Часть 2 Изучаем Linux

Раздел 2.1

Лабораторная работа N 6

Программирование в ОС Linux. Основные этапы создания программ. Обработка исключительных ситуаций.

Цель работы: Целью работы является получение основных навыков создания программ на языке высокого уровня, обработки исключительных (ошибочных) ситуаций.

Теоретическое введение

Создание любой программы обычно начинается с базовой идеи, разработки ее блок-схемы и написания исходного текста. Далее следуют этапы компиляции и отладки.

Опустим процесс рождения базовой идеи и разработку блок-схем, полагая, что все это уже сделано. Итак, начнем с исходного текста будущей программы.

6.1 Исходный текст

Исходный текст программы, разработанной для LINUX с использованием языка высокого уровня Си, по большому счету мало отличаются от текстов приложений, создаваемых для других операционных систем. Можно сказать уверенно, что синтаксис языка определяется не операционной системой.

Возможности операционной системы доступны прикладному программисту в виде набора функций, называющегося *интерфейсом прикладного программирования* (Application Programming Interface, API). Отличия в разработке программ возникают как раз на уровне использования средств API, которые для каждой операционной системы разные.

В среде программирования UNIX системные вызовы определяются как функции Си, независимо от фактической реализации вызова функции ядра операционной системы. В UNIX используется подход, при котором каждый системный вызов имеет соответствующую функцию (или функции) с тем же именем, хранящуюся в стандартной библиотеке языка Си (в дальнейшем эти функции будем для простоты называть системными вызовами). Функции библиотеки выполняют необходимое преобразование аргументов и вызывают требуемую процедуру ядра, используя различные приемы. Заметим, что в этом случае библиотечный код выполняет только роль оболочки, в то время как фактические инструкции расположены в ядре операционной системы.

Помимо системных вызовов программисту предлагается большой набор функций общего назначения. Эти функции не являются точками входа в ядро операционной системы, хотя в процессе выполнения многие из них выполняют системные вызовы. Например, функция *printf* использует системный вызов *write* для записи данных в файл, в то время как функции *strcpy* (копирование строки) или atoi (преобразование символа в его числовое значение) вообще не прибегают к услугам операционной системы. Функции, о которых идет речь, хранятся в стандартных библиотеках Си и наряду с системными вызовами составляют основу среды программирования в UNIX.

Таким образом, часть библиотечных функций является "надстройкой" над системными вызовами, обеспечивающие более удобный способ получения системных услуг,

6.2 Заголовочные файлы

Использование системных функций обычно требует включения в текст программы *файлов заголовков,* содержащих определения функций - число передаваемых аргументов, типы аргументов и возвращаемого значения. Большинство системных файлов заголовков расположены в каталогах */usr/include* или */usr/include/sys*. Если вы планируете использовать малознакомую системную функцию, будет нелишним изучить соответствующий раздел электронного справочника **man.** Там же, помимо описания формата функции, возвращаемого значения и особых ситуаций, вы найдете указание, какие файлы заголовков следует включить в программу.

Файлы заголовков (см. Приложение А) включаются в программу с помощью директивы #include. При этом, если имя файла заключено в угловые скобки < >, это означает, что поиск файла будет производиться в общепринятых каталогах хранения файлов заголовков. Если же имя файла заголовка заключено в кавычки, то используется явно указанное абсолютное или относительное имя файла.

Внимание Нужно также заметить, что наряду со стандартными типами языка Си, например *char,* для параметров системных функций используются производные типы, имеющие окончание t, которые вы в большом количестве встретите при программировании в LINUX, получили название примитивов системных данных. Большинство этих типов определены в файле <sys/types.h>, а их назначение заключается в улучшении переносимости написанных программ. Вместо конкретных типов данных, каковыми являются int , *char* и т, п., приложению предлагается набор системных типов, гарантированно неизменных в контексте системных вызовов. Фактический размер переменных этого типа может быть разным для различных версий системы, но это отразится в изменении соответствующего файла заголовков и потребует только перекомпиляции вашей программы.

6.3 Компиляция

Процедура создания большинства приложений является общей и приведена на рис 1.

Первой фазой является стадия **компиляции,** когда файлы с исходными текстами программы, включая файлы заголовков, обрабатываются компилятором, например **gcc**. Параметры компиляции задаются либо с помощью специального файла *makefile* (или Makefile), либо явным указанием необходимых опций компилятора в командной строке. В итоге компилятор создает набор промежуточных объектных файлов. Традиционно имена созданных объектных файлов имеют суффикс ",o".

На следующей стадии эти файлы с помощью редактора связей ld(ельd) связываются друг с другом и с различными библиотеками, включая стандартную библиотеку по умолчанию, и библиотеки, указанные пользователем в качестве параметров. При этом редактор связей может выполняться в двух режимах: *статическом* и *динамическом,* что задается соответствующими опциями.

В **статическом,** наиболее традиционном режиме связываются все объектные модули и статические библиотеки (их имена имеют суффикс ".a"), производится разрешение всех внешних ссылок модулей и создается единый исполняемый файл, содержащий весь необходимый для выполнения код. Во втором случае, редактор связей по возможности подключает разделяемые библиотеки (имена этих библиотек имеют суффикс ".so"). В результате создается исполняемый файл, к которому в процессе запуска на выполнение будут подключены все разделяемые объекты. В обоих случаях по умолчанию создается исполняемый файл с именем *a.out.*

Для достаточно простых задач все фазы автоматически выполняются вызовом команды:

```
gcc  prog.c  -o prog.exe
```

которые создают исполняемый файл с именем prog.exe. В этом случае умалчиваемое имя исполняемого файла (a.out) изменено на prog.exe с помощью опции –o (буква o). Впрочем, указанные стадии можно выполнять и раздельно, с использованием команд **gcc** и **ld**. Заметим, что на самом деле команда **gcc** является программной оболочкой и компилятора и редактора связей, которую и рекомендуется использовать при создании программ.

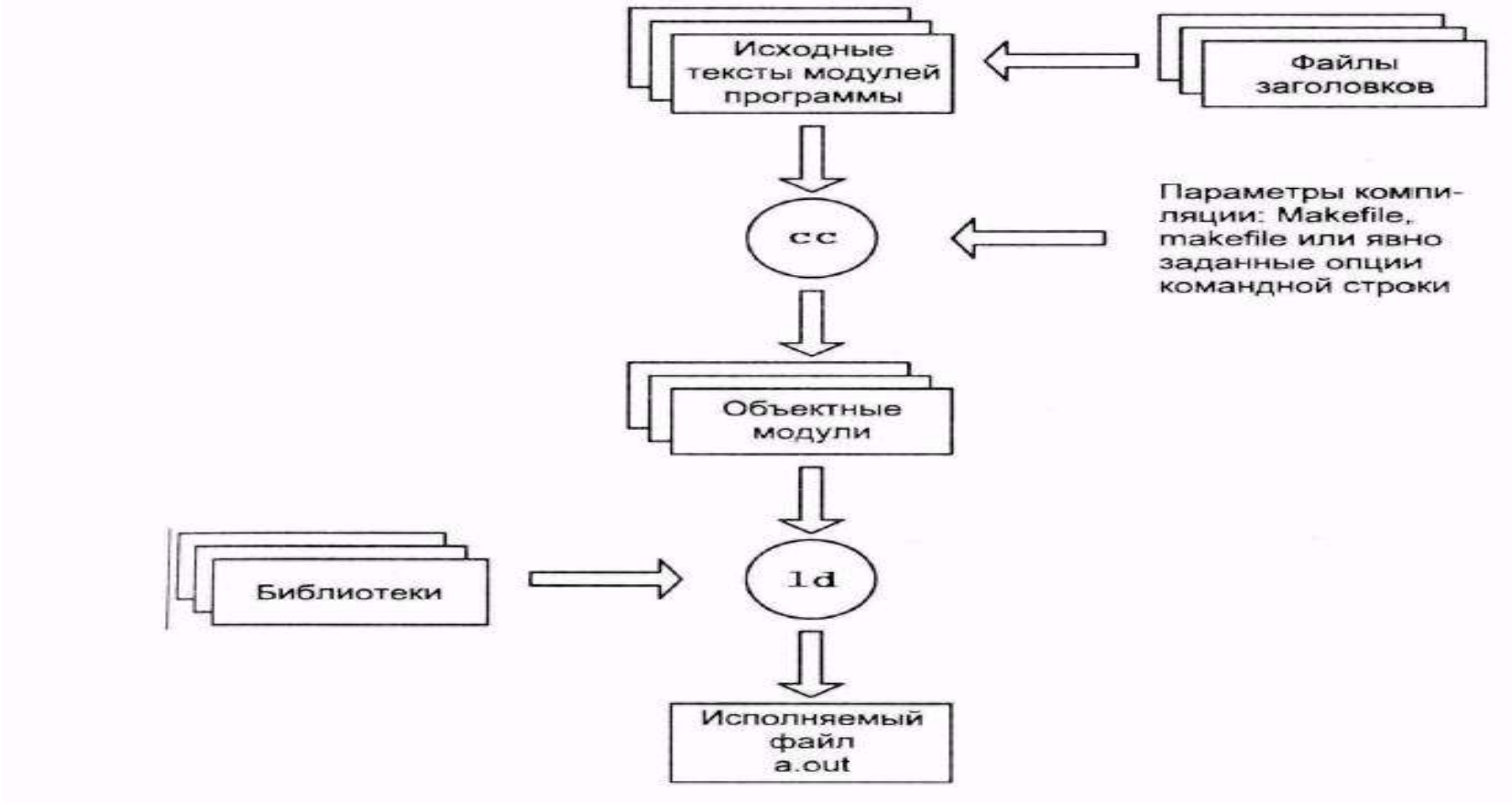

Рис. 1: Компиляция программ, написанных на языке Си

6.4 Выполнение Си-программы в операционной системе UNIX

Выполнение программы начинается с создания в памяти ее образа (процесса) и связанных с процессом структур ядра операционной системы, инициализации и передаче управления инструкциям программы. Завершение программы ведет к освобождению памяти и соответствующих структур ядра. Образ программы в памяти содержит, как минимум, сегменты инструкций и данных, созданные компилятором, а также стек для хранения автоматических переменных при выполнении программы.

