka.org/s93533t1.html)

Такое деление является весьма условным. В одних операционных системах сетевые устройства могут не выделяться в отдельную группу, в некоторых других – отдельные группы составляют звуковые устройства и видеоустройства и т. д. Некоторые группы в свою очередь могут разбиваться на подгруппы: подгруппа жестких дисков, подгруппа мышек, подгруппа принтеров.

В Linux все устройства делятся на две большие категории: **символьные и блочные.** Символьные устройства – это устройства, которые умеют передавать данные только последовательно, байт за байтом, а блочные устройства – это устройства, которые могут передавать блок байтов как единое целое.

К символьным устройствам обычно относятся устройства ввода информации, которые спонтанно (с точки зрения компьютера) генерируют входные данные: клавиатура, мышь, модем, джойстик. К ним же относятся и устройства вывода информации, для которых характерно представление данных в виде линейного потока: принтеры, звуковые карты и т. д. По своей природе символьные устройства обычно умеют совершать две общие файловые операции: ввести символ (байт) и вывести символ (байт) – get и put.

Для блочных устройств, таких как магнитные и оптические диски, ленты и т. п. естественными являются операции чтения и записи блока информации – read и write, а также, для устройств прямого доступа (например, магнитных дисков), операция поиска требуемого блока информации – seek.

Драйверы символьных и блочных устройств должны предоставлять базовой подсистеме ввода-вывода функции для осуществления описанных общих операций. Помимо общих операций, некоторые устройства могут выполнять операции специфические, свойственные

только им – например, звуковые карты умеют увеличивать или уменьшать среднюю громкость звучания, дисплеи умеют изменять свою разрешающую способность. Для выполнения таких специфических действий в интерфейс между драйвером и базовой подсистемой ввода-вывода обычно входит еще одна функция, позволяющая непосредственно передавать драйверу устройства произвольную команду с произвольными параметрами, что позволяет задействовать любую возможность драйвера без изменения интерфейса. В операционной системе Linux такая функция получила название ioctl (от input-output control).

Помимо функций read, write, seek (для блочных устройств), get, put (для символьных устройств) и ioctl, в состав интерфейса обычно включают еще следующие функции:

- функцию инициализации или повторной инициализации работы драйвера и устройства – open.
- функцию временного завершения работы с устройством (может, например, вызывать отключение устройства) – close.

и некоторые другие (в данной работе рассматриваться не будут).

При исполнении команды ls –l, которая выводит на экран информацию об атрибутах файлов, первый символ в строке информации о файле обозначает его тип (см. табл. 12).

Таблица 12

Символ	Тип файла	Назначение
-	регулярный	хранение данных
d	директория	организация доступа к файлам
l	линк (символьная связь)	
b	блочное устройство	работа с устройствами ввода-вывода
c	символьное устройство	
p	FIFO (именованый pipe)	организация взаимодействия процессов
s	сокет	

Для файлов устройств ввода-вывода отведена директория /dev. При выводе на экран атрибутов файлов устройств ввода-вывода, вместо размера файла, выводятся два номера – т.н. **старший и младший номера драйвера**. Старший номер – это, собственно, номер драйвера, то есть, конкретного вида устройства (например, принтеры некоторой марки – конкретный вид символьных устройств), а младший – это номер самого устройства среди всех устройств такого вида, имеющихся в системе (одинаковых принтеров в системе может быть несколько).

Физическое размещение файловой системы на жёстких дисках

Устройства блочного типа делятся в Linux на два подтипа – допускающие организацию на них файловой системы или нет. К устройствам, допускающим организацию файловой системы, относятся, например, жёсткие диски (как магнитные, так и оптические).

Существует много типов файловых систем, с которыми может работать Linux. Они отличаются деталями, которые здесь рассматриваться не будут. Мы рассмотрим основные принципы физической организации файловых систем на дисках, которые у разных файловых систем довольно близки.

Физические диски в операционных системах принято логически делить на **разделы (partitions)** или **логические диски**. Причем слово «делить» не следует понимать буквально, в некоторых системах несколько физических дисков могут быть объединены в один раздел.

В операционной системе Linux физический носитель информации обычно представляет собой один или несколько разделов. В большинстве случаев разбиение на разделы производится линейно, хотя некоторые варианты Linux могут допускать некое подобие древовидного разбиения. Количество разделов и их размеры определяются при форматировании диска.

Наличие нескольких разделов на диске может определяться требованиями операционной системы или пожеланиями пользователя. Допустим, пользователь хочет разместить на одном жестком диске несколько операционных систем с возможностью попеременной работы в них, тогда он размещает каждую операционную систему в своем разделе. Или другая ситуация: необходимость работы с несколькими видами файловых систем. Под каждый тип файловой системы выделяется отдельный логический диск. Третий вариант – это разбиение диска на разделы для размещения в разных разделах различных категорий файлов. Скажем, в одном разделе помещаются все системные файлы, а в другом разделе – все пользовательские файлы.

Для простоты далее здесь будем полагать, что у нас имеется только один раздел и, следовательно, одна файловая система.

Как правило, всё дисковое пространство раздела логически разделяется на две части: заголовок раздела и логические блоки данных. Заголовок раздела обычно располагается в самом начале раздела, и состоит из **суперблока** и **массива индексных узлов**. **Логические блоки данных** хранят собственно содержательную информацию файлов и часть информации о размещении файлов на диске (т. е. какие логические блоки и в каком порядке содержат информацию, записанную в файл).

Суперблок хранит информацию, необходимую для правильного функционирования файловой системы в целом. В нем содержатся, в частности, следующие данные:

- тип файловой системы;
- флаги состояния файловой системы;
- размер логического блока в байтах (обычно кратен 512 байтам);
- размер файловой системы в логических блоках (включая сам суперблок и массив индексных узлов);
- размер массива индексных узлов (т.е. сколько файлов может быть размещено в файловой системе);
- число свободных индексных узлов (т.е. сколько файлов еще можно создать);
- число свободных блоков для размещения данных;
- часть списка свободных индексных узлов;
- часть списка свободных блоков для размещения данных.

В некоторых файловых системах два последних объекта (списки) вынесены за пределы суперблока, но остаются в заголовке раздела. При первом же обращении к файловой системе суперблок обычно целиком считывается в адресное пространство ядра для ускорения последующих обращений. Поскольку количество логических блоков и индексных узлов в файловой системе может быть весьма большим, нецелесообразно хранить списки свободных блоков и узлов в суперблоке полностью.

Индексный узел (**inode**) содержит все атрибуты файла (смотрите лабораторную работу №1, обратите внимание – имя файла в число этих атрибутов не входит) и информацию о логических блоках, в которых размещено содержимое файла. Заметим, что такие типы файлов,

как «symlink», «socket», «pipe» и файлы устройств не занимают на диске никакого иного места, кроме индексного узла (им не выделяется логических блоков). Все необходимое для работы с этими типами файлов содержится в их атрибутах.

Количество индексных узлов в разделе является постоянной величиной, определяемой на этапе генерации файловой системы и хранящейся в суперблоке. Все индексные узлы системы организованы в виде массива, хранящегося в заголовке раздела. Каждому файлу соответствует только один элемент этого массива и, наоборот, каждому непустому элементу этого массива соответствует только один файл. Таким образом, каждый файл на диске может быть однозначно идентифицирован номером своего индексного узла (его индексом в массиве).

Содержимое регулярных файлов (информация, находящаяся в них, и способ ее организации) всецело определяется программистом, создающим файл. В отличие от регулярных файлов, директории имеют жестко заданную структуру. Основным содержимым директории является таблица, каждая запись в которой содержит имя файла, принадлежащего директории и номер индексного узла, ему соответствующего. При этом, первая запись таблицы всегда содержит имя "**.**" и номер индексного узла, соответствующего самой этой директории ("ссылку на себя"), а вторая запись содержит имя "**..**" и номер индексного узла, соответствующего родительской директории ("ссылку на родительскую директорию").

При создании файла операционная система занимает под него свободный inode, и нужное количество логических блоков, изменяя соответствующим образом списки свободных индексных узлов и логических блоков, а при удалении файла – освобождает принадлежавшие файлу логические блоки и его inode, занося их обратно в списки свободных.

Жесткие и мягкие связи

Так как реальная информация о файле хранится в inode, а в директории лежит фактически только именованная ссылка на нужный inode, то нетрудно видеть, что в Linux регулярный файл может иметь несколько имён. В ряде ситуаций это очень удобно.

