Лабораторная работа по курсу "Операционные системы"

Разработка программ на языке C/C++ в ОС Linux

Цель работы: научиться разрабатывать программы на языке C/C++ в ОС Linux; работать с параметрами командной строки и переменными окружения, выполнять системные вызовы.

1. Этапы создания программы на языке C/C++

В ОС Linux имя компилятора языка C - **gcc**, языка C++ - **g++**. Для языка C также можно использовать компилятор **cc**, имеющийся во многих ОС семейства UNIX. На рис. 1 представлена последовательность разработки программы на языке C. Для языка C расширения исходных файлов - **.c** и **.h**, для языка C++ - **.cpp**, **.hpp**, **.cxx**, **.hxx**, **.C** и **.H**. Расширение объектных файлов - **.o**, библиотек объектных файлов - **.a, .so**. Исполняемые файлы в Linux обычно без расширения.

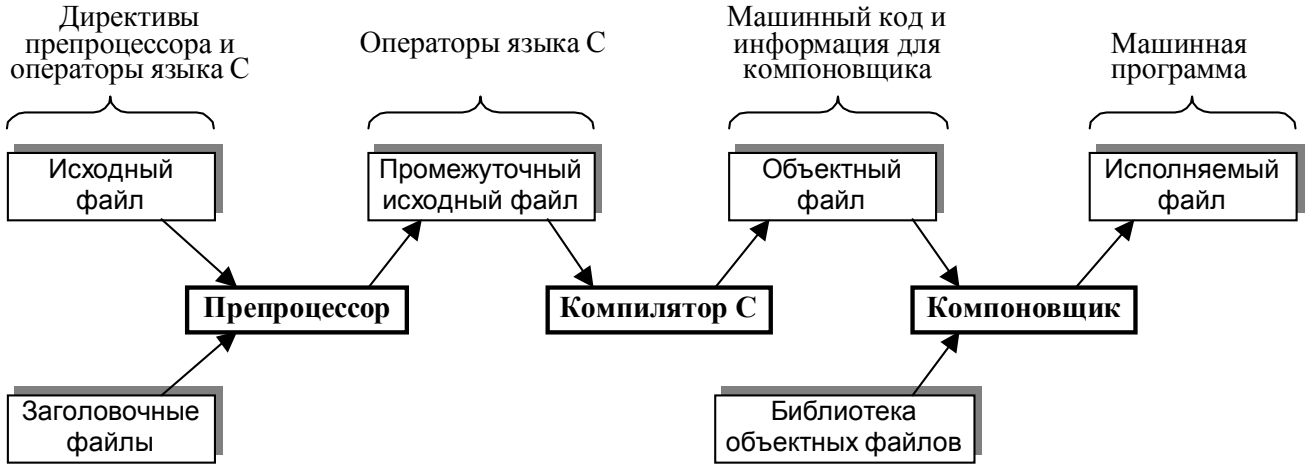


Рис. 1. Последовательность разработки программы на языке C

Задание 1. Запишите назначение основных опций компилятора **gcc** (**-c**, **-S**, **-E**, **-o**). Воспользуйтесь командой **$man gcc** или **$info gcc**

Последовательность команд для создания программы.

1. Программа состоит из одного исходного файла

1) В текстовом редакторе (**gedit**, **pico**, **vi**, **emacs** и т.п.) создать и сохранить исходный текстовый файл (например, **myprog.c**)	**$gedit** ..... работа с gedit
2) Откомпилировать и скомпоновать программу. Результат - исполняемый файл (например, **myprog**)	**$gcc -o myprog myprog.c**
3) Выполнить программу	**$myprog** или **$myprog param1 param2**

2. Программа состоит из нескольких исходных файлов (module1.c, module2.c, module3.c)

1) В текстовом редакторе создать и сохранить каждый из исходных файлов	**$gedit** ..... работа с gedit
2) Откомпилировать отдельно каждый текстовый файл. Результат - объектные файлы (**module1.o**, **module2.o**, **module3.o**)	**$gcc -c module1.c** **$gcc -c module2.c** **$gcc -c module3.c**
3) Скомпоновать объектные файлы. Результат - исполняемый файл.	**$gcc -o myprog module1.o module2.o module3.o**
4) Выполнить программу	**$myprog**

3. В программу нужно включить заголовочный файл, находящийся в некотором каталоге.