Функция **main()** является первой функцией, определенной пользователем (т. е. явно описанной в исходном тексте программы), которой будет передано управление после создания соответствующего окружения запускаемой на выполнение программы. Традиционно функция **main()** определяется следующим образом:

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

Первый аргумент **(argc**) определяет число параметров, переданных программе, включая ее имя. Указатели на каждый из параметров передаются в массиве **argv[],** таким образом, через **argv[0]** адресуется строка, содержащая имя программы, **argv[1]** указывает на первый параметр и т. д. до **argv [argc-1].**

Массив **envp[]** содержит указатели на переменные окружения, передаваемые программе. Каждая переменная представляет собой строку вида

имя_переменной = значение_переменной

6.5 Завершение С-программы

Существует несколько способов завершения программы. Основными являются возврат из функции **main()** и вызов функций **exit,** оба приводят к завершению выполнения задачи. Заметим, что процесс может завершиться по не зависящим от него обстоятельствам, например, при получении сигнала.

Системный вызов exit выглядит следующим образом:

```
void exit   (int status);
```

Аргумент *status,* передаваемый функции **exit,** возвращается родительскому процессу и представляет собой код возврата программы. По соглашению программа возвращает 0 в случае успеха и другую величину в случае неудачи. Значение кода

неудачи может иметь дополнительную трактовку, определяемую самой программой. Наличие кода возврата позволяет программам взаимодействовать друг с другом (например для организации условного запуска программ).

Помимо передачи кода возврата, функция **exit** производит ряд действий, в частности выводит буферизированные данные и закрывает потоки ввода/вывода. Альтернативой ей является функция **_exit,** которая не производит вызовов библиотеки ввода/вывода, а сразу вызывает системную функцию завершения ядра,

Задача может зарегистрировать *обработчики выхода* (exit handler) - функции, которые вызываются после вызова **exit,** но до окончательного завершения процесса. Эти обработчики, вызываемые по принципу LIFO (последний зарегистрированный обработчик будет вызван первым), запускаются только при "добровольном" завершении процесса. Например, при получении процессом сигнала обработчики выхода вызываться не будут Для обработки таких ситуаций следует использовать специальные функции - обработчики сигналов.

Обработчики выхода регистрируются с помощью функции **atexit:**

```
int atexit(void   (*func)(void));
```

Функцией atexit может быть зарегистрировано до 32 обработчиков.

На рис. 2 проиллюстрированы возможные варианты запуска и завершения программы написанной на языке Си,

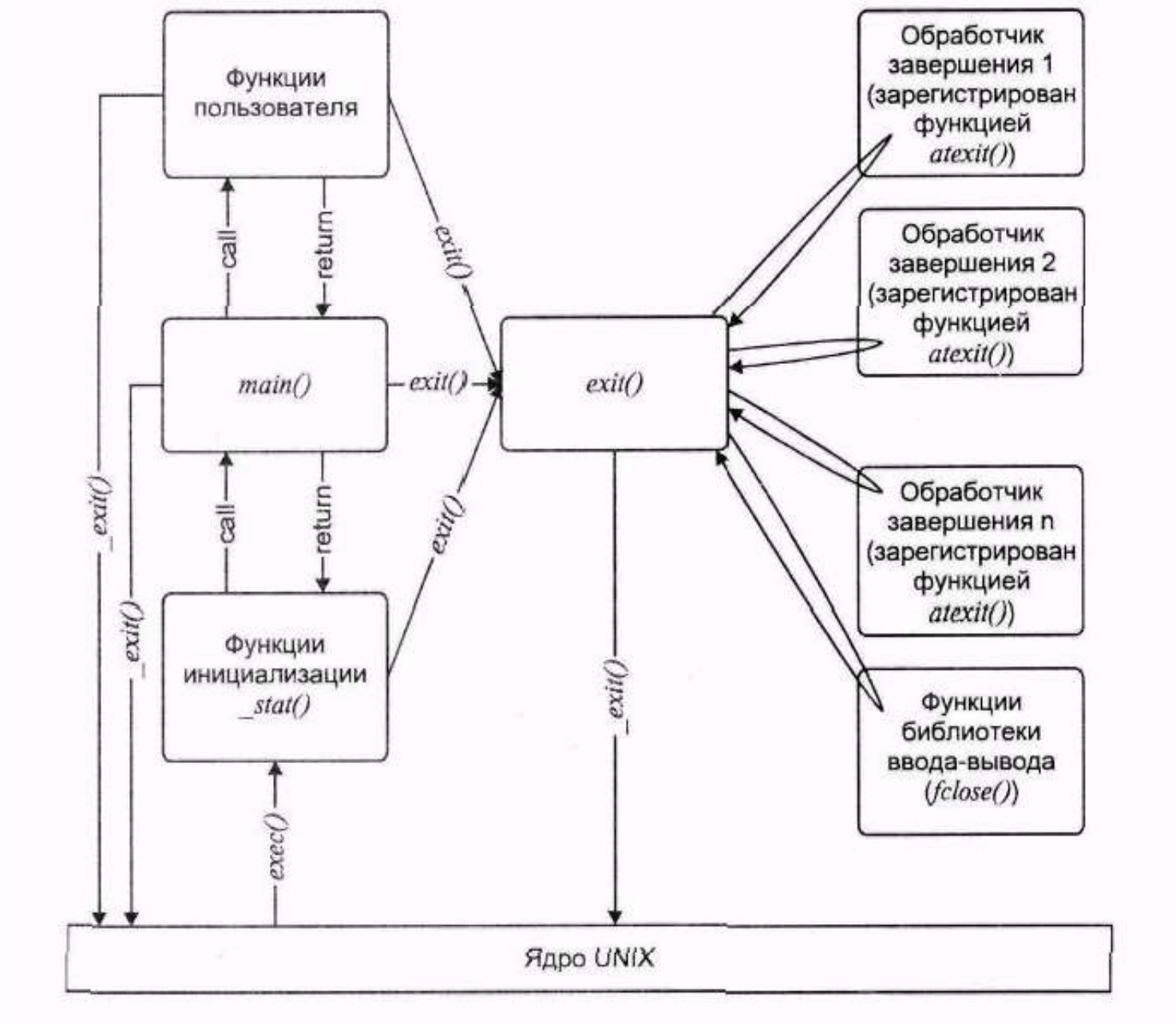

Рис. 2: Механизмы завершения программ

Пример программы, регистрирующей обработчики выхода:

```
#include stdio.h>

#include stdlib.h>

     void myexitl(void)
       { printf   ("Это завершение1.\n"); }

      void myexit2(void)
        {   printf   ("Это завершение2.\n"); }


int main  (int argc,   char ** argv[])
{
    printf ("Программа запущена.\n");
     atexit (myexitl);
    atexit (myexit2);
     exit (10); }
```

6.6 Обработка ошибок

Обычно в случае возникновения ошибки системные вызовы возвращают -1 *и* устанавливают значение переменной **errno,** указывающее причину возникновения ошибки. Так, например, существует более десятка причин завершения вызова **open** с ошибкой, и все они могут быть определены с помощью переменной errno, Файл заголовков <errno.h> содержит коды ошибок, значения которых может принимать переменная errno, с краткими комментариями. Библиотечные функции, как правило, не устанавливают значение переменной errno, а код возврата различен для разных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к электронному справочнику man.

Поскольку базовым способом получения услуг ядра являются системные вызовы, рассмотрим более подробно обработку ошибок в этом случае.

Переменная errno определена следующим образом:

```
external int errno;
```

Следует обратить внимание, что значение errno не обнуляется следующим нормально завершившимся системным вызовом. Таким образом, значение errno имеет смысл только после системного вызова, который завершился с ошибкой.

Стандарт ANSI C определяет две функции, помогающие сообщить причину ошибочной ситуации: strerror и perror.

Функция strerror имеет вид:

```
char *strerror(int errnum);
```

Функция принимает в качестве аргумента errnum номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации.

Функция perror объявлена следующим образом:

```
void perror(const char *s);
```

Функция выводит **в стандартный поток сообщений об ошибках** информацию об ошибочной ситуации, основываясь на значении переменной errno. Строка s, передаваемая функции, предваряет это сообщение и может служить дополнительной информацией, например содержать название функции или программы, в которой произошла ошибка.

Следующий пример иллюстрирует использование этих двух функций:

```
#include <errno.h>

 #include <stdio.h>

int main  (int argc,   char *argv[])
{
  fprintf(stderr,   "ENOMEM:   %s\n",   strerror(ENOMEM));
  errno = ENDEXEC;
  perror (argv [0] )   ;
}
```

.

В приложении В приведены наиболее общие ошибки системных вызовов, включая сообщения, которые обычно выводят функции strerror и реггог, а также их краткое описание.

Содержание отчета

1.Номер и тема лабораторной работы.
2.Вариант задания.
3.Текст программы создания бинарного файла.
4. Текст программы обработки бинарного файла.
5. Итоги работы программ.

Задание к лабораторной работе

В соответствии с вариантом задания разработать две программы: программу создания и программу обработки бинарного файла. Также необходимо обработать ошибки, которые могут возникнуть при открытии, закрытии и выводе файла на экран.

При завершении программы (даже в случае возникновения ошибок) на экран должно быть выдано сообщение об авторе программы, учетное имя, группа.