Например, если при работе какой-то программы ей нужно использовать некоторую библиотеку, которая периодически обновляется. В директории программы файл библиотеки называется так, как нужно этой программе, а в директории, где хранятся библиотеки, этот файл в названии содержит, например, номер версии, что удобно для администрирования. При замене файла библиотеки на новую версию, администратор с помощью специальных средств переставляет все ссылки со старой библиотеки на новую, после чего можно удалить старую или, на всякий случай, оставить, но все программы, которые ей пользуются, будут использовать новую версию.

Операция создания ещё одного имени для существующего файла называется созданием **жёсткой связи** (**link**). Созданная при этом запись ничем не отличается от той, которая возникает при создании файла. Количество таких жёстких связей у файла содержится в соответствующем атрибуте файла, хранящемся в inode (это число, которое выводится на экран командой ls –l сразу после прав доступа к файлу – см. лабораторную №1).

Файл хранится до тех пор, пока на него есть хотя бы одна жёсткая ссылка. Когда какой-то регулярный файл удаляется из директории, то на самом деле, удаляется только жёсткая ссылка на него, находящаяся в этой директории. Если на соответствующий inode есть ещё ссылки, то

файл с диска не будет удалён. Только если эта ссылка была последняя, inode и логические блоки будут переведены в списки свободных, и файл действительно будет удалён.

Для того, чтобы древовидная структура директорий не нарушалась, в Linux жёсткие связи запрещено создавать для директорий. По неизвестным причинам, также запрещено создавать жёсткие связи для файлов устройств ввода-вывода.

Кроме жёстких, в Linux существуют так называемые **мягкие** или **символические связи** (**symlink**). Они очень похожи на ярлыки в Windows. Это файлы специального типа, которые содержат путь (задаваемый при создании) к тому файлу, на который они ссылаются. При открытии файла типа symlink будет рекурсивно (так как одна символьная ссылка может указывать на другую) найден регулярный файл, на который ссылается цепочка symlink-ов (если она не слишком длинная) и открыт именно он. Если цепочка символьных связей превышает по длине некоторый предел, попытка открыть файл выдаст ошибку (через системную переменную errno) “Слишком много мягких связей”. Это сделано для избегания зацикливания.

При выводе атрибутов файла мягкой связи командой ls –l после имени файла будет выведена стрелка и имя, на которое связь ссылается.

Наличие мягких ссылок никак не влияет на возможность удаления файла. Если мягкая связь ведёт к несуществующему файлу, при попытке открыть её будет выдана ошибка “Файл отсутствует” (через системную переменную errno).

Команды оболочки для организации файловой системы и работы с разными типами файлов

Создание и удаление жёстких и мягких ссылок (см. табл. 13).

Таблица 13

Команда	Описание и комментарии
ln [-s] src [dest]	Параметры src и dest могут быть полными или относительными именами файлов, файл src должен существовать. Если имя файла dest не задано, то создаётся, при отсутствии –s, файл жёсткой связи, а при наличии –s, файл мягкой связи к файлу source с тем же именем в текущей директории. При наличии dest файл связи будет иметь это имя.
ln [-s] file1 […] dir	Для всех перечисленных в команде файлов создаются файлы связей заданного типа в директории dir. Они будут иметь те же имена, что и исходные файлы.
unlink file1 […]	Удаление файлов связей (любых типов) с указанными именами. Другое название команды **rm**, которую тоже можно здесь использовать.

Вывод номера индексного узла файла (см. табл. 14).

Таблица 14

Команда	Описание и комментарии
ls –i file1 […]	Выводит номер индексного узла и имя для каждого из указанных файлов.

Системные вызовы для работы с разными типами файлов

Системные вызовы для чтения атрибутов файла

Прототип системных вызовов

```
#include <sys/stat.h>
#include <unistd.h>

int  stat(char *filename, struct stat *buf);
int lstat(char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Описание системных вызовов: системные вызовы stat, fstat и lstat служат для получения информации об атрибутах файла.

Системный вызов stat читает информацию об атрибутах файла, на имя которою указывает параметр filename, и заполняет ими структуру, расположенную по адресу buf. Заметим, что имя файла должно быть полным, либо должно строиться относительно той директории, которая является текущей для процесса, совершившего вызов. Если файл относится к типу “символическая связь”, то (рекурсивно) читается информация об атрибутах файла, на который указывает символическая связь.

Системный вызов lstat идентичен системному вызову stat за одним исключением: если имя файла относится к файлу типа “символическая связь”, то читается информация о самом файле типа “символическая связь”.

Системный вызов fstat идентичен системному вызову stat, только файл задается не именем, а своим файловым дескриптором (то есть, файл к этому моменту должен быть открыт вызывающим или родственным ему процессом).

Для системных вызовов stat и lstat процессу не нужны никакие права доступа к указанному файлу, но могут понадобиться права для поиска во всех директориях, входящих в указанное имя файла.

Структура stat в различных версиях UNIX может быть описана по-разному. В Linux она содержит следующие поля:

```
struct stat {
   dev_t   st_dev;       /* устройство, на котором расположен файл  */
   ino_t   st_ino;       /* номер индексного узла для файла         */
   mode_t  st_mode;      /* тип файла и права доступа к нему        */
   nlink_t st_nlink;     /* счетчик числа жестких связей            */
   uid_t   st_uid;       /* пользователя владельца                  */
   gid_t   st_gid;       /* идентификатор группы владельца          */
   dev_t   st_rdev;      /* тип устройства для файлов устройств     */
   off_t   st_size;      /* размер файла в байтах (если определен   */
                         /* для данного типа файлов */
   unsigbed long st_blksize; /* размер блока для файловой системы   */
   unsigned long st_blocks;  /* число выделенных блоков             */
   time_t   st_atime;        /* время последнего доступа к файлу    */
   time_t   st_mtime;        /* время последней модификации файла   */
   time_t   st_ctime;        /* время создания файла                */
};
```

Для определения типа файла можно использовать следующие логические макросы применяя их к значению поля st_mode:

S_ISLNK(m) – файл типа “символическая связь”?

S_ISREG(m) – регулярный файл?

S_ISDIR(m) – директория?

S_ISCHR(m) – специальный файл символьного устройства?

S_ISBLK(m) – специальный файл блочного устройства?

S_ISFIFO(m) – файл типа “FIFO” (именованный “pipe”)?

S_ISSOCK(m) – файл типа “socket”?

Младшие 9 бит поля st_mode определяют права доступа к файлу подобно тому, как это делается в маске создания файлов текущего процесса.

Возвращаемое значение: системные вызовы возвращают значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов link()

Прототип системного вызова

```
#include <unistd.h>
int link(char *pathname, char *linkpathname);
```

Описание системного вызова: системный вызов link служит для создания жесткой связи к файлу с именем, на которое указывает параметр pathname**.** Указатель на имя создаваемой связи задается параметром linkpathname (полное или относительное имя файла). Во всех существующих реализациях операционной системы UNIX запрещено создавать жесткие связи к директориям, в Linux по неизвестной причине дополнительно запрещено создание связей к файлам устройств ввода-вывода.

Возвращаемое значение: значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов symlink()

Прототип системного вызова

```
#include <unistd.h>
int symlink(char *pathname, char *linkpathuame);
```

Описание системного вызова: системный вызов symlink служит для создания символической (мягкой) связи к файлу с именем, на которое указывает параметр pathname. Указатель на имя создаваемой связи задается параметром linkpathuame (полное или относительное имя файла). Никакой проверки реального существования файла с именем pathname системный вызов не производит.

Возвращаемое значение: значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов unlink()

Прототип системного вызова