Компилятор по умолчанию ищет включаемые (заголовочные) файлы в текущем каталоге и в каталогах, в которых установлены заголовочные файлы для стандартных библиотек. Если заголовочный файл находится в каком-либо другом каталоге, то следует использовать команду для вызова компилятора **gcc/g++** с опцией **-I**. Пусть, например, нужно включить в программу файл **obr.h** из каталога **/home/ivanov/incl** и пусть каталог **/home/ivanov/texts** является текущим. Тогда следует дать команду

$gcc -c -I ../incl obr.c

Или можно указать абсолютное имя для каталога **incl**:

$gcc -c -I /home/ivanov/incl obr.c

При этом в программе директива **include** должна иметь вид **#include "obr.h"**

4. В программу нужно включить нестандартную библиотеку объектных модулей.

Стандартная библиотека С компонуется в исполняемый файл автоматически. Для включения нестандартной библиотеки следует скомпоновать программу с опцией **-l**, например:

$gcc -o obr main.o obr.o -lpthread

По этой команде в программу будет включена библиотека **pthread**. При этом компоновщик будет искать библиотеки в ряде стандартных каталогов, включая каталоги **/lib** и **usr/lib**. Если же библиотека, которую необходимо включить, находится в каком-либо ином каталоге, то в командной строке следует использовать опцию **-L** совместно с опцией **l**, например:

$gcc -o obr main.o obr.o -L/usr/local/libs -lpthread

По этой команде библиотека **pthread** будет включена из каталога **/usr/local/libs**. Если данный каталог является текущим, то команду можно задать в следующей форме:

$gcc -o obr main.o obr.o -L. -lpthread

Задание 2.

2.1. Напишите и выполните программу **Hello**, выводящую строку **"Hello, world"**.

2.2. Определите главное отличие **stdout** от **stderr**. Для этого выполните программу, добавив

… while (1) {fprintf(stdout, "a"); sleep(1);} …

Затем выполните программу, заменив **stdout** на **stderr**. Запишите Ваши выводы в отчет.

2.3. (необязательное задание) Напишите программу **obr**, состоящую из двух модулей: модуль **obr.c** содержит функцию **double obr (int i)**, возвращающую число, обратное числу **i**; модуль **main.c** содержит функцию **main()**, которая запрашивает целое число **i** и выводит **obr(i)**.

2. Использование утилиты make

Утилита **make** используется для автоматизации разработки программ. В текстовом файле с именем **Makefile** указывается следующая информация: целевые файлы (цели), которые необходимо построить; правила для их построения; зависимости, определяющие, когда данную цель необходимо перестроить заново. Так, для программы 2.3 **Makefile** может иметь следующий вид:

```
CFLAGS=-c
CC=gcc
obr: main.o obr.o
 <Tab> $(CC) -o obr main.o obr.o
main.o: main.c
<Tab> $(CC) $(CFLAGS) main.c
obr.o: obr.c
<Tab> $(CC) $(CFLAGS) obr.c
```

Здесь слева от '**:**' указана цель, справа - ее зависимости. Правило для построения цели указано на следующей строке, которая должна начинаться с символа табуляции. **CC** и **CFLAGS** - переменные утилиты **make** (в данном примере **CC** задает имя компилятора, **CFLAGS** - опцию компилятора). Значение для переменной может быть задано в файле **Makefile** (как в примере) и/или в командной строке (например, **$make CC=g++**). После подготовки файла **Makefile** для создания исполняемой программы достаточно ввести команду

$make или

$make имя_make_файла (если **имя_make_файла** отлично от **Makefile**)

По этой команде утилита **make** выполнит файл **Makefile** из текущего каталога и автоматически перетранслирует только те файлы, которые необходимо.