Варианты заданий:

Таблица 1

Вариант	Условие задачи
1.	1.Создать файл Train.dat, содержащий 8 записей следующей структуры: название пункта назначения; номер поезда; время отправления. 2.Написать программу, выполняющую следующую обработку файла Train.dat: • поиск в файле и вывод на экран информации о поезде, номер которого введен с клавиатуры; • если таких поездов нет, выдать соответствующее сообщение на дисплей.
2	1.Создать файл Spravka.dat, содержащий 10 записей следующей структуры: название начального пункта маршрута; название конечного пункта маршрута; номер маршрута; 2.Написать программу, выполняющую следующую обработку файла Spravka.dat: • поиск в файле данных о маршруте, номер которого вводится с клавиатуры; если таких маршрутов нет, выдать соответствующее сообщение на дисплей.
3	1.Создать файл Train.dat, содержащий 8 записей следующей структуры: название пункта назначения; номер поезда; время отправления. 2.Написать программу, выполняющую следующую обработку файла Train.dat: • поиск в файле поездов, отправляющихся после введенного с клавиатуры времени; • если таких поездов нет, выдать соответствующее сообщение на дисплей.
4.	1.Создать файл Spravka.dat, содержащий 10 записей следующей структуры: название начального пункта маршрута; название конечного пункта маршрута; номер маршрута; 2.Написать программу, выполняющую следующую обработку файла Spravka.dat: • поиск в файле данных о маршрутах, которые начинаются или заканчиваются в пункте, название которого вводится с клавиатуры; • если таких маршрутов нет, выдать соответствующее сообщение на дисплей.
5	1.Создать файл Wedomost.dat, содержащий 10 записей следующей структуры: фамилия и инициалы студента; номер группы; успеваемость по трем предметам; 2.Написать программу, выполняющую следующую обработку файла Wedomost.dat: • поиск в файле информации о студентах, имеющих хотя бы одну оценку 2; • если таких студентов нет, выдать соответствующее сообщение на дисплей.

6	1.Создать файл Spravka.dat, содержащий 10 записей следующей структуры: название пункта назначения; номер рейса; тип самолета. 2.Написать программу, выполняющую следующую обработку файла Spravka.dat: • поиск в файле номеров рейсов, вылетающих в пункт, название которого вводится с клавиатуры; • если таких рейсов нет, выдать соответствующее сообщение на дисплей.
7	1. Создать файл Wedomost.dat, содержащий 8 записей следующей структуры: фамилия и инициалы студента; номер группы; успеваемость по трем предметам; 2. Написать программу, выполняющую следующую обработку файла Wedomost.dat: • поиск в файле информации о студентах, имеющих только оценки 4 и 5; • если таких студентов нет, выдать соответствующее сообщение на дисплей.
8	1. Создать файл Wedomost.dat, содержащий 6 записей следующей структуры: фамилия и инициалы студента; номер группы; успеваемость по трем предметам; 2. Написать программу, выполняющую следующую обработку файла Wedomost.dat: • поиск в файле информации о студентах, имеющих средний балл меньше 4; • если таких студентов нет, выдать соответствующее сообщение на дисплей.
9	1.Создать файл Spravka.dat, содержащий 10 записей следующей структуры: название пункта назначения; номер рейса; тип самолета. 2.Написать программу, выполняющую следующую обработку файла Spravka.dat: • поиск в файле номеров рейсов, обслуживаемых самолетом, тип которого вводится с клавиатуры; • если таких рейсов нет, выдать соответствующее сообщение на дисплей.
10	1. Создать файл Train.dat, содержащий записи следующей структуры: название пункта назначения; номер поезда; время отправления; 2. Написать программу, выполняющую следующую обработку файла Train.dat: • поиск в файле поездов, отправляющихся в пункт, название которого вводится с клавиатуры; • если таких поездов нет, выдать соответствующее сообщение на дисплей.
11	1.Создать файл WORKER.dat, содержащий 6 записей следующей структуры: фамилия и инициалы; номер телефона; день рождения (массив из трех чисел); 2. Написать программу, выполняющую следующую обработку файла WORKER.dat: • поиск в файле информации о человеке, чья фамилия введена с клавиатуры; • если такого человека нет, выдать соответствующее сообщение на дисплей.
12	1. Создать файл STUDENT.dat , содержащий записи следующей структуры: ФИО студента; его экзаменационные оценки по трем дисциплинам. 2. Написать программу, которая выбирает из файла студентов, имеющих хотя бы одну задолженности. Вывести ФИО этих студентов и количество несданных экзаменов.
13	1. Создать файл F1.dat, содержащий 8 записей следующей структуры: ФИО; номер телефона; день рождения (массив из трех чисел) 2. Написать программу, которая переписывает файл F1.dat в файл F2.dat таким образом, чтобы записи расположились а алфавитном порядке.

14	1. Создать файл ZNAK.dat, содержащий записи следующей структуры: фамилия и инициалы; знак Зодиака; день рождения (массив из трех чисел); 2. Написать программу, выполняющую следующие действия: • поиск в файле ZNAK.dat информации о людях, родившихся под знаком, название которого введено с клавиатуры; • если таких людей нет, выдать соответствующее сообщение на дисплей.
15	1. Создать файл Work.dat, содержащий 6 записей следующей структуры: ФИО рабочих; их среднемесячный заработок. 2. Написать программу, выполняющую следующую обработку файла Work.dat: Вывести ФИО рабочих, имеющих наибольший заработок.
16	1. Создать файл WORKER.dat, содержащий 5 записей следующей структуры: фамилия и инициалы; номер телефона; день рождения (массив из трех чисел); 2. Написать программу, выполняющую следующую обработку файла WORKER.dat: • поиск в файле информации о людях, родившихся в месяц, номер которого введен с клавиатуры; • если таких людей нет, выдать соответствующее сообщение на дисплей.
17	1. Создать файл ABONENT.dat, содержащий 5 записей следующей структуры: ФИО абонента; его номер телефонов. 2. Составить программу, которая по ФИО абонента, введенной с клавиатуры, выводит его номер телефона. • если такого абонента нет, выдать соответствующее сообщение на дисплей.
18	1. Создать файл AEROFLOT.dat, содержащий 6 записей следующей структуры: номер рейса самолета; количество свободных мест. 2. Составить программу, которая по вводимому с клавиатуры номеру рейса выводит из файла сведения о количестве свободных мест на этот рейс. • если такого рейса нет, выдать соответствующее сообщение на дисплей.
19	1. Создать файл PRICE.dat, содержащий 5 записей следующей структуры: название детской игрушки; ее стоимость. 2. Написать программу, выполняющую следующую обработку файла PRICE.dat: Вывести наименование и стоимость самой дорогой игрушки.
20	1. Создать файл PRICE.dat, содержащий 5 записей следующей структуры: название детской игрушки, ее стоимость, название магазина, в котором она продается. 2. Написать программу, выполняющую следующую обработку файла PRICE.dat: Вывести полную информацию об игрушке, название которой введено с клавиатуры Если игрушки такой нет в PRICE, выдать сообщение на дисплей.
21	1. Создать файл PriceList.dat, содержащий записи следующей структуры: название товара; название магазина, в котором он продается; стоимость; 2. Написать программу, выполняющую следующую обработку файла PriceList.dat: • поиск в файле информации о товаре, название которого введено с клавиатуры; • если такого товара нет, выдать соответствующее сообщение на дисплей.

22	1. Создать файл PriceList.dat, содержащий записи следующей структуры: название товара; название магазина, в котором он продается; стоимость; 2. Написать программу, выполняющую следующую обработку файла PriceList.dat: • поиск в файле информации о товарах, имеющихся в магазине, название которого введено с клавиатуры; • если такого магазина нет, выдать соответствующее сообщение на дисплей.
23	1. Создать файл ORDER.dat, содержащий записи следующей структуры: расчетный счет плательщика; расчетный счет получателя; перечисляемая сумма. 2. Написать программу, выполняющую следующую обработку файла ORDER.dat: • поиск в файле информации о сумме и расчетном счете получателя, которому перечислил эту сумму плательщик, номер которого введен с клавиатуры; • если такого плательщика нет, выдать соответствующее сообщение на дисплей.
24	1. Создать файл ZNAK.dat, содержащий записи следующей структуры: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел); 2. Написать программу, выполняющую следующую обработку файла ZNAK.dat: • поиск в файле информации о людях, родившихся в одно и тоже число, значение которого введено с клавиатуры; • если таких людей нет, выдать соответствующее сообщение на дисплей.
25.	1. Создать файл F1.dat, содержащий 8 записей следующей структуры: ФИО; номер телефона; день рождения (массив из трех чисел) 2. Написать программу, которая переписывает файл F1.dat в файл F2.dat таким образом, чтобы записи расположились в порядке убывания дат рождений.

4 Контрольные вопросы

1. Этапы создания программы с использованием языка высокого уровня.
2. Заголовочные файлы. Зачем используются, где находятся?
3. В чем отличие использования символов < > и " " в директиве препроцессора #include
4. Зачем используются производные системные типы (имеющие окончание t)?
5. Этапы компиляции программ, написанных на языке Си.
6. Зачем нужны файлы, имеющие суффикс ".o"?
7. Зачем нужна программа библиотекарь ld?
8. Способы завершения программ. Команды exit, _exit.
9. Обработчики завершения программ.
10. Функции обработки ошибок в программах.