```
#include <unistd.h>
int unlink(char *pathname);
```

Описание системного вызова: системный вызов unlink служит для удаления имени, на которое указывает параметр pathname, из файловой системы. Если после удаления имени счетчик числа жестких связей у данного файла стал равным 0, то возможны следующие ситуации:

- Если в операционной системе нет процессов, которые держат данный файл открытым, то файл полностью удаляется с физического носителя.

- Если удаляемое имя было последней жесткой связью для регулярного файла, но какой либо процесс держит его открытым, то файл продолжает существовать до тех пор, пока не будет закрыт последний файловый дескриптор, ссыпающийся на данный файл.

- Если имя относится к файлу типа socket, FIFO или к специальному файлу устройства, то файл удаляется независимо от наличия процессов, держащих его открытым, но процессы, открывшие данный объект, могут продолжать пользоваться им.

- Если имя относится к файлу типа «символическая связь», то он удаляется, и мягкая связь оказывается разорванной.

Возвращаемое значение: значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Задание к лабораторной работе

1. Провести подготовку данных следующим образом: в директории input, в которой находятся файлы данных, использовавшиеся в предыдущих лабораторных работах, создать на каждый из этих файлов по две-три связи (жёстких и мягких).

2. Написать программу согласно заданному варианту. При этом для получения информации о файлах жёстких и мягких ссылок, использовать системный вызов **lstat**, и не использовать **stat**, так как **stat** с мягкими ссылками не работает, он выдаёт информацию не о файле мягкой ссылки, а о файле, на которую мягкая ссылка указывает.

Варианты:

1. Написать программу, которая должна для регулярных файлов в указанной директории определять номер их inode и выводить эти номера *без дублирования*, по одному на строку, отсортированные по возрастанию. Если на один и тот же inode ссылаются несколько файлов этой директории, их количество должно быть выведено на той же строке, что и соответствующий номер inode во второй колонке.

2. Написать программу, которая должна для файлов в указанной директории определять номер их inode, и если на один и тот же inode ссылаются несколько файлов, удалить все жёсткие ссылки, кроме одной в данной директории. Жёсткие ссылки из другой директории и мягкие ссылки не трогать.

3. Написать программу, которая должна выполнить один из вариантов из лабораторной работы №1 (на выбор), но при этом, каждый входной регулярный файл она должна прочитать только 1 раз. Если среди файлов в директории попадётся ссылка на уже прочитанный файл (то есть, файл с тем же inode), то этот файл надо игнорировать, но вывести об этом сообщение на экран.

4. Написать программу, определяющую номер inode, на который в заданной директории больше всего жёстких ссылок и вывести на экран номер этого inode и имена всех файлов, которые на него ссылаются. Если таких inode несколько, вывести их все.

5. Написать программу, определяющую номер inode, на который в заданной директории больше всего мягких ссылок и вывести на экран номер этого inode и имена всех файлов, которые на него ссылаются. Если таких inode несколько, вывести их все.

Во время отчёта по лабораторной работе будет проверяться знание основ устройства файловой системы Linux, знание команд оболочки, системных вызовов и функций для работы файлами, правильность работы написанных программ и понимание их исходного кода.

ЛАБОРАТОРНАЯ РАБОТА №5

1. Этапы компиляции. Изучение этапов компиляции с помощью компилятора gcc.
2. Лексический, синтаксический и семантический анализ как составные части этапа собственно компиляции.
3. Регулярные выражения как инструмент описания лексем в LEX.
4. Входной файл программы LEX.
5. LEX для создания лексического анализатора компилятора.

Этапы компиляции. Изучение этапов компиляции с помощью компилятора gcc

Процесс компиляции программы на языке С можно разбить на 4 основных этапа: обработка препроцессором, собственно компиляция, ассемблирование, линковка (связывание). Для других языков может отсутствовать обработка препроцессором, но остальные этапы (или им эквивалентные) присутствуют.

Препроцессор осуществляет подготовку исходного файла к компиляции – вырезает комментарии, добавляет содержимое заголовочных файлов (директива препроцессора #include), реализует раскрытие макросов (символических констант, директива препроцессора #define).

После препроцессинга наступает очередь **компиляции**. Компилятор преобразует исходный текст программы на языке высокого уровня в код на языке ассемблера. (Необходимо помнить, что в литературе компиляцией часто называют результат всех четырёх этапов, ведущий к созданию исполняемого файла, но здесь термин “компиляция” используется в узком смысле.).

Следующий этап – **ассемблирование** – это трансляция кода на языке ассемблера в машинный код. Результат операции – объектный файл. Объектный файл содержит блоки готового к исполнению машинного кода, блоки данных, а также список определенных в файле функций и внешних переменных (**таблицу символов**), но при этом в нем не заданы абсолютные адреса ссылок на функции и данные. Объектный файл не может быть запущен на исполнение непосредственно, но в дальнейшем (на этапе линковки) может быть объединен с другими объектными файлами (при этом, в соответствии с таблицами символов, будут вычислены и заполнены адреса существующих между файлами перекрестных ссылок).

На этапе **линкования**, которую производит часть компилятора, называемая линковщиком (или линкером), вычисляются адреса ссылок, добавляется код запуска и завершения программы, код вызова библиотечных функций, и в результате получается готовый исполняемый файл программы. Опции gc**c** позволяют прервать процесс компиляции на любом из этих этапов (см. табл. 14).

Таблица 14

Команда	Результат
gcc –E -o file.i file.c	**-E** прерывает процесс компиляции после обработки препроцессором. Результат обработки файла file.c помещается в файл file.i. По соглашениям gcc, файл с расширением .i содержит исходный код на языке С, не требующий обработки препроцессором. При его компиляции обработка препроцессором будет пропущена.
gcc -S file.c	**-S** прерывает процесс создания исполняемого файла после этапа компиляции. Задавать имя выходного файла с помощью опции -o в данном

	случае не требуется, **gcc** автоматически создаст file.s. Расширение .s стандартное для файлов с исходным кодом на языке ассемблера.
gcc -c file.c	**-c** прерывает процесс создания исполняемого файла после этапа ассемблирования. Результат помещается в файл file.o. Расширение .o стандартное для файлов с объектным кодом.
gcc file.c **gcc file.i** **gcc file.s** **gcc file.o**	В каждом из приведённых случаев проходят недостающие этапы компиляции. При отсутствии опции –o с именем выходного файла, результат – исполняемый файл – будет называться a.out.

Лексический, синтаксический и семантический анализ как составные части этапа собственно компиляции

Этап собственно компиляции принято разбивать на подэтапы, которые называются лексическим, синтаксическим и семантическим анализом.

Лексический анализ – процесс разделения входной последовательности символов на распознанные группы – **лексемы**, т.е. минимальные последовательности символов, имеющие смысл в языке программирования (константы, идентификаторы, ключевые слова и т.д.). Часто лексемы на выходе лексического анализатора называют **токенами**. Поток токенов с выхода лексического анализатора поступает на вход синтаксического анализатора.

Синтаксический анализ – процесс проверки последовательности токенов на соответствие формальной грамматике языка программирования. Часто к этому этапу относят построение некоторых структур данных (дерево синтаксического разбора, таблицы констант, идентификаторов и т.д.) для последующего этапа семантического анализа (при другом подходе построение таблиц идентификаторов и т.д. относят к этапу семантического анализа).

Семантический анализ – процесс построения и проверки структур данных, (в том числе, построенных на этапе синтаксического анализа) на соответствие семантическим (смысловым) требованиям языка программирования (например, описания всех идентификаторов один и только один раз, допустимость типов операндов в операциях и аргументов в вызовах функций и т.д.). Проверенные на отсутствие семантических ошибок структуры данных передаются на подэтап семантического синтеза, который относится к этапу синтеза кода.

Данная работа посвящена знакомству с лексическим анализом. Основной метод построения лексического анализатора – создание программного детерминированного конечного автомата (ДКА). ДКА принимает символы из входного потока, и в зависимости от очередного символа и от своего внутреннего состояния, производит некоторые действия, в том числе изменяет своё внутреннее состояние, создаёт и помещает в выходной поток токены и т.д. Алгоритм действия ДКА удобно описывать с помощью ориентированного графа его внутренних состояний. Вершинам графа соответствуют различные внутренние состояния ДКА, а дугам – возможные переходы между состояниями. ДКА работает дискретными шагами, при этом на каждом шаге одна вершина является текущей – это текущее внутреннее состояние автомата. Получив очередной символ из входного потока, ДКА анализирует его, и принимает решение, по какой из исходящих из текущей вершины дуг он перейдёт в следующее состояние. При этом, он совершает действия, ассоциированные с этой дугой.

В теории конечных автоматов существуют строго доказанные результаты, что множество слов, распознаваемых ДКА, может быть описано набором регулярных выражений, которые

являются удобным языком для описания лексем языка программирования. При этом, известны алгоритмы, позволяющие по набору лексем построить соответствующий ДКА. Они используются в т.н. **генераторах лексических анализаторов.** В Linux самым известным генератором лексических анализаторов является программа Lex, с помощью которой были созданы лексические анализаторы первых версий gcc (для более поздних версий лексический анализатор, созданный Lex-ом, был вручную улучшен с целью увеличения производительности, а при создании современных версий gcc использовались усовершенствованные версии Lex, например FLEX).

Регулярные выражения как инструмент описания лексем в Lex

Регуля́рные выраже́ния (*regular expressions*) – формальный язык (то есть, набор правил) поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (*wildcard characters*). Для поиска используется строка-образец (*pattern*, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы.

Основные обозначения регулярных выражение Lex (см. табл. 15).

Таблица 15

Символы и метасимволы	**Значение**
Подстановочные знаки	
.	Соответствует любому одиночному символу (букве, цифре или символу _)
?	Означает, что предшествующий символ может присутствовать или отсутствовать в строке (то есть присутствует 0 или 1 раз)
+	Означает, что предшествующий символ присутствует и может повторяться несколько раз (то есть присутствует 1 раз и более)
*	Означает, что предшествующий символ может отсутствовать, присутствовать или повторяться несколько раз (присутствует 0 раз и более)
|	Означает соответствие ИЛИ. Нельзя использовать в конце выражения
/	Стоящее перед / выражение учитывается только тогда, когда после него сразу следует выражение, стоящее после / Пример: ab/cd – ab в строке abcd соответсвует, ab в строке abdc не соответствует
Позиция в строке	
^	Означает, что следующие за ним символы находятся в начале строки
$	Означает, что стоящие перед ним символы находятся в конце строки
Группировка	
( )	Означает, что заключенные в скобки символы присутствуют в строке указанном порядке (могут быть в любом месте строки) Также используется для группировки других регулярных выражений
[ ]	Означает, что заключенные в скобки символы присутствуют в строке в любом порядке (и в любом месте)
[^]	Означает, что заключенные в скобки символы не присутствуют в строке
-	Создает в скобках диапазон символов, которые могут присутствовать в любом месте строки. Примеры возможных диапазонов: 0-9 – все цифры, a-z – все строчные буквы (латинские), A-Z – все заглавные буквы (латинские). Диапазоны можно указывать подряд. Пример: a-zA-Z – любые латинские буквы

Последовательности со специальным значением	
\n	Символ новой строки
\t	Символ табуляции
\b	Возврат курсора на один шаг назад
Исключения	
****	Означает, что следующий символ, который может быть метасимволом регулярного выражения, в данном случае нужно интерпретировать буквально, а не как метасимвол. Пример: \\ - символ обратной косой черты
" "	Любая последовательность символов, заключённых в кавычки, теряет своё специальное значение и интерпретируется, как последовательность символов. Пример: "\n" – не перевод строки, а символы \ и n, идущие подряд. **Внимание:** Вне квадратных скобок любой (а для единообразия и в них тоже) пробел в регулярном выражении надо заключать в кавычки (особенность Lex)
Подстановки Lex	
{name}	Будет подставлено определение **name** (определённое в программе Lex регулярное выражение с эти именем)
{n,m}	Здесь **n** и **m** натуральные числа, **n<m** – число возможных вхождений регулярного выражения, стоящего перед скобками. Пример: x{2,4} – от 2 до 4 вхождений x
Учёт состояния конечного автомата лексического анализатора (объяснение будет дано ниже)	
<state>	Идущее после < > регулярное выражение учитывается, только если конечный автомат лексического анализатора находится в состоянии **state**. Пример: **<START>x** – x будет учитываться только, если конечный автомат лексического анализатора находится в состоянии START

Входной файл для программы LEX

Установленный LEX для систем Linux включает следующие файлы:

/usr/ccs/bin/lex
/usr/ccs/lib/lex/ncform
/usr/lib/libl.a
/usr/lib/libl.so
lex.yy.c;

В каталоге /usr/ccs/lib/lex имеется файл-заготовка ncform, который LEX используется для построения лексического анализатора (ЛА). Этот файл является уже готовой программой лексического анализа. Но в нем не определены действия, которые необходимо выполнять при распознавании лексем, отсутствуют и сами лексемы, не сформированы рабочие массивы и т.д. С помощью команды lex файл ncform достраивается. В результате мы получаем файл со стандартным именем lex.yy.c. Если LEX-программа размещена в файле program.l, то для получения программы ЛА с именем scanner необходимо выполнить следующий набор команд:

```
lex program.l
gcc lex.yy.c -ll -o scanner
```

Если имя входного файла для команды lex не указано, то будет использоваться файл стандартного ввода. Флаг -ll требуется для подключения /usr/ccs/lib/libl.a – библиотеки LEX. Если необходимо получить самостоятельную программу (как в предыдущем примере) лексического анализа, подключение библиотеки обязательно, поскольку из нее подключается главная функция main. В противном случае, если нужный main находится в файле LEX-

программы, библиотеку не надо подключать. А если имеется необходимость включить ЛА в качестве функции в другую программу (например, в программу синтаксического анализа), эту библиотеку необходимо вызвать уже при сборке конечной программы. Тогда, если main определен в вызывающей ЛА программе, редактор связей не будет подключать раздел main из библиотеки LEX.

Структура LEX-программы

LEX-программа, написанная на языке LEX, имеет следующий формат:

```
определения
%%
правила
%%
пользовательские_подпрограммы
```

То есть, LEX-программа включает секции определений (definitions), правил (rules) и пользовательских подпрограмм (user subroutines). Любая из этих секций может быть пустой. Простейшая LEX - программа имеет вид:

```
%%
```

Здесь нет никаких определений и никаких правил. Только разделитель между первой и второй секциями обязателен (даже при их отсутствии), второй разделитель – не обязателен (при отсутствии третьей секции). Все строки, в которых занята первая позиция, относятся к LEX-программе. Любая строка, не являющаяся частью **правила** или **действия**, которая начинается с пробела или табуляции (то есть, сама строка не начинается с первой позиции), копируется в генерируемый файл lex.yy.c.

Секция определений LEX-программы

Определения, предназначенные для LEX, помещаются перед первым %%. Любая строка этой секции, не содержащаяся между %{ и %}, и начинающаяся в первой колонке, является определением LEX. Секция определений LEX-программы может включать:

- начальные условия;
- определения;
- фрагменты пользовательских подпрограмм;
- комментарии в формате хост-языка;
- указатели хост-языка;
- таблицы наборов символов;
- изменения размеров внутренних массивов.

В данной работе последние три пункта не используются, поэтому ниже кратко описаны только первые четыре.

Начальные условия (состояния конечного автомата) задаются в форме:

```
%START имя1 имя2 ...
```

Если начальные условия определены, то эта строка должна быть первой в LEX-программе.

Определения задаются в форме:

```
name translation
```

В качестве разделителя используется один или более пробелов или табуляций. Имя name – это, как обычно, любая последовательность букв и цифр, начинающаяся с буквы. translation – это регулярное выражение (или его часть), которое будет подставлено всюду, где указано имя.

Пример:

```
LETTER          [A-ZA-Яа-za-я_]
DIGIT           [0-9]
IDENTIFIER      {LETTER}({LETTER}|{DIGIT})*
. . .
%%
{IDENTIFIER}    printf("\n%s", yytext);
. . .
```

В этом примере по данным определениям и одному правилу из раздела правил (первая строка после %%) LEX построит ЛА, который будет, в том числе, находить и выводить на стандартный вывод все идентификаторы из входного файла, сформированные по правилу “непустая последовательность букв и цифр, начинающаяся с буквы”. Здесь IDENTIFIER будет сначала заменён на

LETTER (LETTER | DIGIT)*

(см. раздел “Подстановки Lex” в таблице выше), а затем на

A-ZA-Яа-za-я(A-ZA-Яа-za-я|0-9)*

Имя yytext – это т.н. внешний массив символов программы, которую строит LEX; yytext формируется в процессе чтения входного файла и содержит текст, для которого установлено соответствие текущему регулярному выражению из раздела определений (правой части определения), или левой части текущего активного правила из секции правил (см. ниже). Этот массив доступен всем функциям LEX-программы.

Фрагменты пользовательских подпрограмм вводятся двумя способами:

отступ фрагмент;

(то есть, начинается не с первой колонки) или

```
%{
фрагмент
%}
```

Включение пользовательского фрагмента, начинающегося с первой колонки строки, необходимо, например, для ввода макроопределений Си, которые должны начинаться в первой колонке строки. Все строки фрагмента пользовательской подпрограммы, размещенные в разделе определений, будут, как правило, помещаться в начало файла lex.yy.c. и будут внешними для любой функции из файла lex.yy.c.

Комментарии в разделе определений задаются в формате хост-языка (то есть, языка, на котором будет написан ЛА) и должны начинаться не с первой колонки строки. Хост-языком по умолчанию является С, это можно изменить с помощью указателя хост-языка.

Секция правил LEX-программы

Все, что находится после первой пары %% и до второй пары %%, если она присутствует (или до конца LEX-программы, если третья секция отсутствует), относится к секции правил. Секция правил может содержать правила и фрагменты пользовательских подпрограмм.

Секция правил может включать список активных и неактивных (помеченных) правил. Активные и неактивные правила могут располагаться в любом порядке. Любое правило должно начинаться с первой позиции строки, пробелы и табуляции в начале строки с правилом недопустимы.

Активное правило имеет вид:

```
reg_expr  action
```

Активные правила выполняются всегда. По регулярным выражениям (reg_expr), содержащимся в левой части правил, LEX строит С-программу, реализующую детерминированный конечный автомат.

Действие (action) может быть либо **специальным действием LEX**, либо блоком или оператором Си. Действие в правиле записывается не менее чем через один пробел или табуляцию после регулярного выражения (начинается обязательно в той же строке), а его продолжение может быть в следующих строках только в том случае, если действие оформлено как блок или оператор Си-программы. Действия в правилах LEX-программы выполняются, если правило активно, т.е., если ЛА распознает цепочку символов из входного потока как соответствующую регулярному выражению данного правила.

Если входная цепочка символов не распознана, то она просто копируется из входного потока в выходной. Комбинация символов, не учтенная в правилах и появившаяся на входе, будет напечатана на выходе. Можно сказать, что копирование осуществляется "по умолчанию", а действие - это то, что делается вместо копирования в случае успешного распознавания лексемы. Таким образом, ЛА, построенный на основе только простейшей LEX-программы (пустой программы) и файла-заготовки ncform, будет просто копировать входной поток в выходной.

Пример простейшего активного правила, удаляющего пробельные символы и переводы строки из входного потока:

```
[ \t\n]    ;
```

здесь в качестве действия используется пустой оператор С.

Специальные действия языка Lex: **| ECHO REJECT BEGIN.** Если для нескольких регулярных выражений надо указать одно и то же действие, то для упрощения записи используется специальное действие |, которое свидетельствует о том, что действие для данного регулярного выражения совпадает с действием для следующего. Например,

```
" " |
\t  |
\n  ;
```

аналогично правилу из предыдущего примера.

Специальное действие **ECHO;** – это сокращённая форма записи для printf("%s",yytext); – то есть, для вывода распознанной левой части правила на печать.

Когда LEX ищет правило, соответствующее текущему вводу, он из всех подходящих активирует то, в котором левая часть самая длинная. А если левые части у двух подходящих правил одинаковы по длине, активирует то, которое в списке правил стоит раньше. И далее эту часть текста больше не проверяет на соответствие другим правилам. Если же после срабатывания правила эту часть текста надо проверять на совпадение дальше, то в описании действия надо использовать **REJECT**. Пример:

```
she   {s++; REJECT;}
he    {h++; REJECT;}
```

посредством этих правил в переменной s подсчитывается число вхождений в текст последовательности **she**, а в переменной h – число вхождений последовательности **he**. Если не поставить в правиле для she REJECT; то тот he, который является частью she, в h подсчитан не будет.

Специальное действие BEGIN – перевод конечного автомата ЛА в заданное состояние:

```
BEGIN state;
```

Имя этого состояния должно быть задано в строке %START в секции определений. Делается это для того, чтобы можно было активировать условные правила – те правила, которые зависят от состояния конечного автомата. Они имеют вид

```
<state>reg_expr  action
```

и активируются только тогда, когда конечный автомат ЛА находится в состоянии state, и в нём читает из входного потока строку, соответствующую reg_expr. Перевод конечного автомата в исходное положение (которое было до считывания из входного потока первого символа) производится специальным действием BEGIN 0; .

Если имеется необходимость выполнить достаточно большой набор преобразований, то действие оформляют как блок Си-программы (он начинается открывающей фигурной скобкой и завершается закрывающей), содержащий необходимые фрагменты. Область действия переменных, объявленных внутри блока, распространяется только на этот блок. Внешними переменными для всех действий будут только те переменные, которые объявлены в секции определений.

Фрагменты пользовательских подпрограмм, содержащиеся в секции правил (но не входящих в какое-либо правило), просто копируются внутрь функции yylex файла lex.yy.c. Если необходимо, чтобы в них строка начиналась с первой колонки (например, как инструкции препроцессору, начинающиеся с #), то они, аналогично секции определений, вводятся так:

```
%{
фрагмент
%}
```

Пример:

```
%%
%{
#include file.h
%}
```

Здесь строка #include file.h станет строкой функции yylex (а не внешней по отношению ко всем функциям файла lex.yy.c, как фрагменты пользовательских подпрограмм из секции определений).

Секция пользовательских подпрограмм LEX-программы

Все, что размещено после второй пары %%, относится к секции пользовательских подпрограмм. Содержимое этой секции копируется в выходной файл `lex.yy.c` без каких-либо

изменений. Как правило, там находятся определения функций, которые могут вызываться в действиях правил. Также там может находится функция main, если стандартная функция main из библиотеки libl не подходит.

Комментарии можно вводить во всех разделах LEX-программы. Формат комментариев должен соответствовать формату комментариев хост-языка, т.е. языка Си. Однако в каждой секции LEX – программы комментарии вводятся по-разному:

1. В секции определений комментарии должны начинаться не с первой позиции строки;
2. В секции правил комментарии можно записывать только внутри блоков, принадлежащих действиям;
3. В секции подпрограмм комментарии записываются как в Си.

Специальные конструкции и функции Lex (см. табл. 16).

Таблица 16

Имя	Описание
yyin, yyout	стандартные потоки ввода и вывода Lex
yylex()	построенная функция ЛА
yytext	указатель на распознанную цепочку символов, оканчивающуюся нулевым символом
yyleng	длина этой цепочки
yyunput(c)	поместить символ с во входной поток
yyless(n)	вернуть последние n символов цепочки yytext обратно во входной поток
yymore()	считать следующие символы в буфер yytext после текущей цепочки
yywrap()	вызывается при достижении конца входного файла

Пример Lex программы:

```
            /*  Программа  */
%{
#include <stdio.h>
%}

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD\n");
[0123456789]+           printf("NUMBER\n");

%%
/* Конец программы ------------------------------------*/
```

LEX для создания лексического анализатора компилятора

В рассмотренном выше примере, с помощью LEX написана самостоятельная программа, осуществляющая некоторую обработку текста. Однако, главное назначение LEX, для которого он создавался – это быть генератором лексических анализаторов для использования в компиляторах. При этом, кроме собственно выделения лексем языка, лексическому анализатору необходимо производить ещё некоторую дополнительную работу для последующего использования синтаксическим анализатором. Рассмотрим некоторые аспекты этого процесса.

Прежде всего, в компиляторе лексический анализатор, как правило, реализуется в виде функции, к которой обращается синтаксический анализатор, и которая должна возвращать ему

очередной токен из входного потока. При этом, кроме типа токена, нужна ещё и другая информация, например, значение токена: для числовой константы это её значение, для строковой константы – указатель на начало соответствующей строки, для идентификатора – указатель на строку в таблице имён, где его имя записано (такую таблицу, соответственно, надо организовать), и т.д.

Изначально LEX предназначался для работы в паре с генератором синтаксических анализаторов YACC, и потому многие аспекты их работы согласованы друг с другом, мы будем обращать на это внимание, а с самим YACC-ом познакомимся на следующей лабораторной работе.

Рассмотрим пример лексического анализатора для программы-калькулятора, вычисляющего значения арифметических выражений с вещественными (двойной точности) числами в формате C, синтаксический анализатор для которых написан на YACC. При этом также можно присваивать значения переменным, и использовать математические функции.

```
            /*  Программа LEX сканера для калькулятора */
%{
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "y.tab.h"
#include "symbol_table.h"
struct symtab symtab[NSYMS];
%}

%%
[ \t]   ;    /* ignore whitespace */

([0—9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)  {
  yylval.dval = atof(yytext);
  return NUMBER;
  }

[A-Za-z][A-Za-z0-9]*  {
   struct symtab *sp = symlook(yytext);
   yylval.symp = sp;
   return NAME;
   }

"$"      { return 0;}

\n     |
.  return yytext[0];
%%

/* look up a symbol table entry, add if not present */

struct symtab *symlook(char *s)
```

```
{
struct symtab *sp;

for( sp = symtab; sp< &symtab[NSYMS]; sp++){
   if(sp->name && !strcmp(sp->name, s)) return sp; /* name is found */
   if(!sp->name){sp->name = strdup(s);  return sp;} /* make new row */
   } /* otherwise continue to the next row until end of table */

yyerror("Too many symbols");
exit(1);    /* cannot continue */

}  /* symlook */

/* Конец программы сканера ---------------------- */

     /* заголовочный файл symbol_table.h */

#define NSYMS 20 /* размер таблицы имён */

struct symtab {
char *name;             /* имя переменной или функции */
double (*funcptr)(double); /* для функции указатель на её вычисление*/
double value;               /* для переменной её значение */
};

struct symtab *symlook(char*);