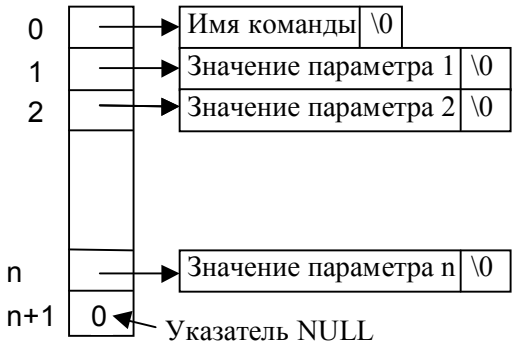
Задание 3. (Необязательное задание)
3.1. Создайте файл **Makefile** для программы 2.3, предварительно удалив файлы ***.o** и **obr** из текущего каталога. Затем выполните команду **$make.** Запишите в отчет текст файла **Makefile** с пояснениями и команды, выполненные утилитой **make**.
3.2. Незначительно модифицируйте файл **obr.c**, выполните команду **$make** и запишите команды, выполненные утилитой **make**.

3. Доступ к параметрам командной строки

При запуске программы в командной строке после имени программы могут указываться **параметры** (**аргументы**) **командной строки**. Они разделяются пробелами или, если параметры сами содержат пробелы, заключаются в апострофы. Для доступа к этим параметрам в программе заголовок функции **main** должен содержать два параметра (рис. 2): количество параметров (**argc**) и массив указателей на строки (**argv**), оканчивающийся указателем NULL. **i**-ая строка является значением **i**-го параметра, при этом строка с номером 0 содержит имя программы.

Заголовок функции **main**: `main(int argc, char *argv[])`
или `main(int argc, char **argv)`

Структура ***argv[]**:

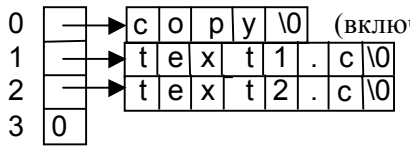


Использование argv в программе:

argv[0] - указатель на имя программы
argv[1] - указатель на первый аргумент
*argv - указатель на имя программы
*argv[2] - первый символ второго аргумента!

Пример: `copy text1.c text2.c`

Структура ***argv[]**:



(включая путь)

argv[0] - указатель на "<путь>/copy\0"
argv[1] - указатель на "text1.c\0"
argv[2] - указатель на "text2.c\0"
argv[3] - указатель NULL

Рис. 2. Доступ к параметрам командной строки из программы на языке С

Задание 4. Запишите в отчет текст примеров и результаты их тестирования с параметрами.
4.1.

```
/* Ex4_1.c Печать параметров командной строки*/
#include  <stdio.h>
main ( int argc, char *argv[] )
{  if  ( argc < 2 )
   { printf( "Usage : %s parameter\n", argv[0] ) ;
      exit ( 1 ) ;
    }
     printf("Starting program %s \n", argv[0] ) ;
     printf("with %d parameter(s)\n", argc-1 ) ;
     printf("First parameter is %s\n", argv[1] ) ;
     exit ( 0 ) ;
}
```

4.2.

```
/* Ex4_3.c Печать произвольного числа параметров */
#include <stdio.h>
main(int argc, char *argv[])
{  for  ( ; *argv ; ++argv )
```

```
        printf("%s\n", *argv ) ;
    }
```

4.3. Измените предыдущий пример, задав параметры целого типа. Для преобразования строки в целое число используйте функцию **atoi**. Получить информацию о стандартных функциях С можно по команде **$info libc**.

4. Переменные окружения

Любая исполняемая программа имеет свое окружение, представляющее собой набор строковых пар вида **переменная=значение**. В соответствии с соглашением имена переменных пишутся заглавными буквами. Доступ к переменным окружения из оболочки был рассмотрен в лабораторной работе 1.

В программе можно получить значение переменной окружения при помощи функции **getenv** из **<stdlib.h>.** Эта функция принимает имя переменной и возвращает ее значение в виде строки символов или NULL, если переменная не определена в окружении. Для установки и очистки переменных окружения используются функции **setenv** и **unsetenv** соответственно. Для доступа ко всем переменным окружения используется специальная глобальная переменная **environ**. Эта переменная имеет тип **char**** и является массивом указателей на символьные строки, заканчивающийся указателем NULL.