A Стандартные заголовки Си библиотек

Таблица 2: Стандартные файлы заголовков

Файл заголовка	Назначен
<assert.h>	Содержит прототип функции assert, используемой для диагностики
<cpio-h>	Содержит определения, используемые для файловых архивов cpio
<ctype.h>	Содержит определения символьных типов, а также прототипы функций определения классов символов (ASCII, печатные, цифровые и т. д.) - isascii, isprint, isdigit и т.д.
<dirent.h>	Содержит определения структур данных каталога, а также прототипы функций работы с каталогами opendir, readdir и т. д.
<errno.h>	Содержит определения кодов ошибок (см. раздел "Обработка ошибок")
<fcntl.h>	Содержит прототипы системных вызовов fcntl, open и creat, а также определения констант и структур данных, необходимых при работе с файлами
<float.h>	Содержит определения констант, необходимых для операций с плавающей точкой
<ftw.h>	Содержит прототипы функций, используемых для сканирования дерева файловой системы (file tree walk) ftw и nftw, а также определения используемых констант
<grp.h>	Содержит прототипы функций и определения структур данных, используемых для работы с группами пользователей: getgrnam, getgreiit, gctgrgid и т. д.
<langinfo.h>	Содержит определения языковых констант: дни недели, названия месяцев и т. д., а также прототип функции langinfo
<limits.h>	Содержит определения констант, определяющих значения ограничений для данной реализации: минимальные и максимальные значения основных типов данных, максимальное значение файловых связей, максимальная длина имени файла и т. д.
<localc.h>	Содержит определения констант, используемых для создания пользовательской среды, зависящей от языковых и культурных традиций (форматы дат, денежные форматы и т. д.), а также прототип функции setlocale
<math.h>	Содержит определения математических констант
<nltypes.h>	Содержит определения для каталогов сообщений (message catalog), а также прототипы функций catopen и catclose
<pwd.h>	Содержит определение структуры файла паролей /etc/passwd, а также прототипы функций работы с ним: getpwnam, getpwent, getpwuid и т. д.
<regex.h>	Содержит определения констант и структур данных, используемых в регулярных выражениях, а также прототипы функций для работы с ними: regexcc и т, д.

Файл заголовка	Назначен
<search/h>	Содержит определения констант и структур данных, а также прототипы функций, необходимых для поиска: hsearch, hcreate, hdestroy
<setjmp/h>	Содержит прототипы функций перехода setjmp, sigsetjmp, longjmp, siglongjmp, а также определения связанных с ними структур данных
<signal.h>	Содержит определения констант и прототипы функций, необходимых для работы с сигналами: sigsctops, sigempfyset, sigaddsef и т. д.
<stdarg.n>	Содержит определения, необходимые для поддержки списков аргументов переменной длины
<stddcf.h>	Содержит стандартные определения (например sizet)
<stdio.h>	Содержит определения стандартной библиотеки ввода/вывода
<stdlib,h>	Содержит определения стандартной библиотеки
<string.h>	Содержит прототипы функций работы со строками siring, strcasecmp, strcat, strcpy и т. д.
<tar.h >	Содержит определения, используемые для файловых архивов tar.
<termios.h>	Содержит определения констант, структур данных и прототипы функций для обработки терминального ввода/вывода
<time.h>	Содержит определения типов, констант и прототипы функций для работы со временем и датами: time, ctime, localtime, tzset, а также определения, относящиеся к таймерам gefitimer, setitimer.
<ulimit. h>	Содержит определения констант и прототип системного вызова ulimit для управления ограничениями, накладываемыми на процесс.
<unistd.h>	Содержит определения системных символьных констант, а также прототипы большинства системных вызовов
<utime.h>	Содержит определения структур данных и прототип системного вызова iitime для работы с временными характеристиками файла (временем доступа и модификации)
<sys/ipc.h>	Содержит определения, относящиеся к системе межпроцессного взаимодействия (IPC),
<sys/msg.h>	Содержит определения, относящиеся к (сообщениям) подсистеме IPC.
<sys/resource.h>	Содержит определения констант и прототипы системных вызовов управления размерами ресурсов, доступных процессу: getrlimit и sctrlimit,
<sys/scm.h>	Содержит определения, относящиеся к семафорам подсистеме IPC.
<sys/shm.h>	Содержит определения, относящиеся к (разделяемой памяти) подсистеме 1PC.
<svs/stat.h>	Содержит определения структур данных и прототипы системных вызовов, необходимых для получения информации о файле: stat, lstat, fstat,

Файл заголовка	Назначен
<sys/times. h>	Содержит определения структур данных и прототипа сиетемного вызова times, служащего для получения статистики выполнения процесса (времени выполнения в режиме ядра, задачи и т. д.)
<sys/typcs,h>	Содержит определения примитивов системных данных
<sys/utsname.h>	Содержит определения структур данных и прототип системного вызова uname, используемого для получения имен системы (компьютера, операционной системы, версии и т.д.)
<sys/wait,h>	Содержит определения констант и прототипы системных вызовов wait, waifpid, используемых для синхронизации выполнения родственных процессов

Раздел 2.2 ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ И НИТЕЙ

Как было сказано ранее, идея многопроцессных (многопрограммных) операционных систем заключается в том, что на одной ЭВМ (псевдо)одновременно исполняется несколько программ (процессов)[1]. При этом считается, что процессы: 1) независимы и 2) ничего не знают о существовании друг друга[2]. Каждому процессу выделяется свой набор ресурсов, которыми он монопольно владеет, и никто другой (кроме операционной системы) не имеет прямого доступа к ним.

Каждый процесс, в свою очередь, может выполнять несколько нитей (потоков инструкций). При этом все нити одного процесса также выполняются независимо друг от друга и обладают некоторым набором «собственных» ресурсов, но без труда могут получить доступ к общей памяти процесса.

Очевидно, что нити изначально предназначены для взаимодействия друг с другом. А могут ли возникнуть ситуации, когда одному процессу необходимо «сообщить» другому (или нескольким другим) процессу(-ам) какую-либо информацию? Ответ очевиден – да. С такими взаимодействиями мы уже сталкивались, когда изучали конвейерный способ запуска программ. Кроме этого, процессу, например, может потребоваться:

- сообщить о своей готовности обработать данные;
- «поделиться» с другим процессом объемом выполняемой работы или взять часть работы другого процесса на себя;
- обеспечить строгую последовательность работы нескольких процессов (например, чтобы начисление процентов на счете в банке производилось только после его пополнения);
- и т.п.

К средствам межпроцессного взаимодействия (англ. Interprocess Communications, IPC), реализованных в операционной системе Linux, относятся:

- ожидание завершения родственного процесса;
- сигналы;
- каналы (именованные и неименованные);
- очереди сообщений;
- семафоры;
- разделяемая память;
- сокеты.

[1] Очевидно, что в случае многопроцессорной (многоядерной) ЭВМ процессы на самом деле выполняются параллельно.

[2] Вспомним, что каждый процесс «знает» лишь номер своего родительского процесса (поле PPID дескриптора).

Последний способ IPC позволяет реализовать взаимодействие между процессами, выполняющимися на разных ЭВМ, соединённых каналами передачи данных.

Все средства IPC условно можно разделить по типу процессов, участвующих во взаимодействии:

- «любой процесс». Ресурсы IPC доступные всем процессам (согласно правам доступа). Такие ресурсы создаются и управляются самими процессами. К этому типу IPC относятся: именованные каналы, очереди сообщений, семафоры, разделяемая память;
- «родственные процессы». Такие ресурсы создаются и управляются операционной системой. К этому типу ресурсов относятся: сигналы, неименованные каналы, сокеты.

Кроме того, процессы могут получать информацию о том, каким образом они были запущены и в какой среде. С этого типа взаимодействия и начнём.

Тема 7 Взаимодействие программ с командной оболочкой и средой окружения

7.1. Взаимодействие программ с командной оболочкой (строкой запуска)

В программе, написанной на языке Си, первой выполняемой функцией является функция **main**. Традиционно функция *main* определяется следующим образом[3]:

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

Из заголовка функции видно, что она имеет два параметра и возвращает целое значение. Первым параметром (**argc**) в функцию передаётся число аргументов, указанных в командной строке при запуске программы, включая её имя. Указатели на каждый из аргументов командной строки передаются в массиве **argv[].** Таким образом, *argv*[0] указывает на строку, содержащую имя программы, *argv*[1] – на первый аргумент и т.д. до *argv* [*argc* – 1]. Элемент *argv* [*argc*] всегда содержит NULL.

Возвращаемое значение функции *main* является статусом завершения программы (процесса).

Приведём пример программы, которая выводит значения всех аргументов, переданных функции *main* (листинг 1).

[3] Некоторые компиляторы с языка Си допускают и другие определения функции *main*. Например, Borland TurboC допускает определение типа возвращаемого значения как *void*. В gcc можно разрешить игнорирование аргументов функции *main*.

Листинг 1. Вывод информации о командной строке

```
1. #include <stdio.h>
2. int main(int argc, char *argv[]){
3.   int i;
4.   printf ("Имя программы - %s\n", argv[0]);
5.   printf ("Число параметров - %d\n", argc-1);
6.   for (i = 1; i < argc; i++)
7.     printf("argv[%d] = %s\n", i, argv[i]);
8.   return (0);
9. }
```

Чтобы определить, были ли указаны в командной строке опции, аргументы командной строки могут быть обработаны при помощи следующих функций:

```
#include <unistd.h>

extern char *optarg;
extern int optind, opterr, optopt;

int getopt(int argc, char *argv[], char
*optstring);

#include <getopt.h>

int getopt_long(int argc, char *argv[],
                char *optstring,
                struct option *longopts,
                int *longindex);

int getopt_long_only(int argc, char *argv[],
                     char *optstring,
                     struct option *longopts,
                     int *longindex);
```

Напомним, что *опцией* называется аргумент командной строки, начинающийся с символов «-»(тире) или «--» (двойное тире). Причём, если используется один знак тире, то опция считается односимвольной или ***короткой***, если же два знака тире, то многосимвольной или ***длинной.***

Функция **getopt** просматривает аргументы командной строки по очереди и определяет, есть ли в них заданные короткие опции. Поиск только длинных опций осуществляется функцией **getopt_long_only**. Функция **getopt_long** осуществляет поиск как коротких, так и длинных опций.