/* конец файла symbol_table.h ---------------------- */
```

Файл **y.tab.h** формируется YACC-ом и содержит определения имён токенов, как целых констант для C (в данном случае – это NUMBER и NAME), и определение союза (union) **yylval**, куда LEX должен помещать значения токенов (для NUMBER это значение числа (поле dval), для NAME – указатель на строку таблицы имён **symtab** (поле symp), куда LEX поместил соотвествующее имя с помощью функции **symlook**). Символ $ должен быть последним символом ввода. Символы, не входящие в лексемы типа NUMBER и NAME, кроме пробелов, копируются на выход.

Задание к лабораторной работе

1. Напишите простейшую программу на C (например, “Hello, world!”) и изучите с её помощью результаты различных этапов компиляции программы. В программе на языке ассемблера попробуйте определить соответствие между операторами программы и соответствующими им командами ассемблера.

2. Изучите пример программы для Lex, и определите, что она делает. Создайте соответствующий файл, с помощью Lex и gcc получите исполняемый файл, и проверьте правильность Ваших выводов.

3. Напишите программу на Lex согласно заданному варианту. Убедитесь в правильности её работы.

Варианты:

1. Написать программу, которая принимала бы на входе программу на C, а на выходе заменяла бы на названия типов лексем сами лексемы языка для следующих объектов:

- символьные константы на CHAR_CONST,
- строковые константы на STRING_CONST.
- вещественные константы на FLOAT_CONST.

Остальной текст программы оставить без изменений.

2. Написать программу, которая заменяла бы в тексте целые числа на INTEGER и вещественные числа на FLOAT (и те и другие – в формате C). Остальной текст оставить без изменений.

3. Написать программу, которая заменяла бы в тексте каждое предложение (то есть, последовательность слов, у первого из которых первая буква заглавная, а заканчивается последовательность точкой), на SENTENCE (N characters), где N число символов в предложении (всех, включая пробельные).

4. Изучите программу LEX сканера для калькулятора, объясните назначение и смысл различных её элементов.

Во время отчёта по лабораторной работе будет проверяться знание опций gcc, знание основ работы с Lex, правильность работы написанных программ и понимание их исходного кода.

ЛАБОРАТОРНАЯ РАБОТА №6

1. Синтаксический анализ и формальные грамматики.
2. Программа построения синтаксических анализаторов YACC.
3. Программа YACC для калькулятора.
4. Компиляция программы калькулятора с помощью make.

Синтаксический анализ и формальные грамматики

Синтаксический анализ в узком смысле – это процесс проверки последовательности лексем (токенов) на соответствие формальной грамматике языка программирования. Если формальная лексика языка программирования может быть описана в терминах регулярных выражений и для лексического анализа может быть построен конечный автомат, то для синтаксического анализа этот математический аппарат не обладает достаточной мощностью. Например, конечный автомат не может распознавать вложенные скобки с неограниченным уровнем вложенности в выражениях и операторах. Поэтому для синтаксического анализа используется механизм т.н. **контестно-свободных формальных грамматик**, соответствующие абстрактные вычислительные устройства для которых – **стековые автоматы** (от конечных автоматов они отличаются тем, что у них есть ещё бесконечный стек, куда могут помещаться токены, прочитанные из входного потока, и действия автомата на каждом шаге теперь будут зависеть не только от внутреннего состояния и входного токена, но и от состояния стека).

Традиционно конструкции, распознаваемые лексическим анализатором (токены), на этапе синтаксического анализа называют **терминальными символами**, а составные из токенов конструкции, обозначаемые в формальных грамматиках едиными именами – **нетерминальными символами (нетерминалами)**. В дальнейшем, во избежание путаницы, терминальные символы мы будем продолжать называть токенами (tokens).

Формальная грамматика – это набор правил преобразования текстов. Правило состоит из двух частей, левой и правой, каждая из которых в общем случае представляет собой строку из токенов и нетерминалов. Также, с правилом может быть ассоциирована совокупность действий, которые надо выполнить при активизации правила. Синтаксический анализатор, читая входной поток, ищет строку, совпадающую с **правой** частью правила, и когда находит её, то заменяет **правую часть на левую** и выполняет ассоциированные с правилом действия (если они есть). Среди всех нетерминалов грамматики выделен один нетерминал, называемый **начальным**. В результате применения к потоку токенов правил грамматики, при отсутствии синтаксических ошибок в конце должен остаться только этот самый начальный нетерминал (правило, в котором левой частью является начальный нетерминал, обязательно должно присутствовать в грамматике, хотя бы одно). В этом случае синтаксический разбор успешно закончен.

Такая странная, на первый взгляд, терминология и действия, происходят из того, что в разделе дискретной математики, занимающейся задачами преобразования текстов – теории формальных языков и грамматик, традиционно рассматривается не процесс анализа готового текста, а **процесс порождения текста**, удовлетворяющего заданным правилам. Процесс порождения синтаксически правильной (относительно некоторой формальной грамматики) программы, тогда выглядит следующим образом: он начинается со строки, в которой

находится один только начальный нетерминал, и на первом шаге активизируется какое-либо правило, где он является **левой** частью – он заменяется на правую часть. Потом в той строке, что получилась, ищется левая часть какого-либо правила и заменяется на правую часть правила и т.д., до тех пор пока в строке не останется нетерминалов. Как только в строке останутся одни токены, процесс порождения какой-то синтаксически правильной программы будет закончен.

А процесс синтаксического анализа программы – это процесс обратный к порождению программы, и потому при использовании грамматики, предназначенной для порождения, приходится действовать так, как описано выше. Но именно порождающие грамматики используются в синтаксическом анализе, потому что, как показала практика, такие грамматики для человека гораздо легче создать при разработке нового языка, чем “работающие в обратную сторону”. Алгоритмы, которые по порождающей формальной грамматике, создают программы синтаксического анализа, называются **генераторами синтаксических анализаторов**, или **компиляторами компиляторов** (это традиционное название, потому что в реальном компиляторе, как отмечалось выше, есть ещё много других этапов).

Контекстно-свободные грамматики, использующиеся при описании языков программирования, это такие грамматики, левая часть правил которых состоит всегда только из одного нетерминала. Рассмотрим пример простой грамматики такого рода, и того, как она может быть применена для синтаксического анализа некоторым стековым автоматом. Грамматика (левая и правая часть разделяются символом →, имена токенов пишутся заглавными буквами, имена нетерминалов – строчными):

statement → NAME = expression
expression → NUMBER + NUMBER
expression → NUMBER - NUMBER

Начальный символ - statement. Пусть изначально в обрабатываемом файле содержалась строка:

```
fred = 12 + 13
```

После лексического анализа (по правилам языка С, для простоты “;” в конце оператора рассматривать не будем, а “=”, “+”,”-“ будем считать токенами), она превратиться в строку токенов

NAME = NUMBER + NUMBER

Далее синтаксический анализатор будет действовать следующим образом:

- читает NAME и помещает его в стек, пока никакое правило не распознал;
- читает = и помещает его в стек, пока никакое правило не распознал;
- читает NUMBER и помещает его в стек, пока никакое правило не распознал;
- читает + и помещает его в стек, пока никакое правило не распознал;
- читает NUMBER и помещает его в стек, распознаёт в стеке правую часть второго правила;
- заменяет в стеке распознанную правую часть (три верхних токена) на левую часть – нетерминал expression;
- распознаёт в стеке правую часть первого правила;
- заменяет в стеке распознанную правую часть (три верхних токена) на левую часть – нетерминал statement;

- обнаруживает конец входного потока, смотрит, что осталось в стеке – в стеке находится только начальный нетерминал statement – синтаксический анализ успешно завершён.

Программа построения синтаксических анализаторов YACC

Yacc на основе входных спецификаций осуществляет синтаксический разбор текста, пользуясь при этом результатами лексического анализа, выполняемого внешней (по отношению к нему) подпрограммой, которая должна читать символы из входного потока, распознавать конструкции низкого уровня (лексемы) и передавать их на вход синтаксического анализатора, создаваемого Yacc.

Разбираемый текст может (иногда) не соответствовать спецификациям, что является синонимом синтаксической ошибки. Такие ошибки обнаруживаются настолько быстро, насколько это возможно теоретически при сканировании слева направо. Частью входных спецификаций являются специальные правила обработки ошибок, позволяющие либо исправить неправильные символы, либо продолжить разбор после пропуска неправильных символов.

В некоторых случаях компилятором Yacc не удастся построить анализатор по входным спецификациям, так как либо заданная грамматика не является LALR(1) (частный случай контекстно-свободной грамматики, здесь рассматриваться не будет), либо у Yacca не хватает мощности. Обычно эта проблема решается пересмотром системы правил, либо созданием более мощного лексического анализатора. Следует также заметить, что конструкции, неподвластные Yacc, часто также непосильны и человеческому разуму...

Входные спецификации

Имена относятся либо к токенам, либо к нетерминалам. Компилятор Yacc требует, чтобы все токены были специально описаны. Файл входных спецификаций состоит из трех секций: объявлений (declarations); грамматики (правил/rules); программ (подпрограмм) пользователя.

Секции отделяются друг от друга двойным процентом (%%). (Процент вообще используется Yaccом как специальный символ). Другими словами, файл входных спецификаций выглядит так:

```
declarations (объявления)
%%
rules (правила)
%%
programs (программы)
```

Секция объявлений может быть пустой, кроме того, если пуста секция программ, можно опустить и второй разделитель (%%). Таким образом, самая короткая спецификация может выглядеть так:

```
%%
rules
```

Пробелы, табуляции и возвраты каретки игнорируются, но не должны появляться в именах и зарезервированных символах. Везде, где может стоять имя, может быть и комментарий (как в Си):

```
/* . . . */
```

Секция правил состоит из одного или более грамматических правил в форме:

```
A : BODY ;
```

здесь A – имя нетерминала, а BODY представляет собой последовательность из нуля или более имен и **литералов**. Двоеточие и точка с запятой – символы пунктуации Yacca. Имена могут быть произвольной длины и состоять из букв, точек, подчеркиваний и цифр, но не должны начинаться с цифры. Прописные и строчные буквы различаются. Имена, использованные в теле правила, могут быть именами токенов или нетерминалов. Литерал – это символ, заключенный в одиночные кавычки. Как и в Си бэкслэш (\) является спецсимволом, а все спецпоследовательности языка С типа \n и \t воспринимаются.

По нескольким причинам технического свойства символ с кодом 0 (NULL) не должен никогда использоваться в правилах.

Если есть несколько правил с одинаковой левой частью, можно вместо дублирования левой части использовать символ |. Кроме того точка с запятой может быть поставлена и перед символом |. Таким образом, правила:

```
A : B C D ;
A : E F ;
A : G ;
```

могут быть записаны так:

```
A : B C D
| E F
| G
;
```

или даже так:

```
A : B C D ;
| E F ;
| G
;
```

Не обязательно, чтобы правила с одинаковой левой частью были написаны вместе (или рядом), но такая запись делает их более читаемыми и легкими для изменения. Если нетерминалу соответствует пустая строка, он может быть записан так:

```
empty : ;
```

Имена токенов должны быть объявлены в секции объявлений:

```
%token name1 name2 . . .
или
%token name1
%token name2
. . .
```

Если для разных токенов их значения имеют разный тип, то в секции объявлений должно быть объявление

```
%union {
    type1 field1;
    type2 field2;
   . . .
    }
```

где перечислены все необходимые типы, и имена полей этого типа. По этому объявлению YACC создаст в заголовочном файле **y.tab.h,** предназначенном для лексического анализатора, глобальную переменную типа союз (union) с данной структурой, с именем

yylval, куда лексический анализатор должен помещать значение распознанного токена перед тем, как передать его тип синтаксическому анализатору (соблюдая соответствие типов). Для самого синтаксического анализатора какие поля соответствуют какому типу токена, должно быть задано в объявлении токена следующим образом:

```
%token <field1> name1
%token <field2> name2
. . .
```

Пример: если для типа токена типа NUMBER значением является число типа double, а для токена типа NAME – указатель на место в таблице имён, то в секции объявлений это можно оформить так:

```
%union {
   double dval;
   struct symtab *symp;
   }
%token <symp> NAME
%token <dval> NUMBER
```

Каждое имя, не указанное в секции объявлений, как токен, предполагается именем нетерминала. Каждый нетерминал должен появиться в левой части хотя бы одного правила. Тип значения нетерминала (если нужен) должен присутствовать среди полей в объявлении %union, и после этого указан в секции объявлений как:

```
%type <fild> nonterminal
```

Пример: если у нетерминала expression значении типа double, то используя %union из предыдущего примера, можно объявить его тип следующим образом:

```
%type <dval> expression
```

Из всех нетерминалов, один, называемый **начальным нетерминалом**, имеет особое значение. Анализатор строится, чтобы разобрать начальный нетерминал, который представляет собой наибольшую и наиобщую конструкцию, описанную правилами. По умолчанию, начальным считается нетерминал, стоящий в левой части первого правила, но можно (и желательно), чтобы он был явно объявлен в секции объявлений как

```
%start  nonterminal
```

Об окончании ввода анализатору говорит специальный токен, называемый **концевым маркером**. Если концевой маркер приходит после разбора правила, соответствующего начальному нетерминалу, то прочитав его, анализатор возвращает управление вызвавшей программе (это успешное завершение синтаксического разбора). Если маркер приходит в ином контексте, это считается синтаксической ошибкой. Вернуть концевой маркер (с кодом 0 или отрицательным) в нужный момент – это забота лексического анализатора. Обычно концевой маркер возвращается по концу записи или файла.

Действия

Каждому правилу можно поставить в соответствие некое действие, которое будет выполняться всякий раз, как это правило будет распознано. Действия могут возвращать значения и могут пользоваться значениями, возвращенными предыдущими действиями. Более того, лексический анализатор может возвращать значения для токенов (дополнительно), если надо. Действие - это обычный оператор языка Си, который может выполнять ввод, вывод, вызывать подпрограммы и изменять глобальные переменные.

Действия, состоящие из нескольких операторов, необходимо заключать в фигурные скобки. Например:

```
A : '(' B ')'
{ hello( 1, "abc" ); }
```

и

```
XXX : YYY ZZZ
{ printf("a message\n"); flag = 25; }
```

являются грамматическими правилами с действиями.

Чтобы обеспечить передачу данных между действиями и правилами, используется спецсимвол "доллар" ($). Чтобы вернуть значение, действие обычно присваивает его псевдопеременной $$. Например, действие, которое не делает ничего, но возвращает единицу:

```
{ $$ = 1;}
```

Чтобы получить значения, возвращенные действиями и лексическим анализатором, действие может использовать псевдопеременные $1, $2 и т. д., которые соответствуют значениям, возвращенным компонентами правой части правила, считая слева направо. Например, если правило имеет вид:

```
A : B C D ;
```

то $2 соответствует значению, возвращенному нетерминалом C, а $3 – нетерминалом D. Более конкретный пример:

```
expr : '(' expr ')' ;
```

Значением, возвращаемым этим правилом, логично считать значение выражения в скобках, что может быть записано так:

```
expr : '(' expr ')'
{ $$ = $2 ; }
```

По умолчанию, значением правила считается значение, возвращенное первым элементом ($1). Таким образом, если правило не имеет действия, Yacc автоматически добаляет его в виде $$=$1; , благодаря чему для правила вида

```
A : B ;
```

обычно не требуется самому писать действие.

В большинстве приложений действия не выполняют ввода/вывода, а конструируют и обрабатывают в памяти структуры данных, например дерево синтаксического разбора. Такие действия проще всего выполнять, вызывая подпрограммы для создания и модификации структур данных. Предположим, что существует функция node, написанная так, что вызов node(L, n1, n2) создает вершину с меткой L, ветвями n1 и n2 и возвращает индекс свежесозданной вершины. Тогда дерево синтаксического разбора может строиться так:

```
expr : expr '+' expr
  { $$ = node( '+', $1, $3 ); }
```

Программист может также определить собственные переменные, доступные действиям. Их объявление и определение может быть сделано в секции объявлений, заключенное в символы %{ и %}. Такие объявления имеют глобальную область видимости, благодаря чему доступны как действиям, так и лексическому анализатору. Например:

```
%{
int variable = 0;
%}
```

Такие строчки, помещенные в раздел объявлений, объявляют переменную variable типа int и делают ее доступной для всех действий. Все имена внутренних переменных Yacca начинаются с двух букв y, поэтому не следует давать своим переменным имена типа yymy.

Предварительный лексический анализ

Программист, использующий Yacc должен написать сам (или создать при помощи программы типа Lex) внешний лексический анализатор, который будет читать символы из входного потока (какого – это внутреннее дело лексического анализатора), обнаруживать терминальные символы (токены) и передавать их синтаксическому анализатору, построенному Yaccом (как правило, вместе с неким значением) для дальнейшего разбора. Лексический анализатор должен быть оформлен как функция с именем yylex, возвращающая значение типа int, которое представляет собой тип (номер) обнаруженного во входном потоке токена. Если надо вернуть еще некое значение (например, указатель на строку в таблице имён), оно может быть присвоено глобальной, внешней (по отношению к yylex) переменной по имени yylval.

Лексический анализатор и Yacc должны использовать одинаковые номера токенов, чтобы понимать друг друга. Эти номера обычно выбираются Yaccом, но могут выбираться и человеком. В любом случае механизм define языка Си позволяет лексическому анализатору возвращать эти значения в символическом виде. Предположим, что токен по имени DIGIT был определен в секции объявлений спецификации Yacca, тогда соответствующий текст лексического анализатора может выглядеть так:

```
yylex()
{
extern int yylval;
int c;
...
c = getchar();
...
switch (c) {
. . .
case '0':
case '1':
...
case '9':
yylval = c - '0';
return DIGIT;
. . .
}
. . .
```

Вышеприведенный фрагмент возвращает номер токена **DIGIT** и значение, равное цифре. Если при этом сам текст лексического анализатора был помещен в секцию программ спецификации Yacca – есть гарантия, что идентификатор **DIGIT** был определен номером токена **DIGIT**, причем тем самым, который ожидает Yacc.

Такой механизм позволяет создавать понятные, легкие в модификации лексические анализаторы. Единственным ограничением является запрет на использование в качестве имени токена слов, зарезервированных или часто используемых в языке Си. Например, использование

в качестве имен токенов таких слов как if или while, почти наверняка приведет к возникновению проблем при компиляции лексического анализатора. Кроме этого, имя error зарезервировано для токена, служащего делу обработки ошибок, и не должно использоваться.

Как уже было сказано, номера токенов выбираются либо Yaccом, либо человеком, но чаще Yaccом, при этом для отдельных символов (например для (или) выбирается номер, равный ASCII коду этого символа. Для других токенов номера выбираются, начиная с 257.

Для того, чтобы присвоить токену (или даже литере) номер вручную, необходимо в секции объявлений после имени токена добавить положительное целое число, которое и станет номером токена или литеры, при этом необходимо позаботиться об уникальности номеров. Если токену не присвоен номер таким образом, Yacc присваивает ему номер по своему выбору.

По традиции, концевой маркер должен иметь номер токена, равный, либо меньший нуля, и лексический анализатор должен возвращать ноль или отрицательное число при достижении конца ввода (или файла).

Очень неплохим средством для создания лексических анализаторов является программа Lex. Лексические анализаторы, построенные с ее помощью прекрасно гармонируют с синтаксическими анализаторами, построенными Yaccом. Lex можно легко использовать для построения полного лексического анализатора из файла спецификаций, основанного на системе регулярных выражений (в отличие от системы грамматических правил для Yacca), но, правда, существуют языки (например Фортран) не попадающие ни под какую теоретическую схему - для них приходится писать лексический анализатор вручную.

Программа YACC для калькулятора

```
%{
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "symbol_table.h"
extern struct symtab symtab[NSYMS];
/* если следующие две строки вызывают ошибку (error), убрать их */
/* если только warning, и компиляция проходит, не трогать */
extern int yylex(void);
int yyerror(const char* format){ return printf("%s\n", format); }
%}

%union {
   double dval;
   struct symtab *symp;
   }

%token <symp> NAME
%token <dval> NUMBER

%left '-' '+'
%left '*' '/'
%nonassoc UMINUS
```

```
%type <dval> expression

%start statement_list
%%
statement_list:  statement '\n'
     |           statement_list statement '\n'
     ;

statement: NAME '=' expression        { $1->value = $3; }
     |      expression           { printf("= %g\n", $1); }
     ;

expression: expression '+' expression { $$ = $1 + $3; }
     |      expression '-' expression { $$ = $1 - $3; }
     |      expression '*' expression { $$ = $1 * $3; }
     |      expression '/' expression
                              {
                              if( $3 == 0.0 )
                                { yyerror("divide by zero"; $$ =
                         $3);}
                              else $$ = $1 / $3;
                              }
     |      '-' expression %prec UMINUS   { $$ = -$2; }
     |      '(' expression ')'            { $$ =  $2; }
     |      NUMBER
     |      NAME                      { $$ = $1->value; }
     |      NAME '(' expression ')'
                             {
                             if( $1->funcptr ) $$ = $1->funcptr( $3 );
                        else {
                               printf("%s not a function", $1->name);
                               $$ = 0.0;}
                             }
     ;
%%
void addfunc(char *name, double (*func)(double))
{
struct symtab *sp = symlook(name);
sp->funcptr = func;
}

main()
{
extern double sqrt(double), exp(double), log(double);

addfunc("sqrt", sqrt);
addfunc("exp",  exp );
addfunc("log",  log );
```

```
yyparse();
}

/* конец программы YACC для калькулятора */
```

Комментарии к программе:

Заголовочный файл **symbol_table.h** приведён в лабораторной работе №5.

Объявления %left и %nonassoc относятся к определению приоритета операций. Объявление %left со списком литералов, обозначающих действия, объявляет, что указанные действия все имеют одинаковый приоритет, и должны выполняться слева направо (%right - справа налево). Объявление %nonassoc означает, что это действие относится к одному операнду. Порядок следования этих описаний в секции задаёт приоритеты среди групп операций – первое описание соответствует самому низкому приоритету, последнее – самому высокому. Имя UMINUS, использованное в объявлении унарного минуса, определяет так называемый **псевдотокен**. Когда в правиле грамматики встречается унарный минус, с помощью команды %prec UMINUS его приоритет приравнивается к приоритету псевдотокена UMINUS, то есть, становится наивысшим из всех здесь используемых.

Функция **symlook** описана в секции подпрограмм пользователя программы LEX для лексического анализатора в лабораторной работе №5, и объявлена, для возможности её корректного здесь использования, в файле **symbol_table.h.**

Компиляция программы калькулятора

Для получения исполняемого файла калькулятора, в одной директории необходимо иметь файл **lex.yy.c** – исходный текст лексического анализатора, полученный с помощью Lex в лабораторной работе №5, файл **symbol_table.h** и файл с программой Yacc для калькулятора, предположим, что он называется **lab6.y** (расширение **.y** считается общепринятым для программ Yacc). Сначала необходимо получить файл исходного кода с помощью Yacc:

```
yacc –d lab6.y
```

При запуске с опцией **–d** вместе с файлом исходного кода, имеющего стандартное название **y.tab.c**, создаётся также заголовочный файл y.tab.h, необходимый для успешной компиляции **lex.yy.c.** В современных дистрибутивах Linux часто вместо Yacc используется Bison – его усовершенствованный вариант. В этом случае, можно запускать его в режиме полной идентичности с Yacc, который включается опцией **–y**:

```
bison –d –y lab6.y
```

При успешном создании обоих файлов, компиляция всей программы-калькулятора может быть осуществлена командой:

```
gcc lex.yy.c y.tab.c –ll –lm –o lab6_1
```

так как необходимо подключать библиотеки libl (библиотека Lex) и libm (библиотека математических функций). Исполняемый файл получится с именем **lab6_1**.

Калькулятор обрабатывает поток символов со стандартного ввода, результат выводит на стандартный вывод.

Задание к лабораторной работе

1. Скомпилируйте программу калькулятора и убедитесь в её работоспособности.

2. Познакомьтесь с содержимым файла **`y.tab.h`**, созданным YACC, объясните, что и для чего там находится.

3. Измените программу калькулятора согласно заданному варианту, скомпилируйте и убедитесь в работоспособности модифицированного варианта.

Варианты:

1. (Простой) Добавьте в список доступных в калькуляторе функций другие функции одного аргумента (например, sin и cos).

2. (Средней сложности) Добавьте в список доступных в калькуляторе операций возведение в степень (знак операции выберите по своему усмотрению).

3. (Сложный) Добавьте в список доступных в калькуляторе функций функции двух аргументов (например, max и min).

Во время отчёта по лабораторной работе будет проверяться знание основ работы с YACC и LEX, правильность работы написанных программ и понимание их исходного кода.

СПИСОК ЛИТЕРАТУРЫ

1. Коньков, К.А. Основы операционных систем: учебник / К.А. Коньков, В.Е. Карпов. – Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. – 346 с. – ISBN 978-5-4497-0889-2.

2. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. – 4-е издание. Серия «Классика computer science». – СПб.: Питер, 2015. – 1120 с. – ISBN 978-5-496-01395-6.

3. Шоттс, У. Командная строка Linux: полное руководство / У. Шоттс. [пер. с англ.]. – Санкт-Петербург [и др.]: Питер, 2019. – 479 с.

4. Ахо, А. Компиляторы. Принципы, технологии, инструменты. / А. Ахо, Р. Сети, Дж. Ульман. – Издательство Диалектика, 2019. – 1175 с. – ISBN: 978-5-8459-1932-8.

5. Мейсон Энтони, Браун Дуг, Левин Джон Р. Lex & Yacc. – O'Reilly Media, 1995. – 387 с. – ISBN 156-5-920-007.