Задание 5. Выполните пример, запишите в отчет его текст и результаты тестирования.

```
/* Ex5_1.c Печать переменных окружения */
#include <stdio.h>
extern char** environ;
int main ()
{
    char** var;
    for (var = environ; *var != NULL; ++var)
        printf ("%s\n", *var);
    return 0;
}
```

5. Обработка ошибок системных вызовов

Большинство системных вызовов возвращает ноль (или положительное значение), если операция выполнена успешно, и отрицательное значение в противном случае. В случае ошибок в глобальную переменную **errno** записывается дополнительная информация - целое число, идентифицирующее причину ошибки. Если включить в программу **<errno.h>**, можно ссылаться на ошибки по их символических именам, например, **EACCES** или **EINVAL**.

Получить в программе текстовое сообщение об ошибке можно двумя способами:

1) При помощи функции **strerror** из **<string.h>**; функция принимает **errno** и возвращает строку с описанием ошибки.
2) При помощи функции **perror** из **<stdio.h>;** функция принимает строку, которую она выводит в качестве префикса перед сообщением об ошибке (например, это может быть имя файла, где произошла ошибка) и выводит описание ошибки в поток **stderr**.

Пример. В программе делается попытка открыть файл. Системный вызов **open** возвращает дескриптор файла (целое положительное число) или -1 в случае ошибки.

Вариант 1.
```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
  fprintf (stderr, "error opening file: %s\n", strerror (errno));
  exit (1);
}
```

Вариант 2.
```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {perror("inputfile.txt");  exit (1);}
```

ВНИМАНИЕ: успешное выполнение системного вызова не приводит к сбросу **errno**!

6. Низкоуровневые функции ввода-вывода

При программировании на языке С/С++ можно использовать два набора функций ввода-вывода: стандартные (или высокоуровневые) функции языка С (**printf**, **fopen**, **fprintf** и т.д.) и низкоуровневые функции, предоставляемые ядром ОС Linux (**open**, **read**, **write** и т.д.). Первые являются надстройкой над вторыми. Как правило, низкоуровневая функция непосредственно преобразуется в соответствующий ей системный вызов.

Так, высокоуровневая функция **printf** форматирует и выводит информацию в стандартный выходной поток ввода-вывода. Обобщенный вариант данной функции, **fprintf**, может выводить в произвольный поток. При этом поток представлен указателем на структуру **FILE**. Указатель на **FILE** возвращается функцией **fopen** при открытии файла. После окончания работы необходимо закрыть файл функцией **fclose**. Аналогично работают и другие функции этой библиотеки (**fputs**, **fputc**, **fscanf**, и т.д.).

При использовании же низкоуровневой функции **write** вместо указателя на **FILE** используется ***файловый дескриптор***. Это целое число, которое ссылается на объект открытого файла, не на сам файл. Файловые дескрипторы могут создаваться не только для файлов, но и для других объектов.

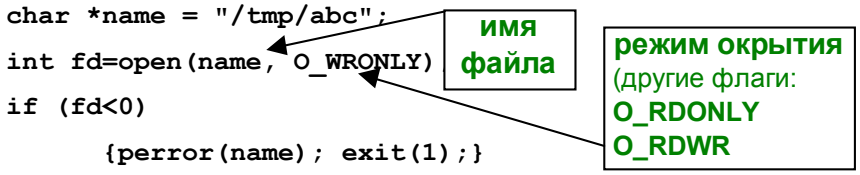
Для работы с низкоуровневыми функциями ввода-вывода (некоторые из них приведены в табл. 1) в программу следует включать заголовочные файлы **<fcntl.h>**, **<unistd.h>**, **<sys/types.h>**, **<sys/stat.h>** (Их имена следует уточнить при помощи команды **man**.)

Таблица 1

Низкоуровневые функции ввода-вывода

Имя	Описание	Возвращаемое значение
open	Открывает (и, возможно, создает) файл для чтения, записи, чтения/записи	Файловый дескриптор или -1
creat	Создает новый (пустой) файл	Файловый дескриптор или -1
read	Читает данные из открытого файла в буфер	Количество реально прочитанных символов или -1
write	Записывает данные в открытый файл из буфера	Количество реально записанных символов или -1
lseek	Передвигает указатель смещения файла для последующих операций **read**/**write**	Текущее смещение в байтах от начала файла или -1
close	Закрывает ранее открытый файл	0 или -1
unlink	Удаляет файл	0 или -1

Пример использования функции **open**:



Если файл открыт только на чтение (с флагом **O_RDONLY**), то при попытке записи в файл произойдет ошибка. Если файл открыт только на запись (с флагом **O_WRONLY**), то при попытке чтения из файла произойдет ошибка. Не все файлы могут быть открыты с любым из трех флагов. Например, разрешения файла могут запрещать его открытие на чтение или запись для конкретного процесса; файл на CD-ROM не может быть открыт для записи и т.п.