В качестве аргументов функциям передаются:

- *argc* – количество аргументов командной строки (размер массива *argv*);
- *argv* – массив указателей на строки-аргументы командной строки;
- *optstring* – указатель на строку, содержащую символы коротких опций;
- *longopts* – массив структур *option*, описывающих длинные опции. Последний элемент массива должен быть заполнен нулями;
- *longindex* – указатель на целую переменную, в которую, в случае обнаружения длинной опции, будет помещён номер соответствующего элемента массива *longopts*.

Обратите внимание, что функции *getopt*, *getopt_long* и *getopt_log_only* на самом деле ничего не знают о командной строке, а всего лишь просматривают массив строк на предмет нахождения в нём заданных элементов.

Структура **option** определена в заголовочном файле *getopt.h* следующим образом:

```
struct option {
  char *name;
  int has_arg;
  int *flag;
  int val;
};
```

Значения полей структуры *option* приведены в табл. 7.1.

Таблица 1. Значения полей структуры *option.*

Поле	Значение
name	Является именем длинной опции.
has_arg	Может принимать значения: *no_argument* (или 0), если опция не имеет аргумента; *required_argument* (или 1), если опция требует обязательного указания аргумента; *optional_argument* (или 2), если опция может иметь необязательный аргумент.
val	Значение, которое должно быть помещено по адресу, указанному в аргументе *flag*, в случае если будет обнаружена данная длинная опция.
flag	Указывает адрес, куда требуется поместить значение *val*. Если *flag* равен NULL, то *val* возвращается как значение функции *getopt_long* (или *getopt_long_only*).

Глобальные переменные **optind**, **optarg**, **opterr** и **optopt** содержат соответственно номер аргумента командной строки, который должен быть проверен на наличие опции, указатель на аргумент найденной опции,

наличие ошибки, возникшей в ходе выполнения функции поиска опций, и дополнительные параметры, конкретизирующие работу функций. Подробнее об этих переменных будет сказано далее.

Поиск коротких опций функцией *getopt* производится так же, как командой *getopts, используемой в скриптах* (см. часть 1 курса). Отличия следующие.

1. Если просмотрены все параметры командной строки, то *getopt* возвращает –1.

2. Если в *optstring* после некоторого символа следует два двоеточия, то это означает, что опция имеет **необязательный** аргумент. Это является дополнением GNU.

3. Если *optstring* содержит символ «W», за которым следует точка с запятой («;»), то опция *-W foo* рассматривается как длинная опция *--foo*. Опция *-W* зарезервирована POSIX.2 для реализации расширений. Такое поведение является дополнительным для GNU и недоступным в библиотеках GNU libc более ранних по сравнению со второй версий.

По умолчанию *getopt* переставляет элементы содержимого *argv* в процессе поиска так, что, в конечном счёте, все аргументы, не являющиеся опциями, оказываются в конце. Возможны два других режима работы функции.

4. Если первым символом *optstring* является «+» или задана переменная окружения **POSIXLY_CORRECT**, то обработка опций прерывается на первом аргументе, не являющемся опцией.

5. Если первым символом *optstring* является «-»(тире), то каждый элемент *argv*, не являющийся опцией, обрабатывается так, как если бы он был аргументом опции с символом, имеющим код 1 (один). Такой режим работы используется программами, которые требуют опции и другие аргументы командной строки.

6. Специальный аргумент «--» (два тире) служит для обозначения конца опций независимо от режима работы функции *getopt*.

Если *getopt* не распознал символ опции, то он выводит в стандартный поток ошибок соответствующее сообщение, заносит символ в переменную *optopt* и возвращает «?». Вызывающая программа может предотвратить вывод сообщения об ошибке, установив нулевое значение **opterr**.

Если найден параметр, начинающийся с двух символов «-» (тире), то считается, что **найдена длинная опция**. В этом случае производится сравнение найденного аргумента (исключая двойное тире) со значениями *name* массива *longopts*.

Если совпадение найдено, то значение *val* помещается либо по адресу, указанному в *flag*, и *getopt_long* завершается с результатом 0, либо *getopt_long* завершается с результатом *val*. В любом из двух случаев, если параметр *longindex* не равен *NULL*, то по указанному в нём адресу помещается номер элемента массива, с которым совпала найденная длинная опция.

Длинная опция может иметь параметр, указываемый через знак равенства (*--опция=параметр*) или через пробел (*--опция параметр*). Наличие параметра указывается в поле *has_arg*.

Функция *getopt_long_only* работает так же, как *getopt_long*, но обрабатывает только длинные опции. В качестве указателя опции могут служить не только символы «--», но и «-». Если опция, начинающаяся с «-» (не с «--»), не совпадает с длинной опцией, но совпадает с короткой, то она обрабатывается как короткая опция.

Пример программы, анализирующей опции командной строки, представлен в листинге 2.

Листинг 2. Работа с переменными среды окружения

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

int main (int argc, char **argv){
  int c;
  int digit_optind = 0;
  int this_option_optind;
  int option_index = 0;
  static struct option long_options[] = {
       {“add”, 1, 0, 0},
       {“append”, 0, 0, 0},
       {“delete”, 1, 0, 0},
       {“verbose”, 0, 0, 0},
       {“create”, 1, 0, ‘c’},
       {“file”, 1, 0, 0},
       {0, 0, 0, 0}};
  while (1){
    this_option_optind = optind ? optind : 1;
    c = getopt_long (argc, argv, “abc:d:012”,
                     long_options, &option_index);
    if (c == -1) break;
    switch (c){
      case 0: printf (“параметр %s”,
                    long_options[option_index].name);
if (optarg) printf (“ с аргументом %s”, optarg);

              printf (“\n”);
              break;
      case ‘0’:
      case ‘1’:
      case ‘2’: if (digit_optind != 0 &&
 digit_optind != this_option_optind)
```

Листинг 2. Работа с переменными среды окружения

```
34.              printf ("цифры используются дважды.\n");
35.                  digit_optind = this_option_optind;
36.                  printf ("параметр  %c\n", c);
37.                  break;
38.        case 'a': printf ("параметр a\n");
39.                  break;
40.        case 'b': printf ("параметр b\n");
41.                  break;
42. case 'c': printf ("c со значением '%s'\n", optarg);
43.
44.                  break;
45.  case 'd': printf ("d со значением '%s'\n", optarg);
46.
47.                  break;
48.        case '?': break;
49.        default: printf ("результат 0%o ??\n", c);
50.      }
51.    }
52.    if (optind < argc){
53.      printf ("элементы ARGV, не параметры: ");
54.      while (optind < argc)
55.        printf ("%s ", argv[optind++]);
56.        printf ("\n");
57.    }
58.    exit (0);
59. }
```

7. 1. Взаимодействие Си-программы со средой окружения

Массив **envp[]**, передаваемый третьим аргументом в функцию *main*, содержит указатели на переменные среды окружения. Каждый элемент массива является указателем на строку вида: «имя_переменной = значение». Последним элементом массива является указатель NULL.

Стандарт ANSI C определяет только два первых параметра функции *main* – *argc* и *argv*. Стандарт POSIX.1 определяет также параметр *envp*, хотя рекомендует передачу окружения программы производить через глобальную переменную **environ**:

```
extern char ** environ;
```

Формат массива *environ* такой же, как и у *env*. Пример вывода переменных среды окружения с использованием двух массивов приведён в листинге 3.

Листинг 3. Вывод информации о командной строке

```
1. #include <stddef.h>
2. #include <stdio.h>
3. extern char **environ;
4. int main(int argc, char *argv[]){
5.   int i;
6.   printf ("Запущена программа %s. ", argv[0]);
7.   printf ("Число параметров %d\n", argc-1);
8.   for (i = 1; i < argc; i ++)
9.     printf("argv[%d] = %s\n", i, argv[i]);
10.  for (i = 0; i < 8; i ++)
11.    if (environ[i] != NULL)
12.      printf("environ[%d] : %s\n",i, environ[i]);
13.}
```

Для доступа к переменным окружения по их именам используются функции: **getenv** и **putenv**:

```
#include <stdlib.h>

char *getenv(const char *name);
int putenv (const char *string);
```

Функция *getenv* возвращает значение переменной окружения с именем *name*, а *putenv* помещает переменную и её значение в окружение процесса. Параметр *string* функции *putenv* содержит строку вида *var_name = var_value*.

В качестве примера приведём программу, похожую по своей функциональности на предыдущую, выводящую выборочно значения переменных и устанавливающую новые значения по желанию пользователя.

Листинг 4. Работа с переменными среды окружения

```
1.  #include <stddef.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.
5.  int main(int argc, char *argv[]){
6.    char *term;
7.    char buf[200], var[200];
8.
9.    /* Проверим, определена ли переменная TERM */
10.   if ((term = getenv("TERM")) == NULL){
11.     /* Если переменная не определена, получим от *
12.      * пользователя её значение и поместим       *
13.      * переменную в окружение программы          */
14.     printf ("Переменная TERM не определена.\n");
```

Листинг 4. Работа с переменными среды окружения

```
15.     printf ("Введите её значение: ");
16.     gets(buf);
17.     sprintf(var, "TERM=%s",buf);
18.     putenv(var);
19.     printf("%s\n", var);
20.   }
21.   else {
22.    /* Если переменная TERM определена, предоставим  *
23.     * пользователю возможность изменить её значение,*
24.     * после чего поместим её в окружение процесса   */
25.     printf("TERM=%s. Change? [N]", term);
26.     gets(buf);
27.     if (buf[0] == 'Y' || buf[0] == 'y'){
28.       printf("TERM="); gets(buf);
29.       sprintf(var, "TERM=%s", buf); putenv(var);
30.       printf("new %s\n", var);
31.     }
32.   }
33. }
```

В строке 11 проверяется, определена ли переменная *TERM*. Если переменная *TERM* не определена, функция *getenv* возвращает NULL.

В строках 17 и 29 формируются аргументы для функции *putenv*, содержащие новые значения переменной *TERM*. Далее, в строках 18 и 29 соответсвенно, устанавливается новое значение для переменной *TERM*.

Введённое значение переменной будет действительно только для данного процесса и порождённых им процессов: если после завершения программы вывести значение *TERM*, то видно, что оно не изменилось.

Лабораторная работа №7.

Взаимодействие программ с командной оболочкой. Обработка исключительных ситуаций.

Задание на лабораторную работу

Написать программу на языке Си, состоящую из двух файлов: заголовочного файла и файла программы.

В заголовочном файле должны быть подключены необходимые системные заголовочные файлы, и описаны все глобальные переменные и функции, используемые в файле программы.

Программа должна выполнять следующие действия:

1) вывести первые 10 переменных окружения на экран двумя различными способами (способ задаётся пользователем в командной строке опциями -1 и -2);
2) обработать все указанные опции командной строки и в случае ошибочных опций выдать сообщение в стандартный поток ошибок о неправильном использовании программы;
3) в командной строке опцией *-f файл* указывается имя файла, который необходимо открыть и вывести его содержимое на экран. Также необходимо обработать ошибки, которые могут возникнуть при открытии, закрытии и выводе файла на экран;
4) при завершении программы (даже в случае возникновения ошибок) на экран должно быть выдано сообщение об авторе программы: ФИО, идентификатор пользователя.

Контрольные вопросы

1. Этапы создания программы с использованием языка высокого уровня.
2. Заголовочные файлы. Зачем используются, где находятся?
3. В чём отличие использования символов < > и " " в директиве препроцессора *#include*?
4. Зачем используются производные системные типы (имеющие окончание «t»)?
5. Этапы компиляции программ, написанных на языке Си.
6. Зачем нужны файлы, имеющие суффикс *.o*?
7. Зачем нужна программа-компоновщик *ld*?
8. Способы получения переменных окружения.
9. Обработка параметров командной строки без использования системных функций.
10. Функции *getopt* и *getopt_long*. Принципы их работы.
11. Способы завершения программ. Команды *exit, _exit*.
12. Обработчики завершения программ.
13. Функции обработки ошибок в программах.

Тема 8. Взаимодействие «родственных» процессов и нитей

Цель работы: Уточнение понятий процесс и программа. Изучение основных принципов порождения и завершения «родственных» процессов.

8.1. Теоретическое введение

Функционирование любой ЭВМ заключается в выполнении строго определённой последовательности действий, называемой **программой** или **задачей**. «*Запустить программу на выполнение*» означает загрузить саму программу (написанную на понятном для ЭВМ языке) и необходимые для неё данные в оперативную память и настроить процессор (значения его

внутренних регистров) так, чтобы он стал выполнять инструкции, начиная с соответствующей ячейки оперативной памяти.

В мультизадачных системах, к которым относится и ОС Linux, в оперативную память может быть загружено сразу несколько программ. При этом, чередуя выполнение на одном процессоре инструкций то одной программы, то другой, ЭВМ создаёт «иллюзию» их одновременного выполнения. Правила, по которым чередуются инструкции разных программ, определяются операционной системой (подсистема планирования процессов или нитей). При этом программы ничего не знают о существовании друг друга[4].

В многопроцессорных (многоядерных) системах каждый процесс может выполняться на собственном процессоре (ядре). При этом каждый процессор, в свою очередь, может использоваться для выполнения нескольких (псевдо)параллельных процессов.

Для того чтобы операционная система могла определить, инструкцию какой программы следует выполнить следующей, должны выполняться два условия: во-первых, после выполнения очередной инструкции одной из программ процессор должен переключиться на операционную систему[5], и, во-вторых, операционная система должна обладать информацией обо всех «выполняющихся» программах (место в оперативной памяти, где располагаются инструкции программы; номер последней выполненной инструкции; значения регистров процессора после выполнения последней инструкции и т.п.). Информация обо всех выполняющихся программах хранится в оперативной памяти вместе с операционной системой.

Совокупность, состоящая из программы, описывающей её информации и дополнительных служебных данных, называется **процессом**. Информация о программе хранится в операционной системе в виде специальной структуры, называемой **дескриптором процесса**. Дескриптор процесса и дополнительная информация о программе вместе называются **контекстом процесса**.

8.2. Атрибуты процесса

Любой процесс в операционной системе описывается **дескриптором**, содержащим ряд параметров, включая:

- уникальный идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- связанный с процессом терминал (TTY);
- идентификаторы пользователя (UID, RUID, EUID, FSUID, SUID) и группы (GID, RGID, EGID, FSGID, SGID), от имени которых

[4] Следует помнить, что защита программ реализуется как на аппаратурном уровне (защищённый режим процессора), так и на уровне операционной системы.

[5] Напомним, что для ЭВМ операционная система является обычной программой, также загруженной в оперативную память.

процесс выполняется и пытается осуществлять действия в рамках операционной системы или файловой системы;
- номер группы процессов (GROUP);
- идентификатор пользовательского сеанса (SESSION);
- и т.д.[6]

Уникальный идентификатор процесса (PID) – это целое число, позволяющее ядру системы различать процессы, т.е. своего рода «имя» процесса. По сути, PID – это номер ячейки в таблице дескрипторов процессов, выполняемых под управлением операционной системы. Когда создаётся новый процесс, ядро операционной системы использует следующую свободную запись в таблице дескрипторов. Если достигнут конец таблицы, то производится поиск свободной ячейки, начиная с начала таблицы. Когда процесс завершает свою работу, ядро освобождает занятый им идентификатор и соответствующий ему дескриптор.

Каждый процесс «запоминает» своего родителя в поле **PPID**. Зная значения поля PID и PPID, можно построить иерархию порождения процессов, начиная с самого первого процесса в системе (он обычно имеет PID = 0). Такая иерархия называется **деревом процессов**. Значение поля PPID играет особую роль при завершении процесса (см. ниже).

Как известно, каждому интерактивному процессу при создании назначаются потоки для: ввода информации, для её вывода и для вывода ошибок (stdin, stdout, stderr)[7]. Обычно все эти потоки указывают на один терминал, который называется «связанным» с процессом, и имя этого терминала (имя файла устройства) помещается в параметр **TTY**.

Все процессы в операционной системе запускаются каким-либо пользователем[8]. Идентификатор пользователя, запустившего процесс, помещается в параметр «**реальный идентификатор пользователя»** (англ. UID или RUID, Real UID). Часто возникает необходимость, чтобы процесс был запущен одним пользователем, а имел возможность получать доступ к ресурсам, принадлежащим другому пользователю. Контроль доступа к ресурсам системы осуществляется, исходя из значения поля «**эффективный идентификатор пользователя**» (англ. EUID, Effective UID). Примером ситуации выполнения действий от имени другого пользователя может служить смена пользовательского пароля с помощью команды *passwd*. Очевидно, что если база паролей (файлы */etc/passwd* или */etc/shadow*) будет доступна на запись кому угодно, тогда нельзя будет говорить ни о какой безопасности системы. Поэтому *passwd* настроена таким образом, что независимо от того, какой пользователь её выполняет, действия будут производиться всегда от имени суперпользователя (англ. root, UID = 0). При

[6] На самом деле дескриптор процесса содержит значительно больше полей. Здесь приведены только те, которые используются в данном учебном пособии.

[7] О переопределении потоков ввода-вывода рассказывалось в п. **Ошибка! Источник ссылки не найден.**.

[8] Считается, что загрузку операционной системы выполняет суперпользователь (администратор, root).

этом ответственность за правильную и допустимую смену пароля несёт *passwd*. «Настроена» означает, что владельцем файла является пользователь *root*, и файл имеет специальное право «*Установить эффективного пользователя*»[9]. При запуске такого файла в поле EUID операционная система поместит идентификатор владельца файла, и все действия будут осуществляться уже от его имени. Очевидно, что значения полей RUID и EUID могут и совпадать (когда пользователь, запустивший процесс, выполняет все действия от своего имени). В ходе выполнения пользовательский процесс имеет возможность изменять значения RUID и EUID (менять их местами или делать одинаковыми[10]). При изменении значения поля EUID его изначальное значение сохраняется в поле «**сохранённый идентификатор пользователя**» (англ. SUID, Saved UID). Кроме эффективного идентификатора в дескрипторе процессов в ОС Linux имеется нестандартное поле **FSUID** (англ. File System UID), которое используется при обращении процесса к файловой системе. При запуске процесса FSUID эквивалентен EUID. Всё сказанное про идентификаторы пользователей справедливо и для основной группы пользователя (поля GID или RGID, EGID, SGID, FSGID).

В операционной системе Linux процессы, выполняющие одну задачу, **объединяются в группы**. Например, если в командной строке задано, что процессы связаны при помощи программного канала, они обычно помещаются в одну группу процессов. Каждая группа процессов обозначается собственным идентификатором. Процесс внутри группы, идентификатор которого совпадает с идентификатором группы процессов, считается **лидером группы процессов**.

В свою очередь, каждая группа процессов принадлежит к **сеансу пользователя**. Сеанс обычно представляет собой набор из одной группы процессов переднего плана, использующей терминал, и одной или более групп фоновых процессов. Каждый сеанс также обозначается идентификатором, который совпадает с PID процесса-лидера. При завершении пользовательского сеанса все принадлежащие ему процессы автоматически завершаются системой. Именно поэтому *процессы-демоны обязательно должны иметь идентификатор сеанса, отличный от сеанса пользователя*, который их запустил.

8.3. Получение информации о состоянии и атрибутах процессов

8.3.2. Из командной строки

Чтобы получить информацию о том, какие процессы запущены в данный момент в системе и в каком состоянии они находятся, можно

[9] Подробнее о правах на объекты файловой системы говорится в п.**Ошибка! Источник ссылки не найден.**.

[10] Пользователю доступны только эти поля. Суперпользователь может изменять поля RUID, EUID, FSUID.

использовать команду **ps** (англ. process status) с определённым набором опций (8.1).

Таблица 8.1. Опции команды *ps*.

Опция	**Действие**
-a	Выдать все процессы системы, включая лидеров сеансов.
-d	Выдать все процессы системы, исключая лидеров сеансов.
Опция	**Действие**
-e	Выдать все процессы системы.
-x	Выдать процессы системы, не имеющие контрольного терминала.
-o	Определяет формат вывода. Формат задаётся как параметр опции в виде символьной строки, поля которой (таблица) разделяются символом «,» (запятая).
-u	Выдать процессы, принадлежащие указанному пользователю. Пользователь задаётся в параметре опции в символьном виде.

Таблица 8.2. Поля форматного вывода команды *ps*.

Формат	**Значение**
F	Статус процесса.
S	Состояние процесса.
UID	Идентификатор пользователя.
PID	Идентификатор процесса.
PPID	Идентификатор родительского процесса.
PRI	Текущий динамический приоритет процесса.
NI	Значение приоритета процесса.
TTY	Управляющий терминал процесса.
TIME	Суммарное время выполнения процесса процессором.
STIME	Время создания процесса.
COMMAND	Имя команды, соответствующей процессу.
SESS	Идентификатор сеанса процесса.
PGRP	Идентификатор группы процесса.

Например, список запущенных процессов Вашей системы можно получить, используя команду *ps -e*. Информацию о группах и сессиях процесса можно получить так:

```
$ ps -axo pid,pgrp,session,command
```

Можно также посмотреть «дерево» процессов, используя команду **pstree**.

Чтобы увидеть использование ресурсов «в динамике», можно использовать команду **top**. Выход из просмотра осуществляется нажатием клавиши «q».

8.3.3. При выполнении процессов

Для получения значений идентификаторов процесс может использовать следующие системные вызовы:

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

pid_t getpid  (void)
pid_t getppid (void)
uid_t getuid  (void);
uid_t geteuid (void);
gid_t getgid  (void);
gid_t getegid (void);
pid_t getpgrp (void);
pid_t getsid (pid_t pid);
```

Назначение функций очевидно из названия. Например, функция *getpid* вернёт идентификатор процесса (PID). Все функции, кроме *getsid*, не принимают никаких параметров и возвращают значение соответствующих полей из дескриптора текущего (вызвавшего функцию) процесса. Функция *getsid* позволяет узнать идентификатор сеанса для любого процесса в системе (если это позволено соответствующему пользователю), идентификатор которого передаётся в качестве параметра. Если передать вызову *getsid* значение 0, то он вернёт идентификатор сеанса вызывающего процесса (т.е. это эквивалентно записи *getsid(getpid());*).

Процесс может создать новую группу или присоединиться к существующей, а также определить новый сеанс при помощи системных вызовов:

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

pid_t setpgid(pid_t pid, pid_t pgid);
pid_t setsid(void);
```

Вызов *setpgid* устанавливает идентификатор группы процесса с идентификатором, равным *pid*, в *pgid*. В случае ошибки возвращается –1.

Изменить значения полей RUID, EUID, FSUID, RGID, EGID, FSGID можно при помощи системных вызовов:

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

int setuid   (uid_t uid);
int seteuid  (uid_t euid);
int setreuid (uid_t ruid, uid_t euid);
int setfsuid (uid_t fsuid);
int setgid   (gid_t gid);
int setegid  (gid_t egid);
int setfsgid (gid_t fsgid);
int setregid (gid_t rgid, gid_t egid);
```

Назначение функций очевидно из названия. Пояснений требуют лишь функции *setreuid* и *setregid*. Обе эти функции предназначены для изменения полей RUID, EUID и RGID, EGID соответственно в ситуации, когда операционная система не поддерживает дополнительных полей SUID, SGID. Эти функции включены в библиотеку для реализации совместимости с такими операционными системами как 4.3 UNIX BSD, которые не имеют в дескрипторах процессов указанных полей. За подробной информацией об использовании этих функции следует обратиться к электронной справочной системе *man*.

Очевидно, что процесс может получить информацию только о своих атрибутах, и при этом процесс всегда находится в состоянии Running (R).

Использование функции установки значений полей дескриптора процесса рассмотрим на примере создания процесса-демона (листинг 5).

Листинг 5. Создание процесса-демона

```
1. #include <unistd.h>
2.
3. int main(void){
4.   close (0);
5.   close (1);
6.   close (2);
7.   setsid(0);
8.   setpgrp(0,0);
9.
10.  /* тело процесса-демона */
11.  sleep(10);
12.  return (0);
13.}
```

В первой строке подключается необходимый заголовочный файл. В данном случае нам нужны только функции по работе с атрибутами процесса, файловыми дескрипторами и функция *sleep*, которая переводит процесс в состояние ожидания. Далее в строках 4-6 процесс отключается от терминала

(вспомним, что «демон» не имеет «присоединённого» терминала). После чего создаёт новый сеанс и группу (строки 7-8). Затем процесс может выполняться независимо от того, находится ли запустивший его пользователь в системе или уже завершил свой сеанс. В нашем случае «демон» просто «спит» 10 секунд (строка 11) и затем завершается (строка 12).

8.4. Потоки одного процесса (нити)

Точно так же, как многозадачная операционная система может выполнять несколько программ одновременно, один процесс может обрабатывать несколько потоков команд, которые называются **нитями.** Каждая нить представляет собой независимо выполняющуюся функцию со своим счётчиком команд, регистровым контекстом и стеком. Понятия процесса и нити очень тесно связаны и поэтому трудноотличимы. Нити даже часто называют *легковесными процессами*[11].

Основные **отличия процесса от нити** заключаются в том, что каждому процессу соответствует своя независимая от других область памяти, таблица открытых файлов, текущая директория и прочая информация уровня ядра. Нити одного процесса наоборот выполняются в общем адресном пространстве, им доступны общие переменные, дескрипторы открытых файлов, терминал и т.д. Другими словами, в системе выполняется несколько независимых процессов, каждый из которых состоит как минимум из одной нити.

8.5. Порождение «дочернего» процесса

Новый «родственный» процесс порождается с помощью системного вызова **fork**:

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

pid_t fork(void);
```

Порождённый или *дочерний процесс* является точной копией родительского процесса за исключением следующих атрибутов:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID). У дочернего процесса он приравнивается к PID родительского процесса (того, кто выполнил вызов *fork*);
- дочерний процесс не имеет очереди сигналов, ожидающих обработки.

[11] В операционной системе Windows NT вводится термин «волокно» (англ. fiber) для обозначения потока, планированием которого занимается не операционная система, а сам процесс.

После завершения вызова *fork*, оба процесса начинают выполнять одну и ту же инструкцию – следующую за вызовом *fork*.

Чтобы процесс мог определить, кем он является (дочерним или родительским), используется возвращаемое значение вызова *fork*. В первом случае, когда процесс является дочерним, *fork* вернёт 0, во втором – PID вновь созданного процесса. Если *fork* возвращает –1, то это свидетельствует об ошибке (естественно, в этом случае возврат происходит только в процесс, выполнивший системный вызов, и новый процесс не создаётся).

Пример программы, порождающей новый процесс, приведён в листинге 6.

Листинг 6. Порождение нового процесса

```
1. #include <stdio.h>
2. #include <unistd.h>
3.
4. int main(void)
5. {
6.   int pid;
7.   pid = fork();
8.   if (pid == -1)
9.   {
10.    perror("fork"); exit(1);
11.  }
12.  if (pid == 0)
13.  {
14.    /*Эта часть кода выполняется дочерним процессом*/
15.    printf("Потомок\n");
16.    sleep(1);
17.  }
18.  else
19.  {
20.    /*Эта часть кода выполняется родительским
21.      процессом*/
22.    printf("Родитель. Создан потомок %u\n", pid);
23.    sleep(1);
24.  }
25.  return (0);
26.}
```

В строках 1-2 подключаются необходимые заголовочные файлы. В данном случае нам требуются функции ввода-вывода (*stdio.h*) и функции по работе с процессами (*unistd.h*). Порождение процесса происходит в строке 7, после чего проверяется возвращаемое значение функции *fork*. Если новый процесс создан, то условие в строке 8 не выполнится, и каждый процесс перейдёт на строку 12. Далее, проверив значение переменной *pid*, дочерний

процесс будет выполнять строки 13-17 (так как в его случае *pid* = 0), а родительский – строки 19-24. Затем оба процесса завершатся с кодом 0 (строка 25).

Особое внимание следует обратить на строки 16 и 22, в которых родительский и дочерний процессы добровольно переходят в «спящее» состояние на 1 секунду. Для чего это сделано? Ответ прост – для того чтобы оба процесса смогли вывести информацию в поток вывода, и она была отображена на экране терминала. Если этого не сделать, то может возникнуть ситуация, когда один из процессов (в силу алгоритмов планирования) завершит свою работу быстрее, чем другой начнёт выводить информацию в поток вывода. В этом случае завершившийся процесс автоматически закроет все открытые файлы, которыми в нашем случае являются потоки ввода-вывода. В силу того, что файловые дескрипторы у родственных процессов одинаковые, ещё не завершившийся процесс начнёт выдавать информацию в поток, у которого соответствующий файл уже закрыт. И на экран никакой информации выдано не будет.

8.6. Неименованные (программные) каналы для взаимодействия «родственных» процессов

Вспомните синтаксис организации программных каналов при работе в командной строке *shell*:

```
$cat myfile | sort
```

При этом (стандартный) вывод программы *cat*, которая выводит содержимое файла *myfile*, передаётся на (стандартный) ввод программы *sort*, которая, в свою очередь, отсортирует предложенный материал.

Таким образом, два процесса обменялись данными. При этом использовался программный канал, обеспечивающий однонаправленную передачу данных между двумя задачами.

Для создания канала в программе используется системный вызов **pipe**.

```
#include <unistd.h>

int pipe (int *filedes);
```

Этот вызов возвращает два файловых дескриптора: *filedes*[1] для записи в канал и *filedes*[0] для чтения из канала. Теперь, если один процесс записывает данные в *filedes*[1], то другой сможет получить эти данные из *filedes*[0]. Вопрос только в том, как другой процесс сможет получить сам файловый дескриптор *filedes*[1]? Вспомним наследуемые атрибуты при создании процесса. Дочерний процесс наследует и разделяет все назначенные файловые дескрипторы родительского. То есть доступ к дескрипторам *filedes* канала может получить сам процесс, вызвавший *pipe*, и его дочерние

процессы. В этом заключается серьёзный недостаток каналов, поскольку они могут быть использованы для передачи данных только между родственными процессами. Каналы не могут использоваться в качестве средства межпроцессного взаимодействия между независимыми процессами.

Для работы с каналом `pipe` используются системные вызовы `write` и `read` соответственно для записи и чтения данных в канал и из канала. Прототипы этих функций имеют следующий вид:

```
ssize_t write( int fd,const void *buf, size_t count);

ssize_t read( int fd, const void *buf,  size_t count);
```

где fd – файловый дескриптор канала, *buf – указатель на буферную переменную, из которой или в которую будет производиться запись или чтение данных в канал или из канала, count- количество байт записываемых или считываемых из канала.

Хотя в приведённом примере может показаться, что процессы *cat* и *sort* независимы, на самом деле оба этих процесса создаются процессом *shell* и являются родственными.

Лабораторная работа №8. Программирование в ОС Linux. Взаимодействие «родственных» процессов.

Задание на лабораторную работу

Написать программу на языке Си,

выполняющую следующие действия:

- создание программного канала и порождение дочернего процесса;
- **основной процесс считывает содержимое файла , *созданного в лабораторной работе №1* , и** передаёт его в канал;
- дочерний процесс получает информацию через канал и выводит эту информацию в файл с новым именем
- передача завершается при поступлении в канал символа с кодом 26.
- при получении информации дочерний процесс производит замену символов по схеме: b $\rightarrow$ N, i $\rightarrow$ A, n $\rightarrow$ D.

Тема 9. Взаимодействие «неродственных» процессов

9.1. Идентификаторы ресурсов в межпроцессном взаимодействии

Средства межпроцессного взаимодействия первого типа требуют наличия средств идентификации совместно используемых объектов. Множество возможных имён объектов конкретного типа межпроцессного взаимодействия называется **пространством имён** (англ. name space). Имена являются важным компонентом системы межпроцессного взаимодействия (англ. InterProcess Communications, IPC) для всех объектов, поскольку позволяют различным процессам получить доступ к общему объекту.

Имя для объектов IPC называется **ключом** (англ. key) и генерируется функцией **ftok**, исходя из: имени некоторого доступного для всех процессов-участников взаимодействий файла и односимвольного идентификатора:

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(char *filename, char proj);
```

В качестве первого параметра используется имя некоторого файла (полное или относительное), известное всем взаимодействующим процессам. Например, это может быть имя программы-сервера. Важно, чтобы этот файл существовал на момент создания ключа, при этом содержание файла значения не имеет. Также нежелательно использовать файл, который создаётся, изменяется и/или удаляется в процессе взаимодействия процессов.

Одному идентификатору (ключу) может соответствовать несколько разнотипных общих ресурсов. Например, набор семафоров и разделяемая память.

Исключением из правила является использование «именованного канала», в котором общий файл непосредственно используется для организации взаимодействий.

9.2. Именованные каналы. FIFO

Название каналов FIFO происходит от выражения First In First Out (первый вошёл, первый вышел). FIFO очень похожи на программные каналы, поскольку являются однонаправленным средством передачи данных, причём чтение данных происходит в порядке их записи. Однако в отличие от программных каналов, FIFO имеют имена, которые позволяют независимым процессам получить к этим объектам доступ. Поэтому иногда FIFO также называют именованными каналами. FIFO является отдельным типом файла в файловой системе UNIX (команда *ls -l* покажет символ «p» в первой позиции).

Для создания FIFO используется системный вызов:

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

int mknod(char *pathname, mode_t mode, dev_t dev);
```

Где *pathname* – имя файла в файловой системе (имя FIFO), *mode* – флаги владения, прав доступа и т.д., *dev* при создании FIFO игнорируется. FIFO может быть создан и из командной строки *shell*:

```
$ mknod name p
```

После создания FIFO может быть открыт для записи и чтения, причём запись и чтение могут происходить в разных независимых процессах. Каналы FIFO и обычные каналы работают по следующим правилам:

1. При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
2. При чтении большего числа байтов, чем находится в канале или FIFO, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов *read* будет заблокирован до появления данных (если для канала или FIFO не установлен флаг отсутствия блокирования *O_NDELAY*).
4. Запись числа байтов, меньшего ёмкости канала или FIFO, гарантированно атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются.
5. При записи большего числа байтов, чем это позволяет канал или FIFO, вызов *write* блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал SIGPIPE, а вызов *write* возвращает 0 с установкой ошибки (*errno=EPIPE*) (если процесс не установил обработку сигнала SIGPIPE, производится обработка по умолчанию, и процесс завершается).

В качестве примера приведём простейшую программу типа клиент-сервер, использующую FIFO для обмена данными. Следуя традиции, клиент

посылает серверу сообщение «Здравствуй, Мир!», а сервер выводит это сообщение на терминал.

Листинг 7. Сервер

```
1.  #include <sys/types.h>
2.  #include <fcntl.h>
3.  #include <stdio.h>
4.  #include <sys/stat.h>
5.  #define FIFO "fifo.1"
6.  #define MAXBUFF 80
7.
8.  int main (void){
9.    int fd, n;
10.   char buff[MAXBUFF]; /*буфер для чтения данных */
11.   /*Создадим специальный файл FIFO */
12.   if (mknod(FIFO, S_IFIFO | 0666, 0) < 0){
13.     printf("Невозможно создать FIFO\n");
14.     exit(1);
15.   }
16.   /*Получим доступ к FIFO*/
17.   if ((fd = open(FIFO, O_RDONLY)) < 0){
18.     printf("Невозможно открыть FIFO\n");
19.     exit(1);
20.   }
21.   /*Прочитаем сообщение ("Здравствуй, Мир!") */
22.   /* и выведем его на экран                  */
23.   while ((n = read(fd, buff, MAXBUFF)) > 0)
24.     if (write(1, buff, n) != n){
25.       printf("Ошибка вывода\n");
26.       exit(1);
27.     }
28.   /* Закроем FIFO, и удалим файл */
29.   close(fd);
30.   if (unlink(FIFO) < 0){
31.     printf("Невозможно удалить FIFO\n"); exit(1);
32.   }
33.   exit(0);
34. }
```

Листинг 8. Клиент

```
1.  #include <sys/types.h>
2.  #include <sys/stat.h>
3.  #include <stdio.h>
4.  #include <fcntl.h>
5.
```

Листинг 8. Клиент

```
6.  /*Соглашение об имени FIFO*/
7.  #define FIFO "fifo.1"
8.
9.  int main (void){
10.   int fd, n;
11.   /*Получим доступ к FIFO*/
12.   if ((fd = open(FIFO, O_WRONLY)) < 0){
13.     printf("Невозможно открыть FIFO\n");
14.     exit(1);
15.   }
16.   /*Передадим сообщение серверу FIFO*/
17.   if (write(fd, "Здравствуй, Мир!\n\0", 18) != 18){
18.     printf("Ошибка записи\n"); exit(1);
19.   }
20.   close(fd);
21.   exit (0);
22. }
```

Лабораторная работа №9.
Взаимодействие процессов. Каналы. FIFO

Задание на лабораторную работу

1. Написать две программы, использующие именованные каналы для обмена информацией. Назовем их условно «клиент» и «сервер»
2. Клиент и сервер должны иметь общий заголовочный файл, определяющий требуемые соглашения (имя FIFO, длины строк и т.д.).
3. Сервер создаёт именованный канал согласно соглашению и ждёт поступления от клиента строки формата: «FILENAME~имя_файла». Информация, поступившая до указанной строки, игнорируется. После поступления указанной строки сервер открывает файл, имя которого было передано, и выполняет обработку указанного файла согласно варианту задания (имя файла и задание взять из занятия 6). Далее, всё, что нашел сервер должно быть записано в новый файл. Процедура взаимодействия заканчивается отправкой клиенту имени созданного нового файла.
4. Клиент выводит приглашение пользователю для ввода имени файла, после чего формирует строку указанного формата и передаёт её серверу. После получения итогового сообщения от сервера клиент выводит информацию из файла на экран.

Листинг 8. Клиент

Контрольные вопросы

1. Могут ли процессы получать непосредственный доступ к данным других процессов? Почему?
2. Программные каналы. Зачем нужны? Принцип функционирования.
3. Функция *pipe*.
4. Именованные каналы и их отличие от программных.
5. Функция *mknod*.