Во втором параметре функции **open** можно указывать дополнительные флаги, объединяя их при помощи побитового "или" (|):

O_TRUNC - записываемые данные будут заменять старое содержимое файла.

O_APPEND - записываемые данные будут добавляться в конец файла.

O_CREAT - если файл существует, то он будет открыт; в противном случае файл будет создан (если существует указанный каталог и у процесса есть разрешение на создание файла в этом каталоге). Для флага **O_CREAT** необходимо указывать в **open** третий параметр - разрешения для нового файла, например:

```
fd = open ("myfile", O_WRONLY | O_CREAT | O_APPEND, 0644);
```

O_EXCL - используется совместно с **O_CREAT**. В случае если файл существует, функция **open** вернет код ошибки.

При создании файла (т.е. если указан флаг **O_CREAT**) третьим параметром функции **open** должны указываться разрешения на доступ к создаваемому файлу. Их можно задавать не толь-

ко в виде восьмеричной константы, но и при помощи побитового "или" соответствующих флагов, например:

```
mode_t  mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;
fd = open ("myfile", O_WRONLY | O_CREAT | O_APPEND, mode);
```

Здесь файл открыт с разрешением чтения/записи для владельца и группы, для остальных пользователей разрешено только чтение.

Задание 6.

6.1. Модифицируйте приведенную ниже программу **copyfile.c** копирования файла, добавив в нее комментарии, проверку количества аргументов командной строки (при неверном количестве параметров установите код возврата=1) и обработку ошибок после системных вызовов **open** (коды возврата 2 и 3), **write** (код возврата 4). Протестируйте программу для разных ситуаций.

ПРИМЕЧАНИЕ. Для моделирования ситуации переполнения диска в качестве имени выходного файла задайте **/dev/full**.

```
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define BUF_SIZE 256
int main (int argc, char *argv [])
{
  int input_fd, output_fd;
  int bytes_in, bytes_out;
  char rec [BUF_SIZE];
  input_fd = open (argv [1], O_RDONLY);  /* Здесь код возврата=2 */
  output_fd = open (argv [2], O_WRONLY | O_CREAT, 0666); /*Код возврата=3 */
  while ((bytes_in = read (input_fd, rec, BUF_SIZE)) > 0) {
            bytes_out = write (output_fd, rec, bytes_in);
/* Здесь код возврата=4, если число прочитанных байтов не равно числу записанных*/
  close (input_fd);
  close (output_fd);
  return 0;
}
```

6.2. Напишите и выполните программу **date.c**, которая: а) при помощи функции **open** создает файл **file1** с разрешением чтения/записи для всех пользователей; б) записывает в конец файла **file1** (задайте необходимый флаг при открытии файла!) текущую дату и время. Выполните программу несколько раз и выпишите в отчет содержимое **file1**.

ПРИМЕЧАНИЕ. Для получения даты в виде символьной строки, последовательно примените функции **time** и **ctime** (см. **man**). Перед записью в файл определите длину получившейся строки при помощи функции **strlen** (см. **info**).

Порядок выполнения лабораторной работы

1. Выполните задания 1-6.
2. Занесите в отчет описание заданий со всеми требуемыми пояснениями и результатами.

Требования

1. При подготовке к лабораторной работе (дома) занесите в отчет тексты программ из заданий 2-6 и предполагаемые результаты выполнения.
2. Студент должен знать ответы на следующие вопросы:

Вопросы

1. Как получить промежуточный (после препроцессора) исходный текст программы?
2. Каково назначение флага **-o** в команде вызова компилятора **gcc**?
3. Как создать программу из нескольких модулей?
4. Как добавить в программу библиотеку объектных модулей?
5. Как использовать утилиту **Make** для создания программ?
6. Как получить доступ к параметрам командной строки и переменным окружения?
7. При помощи каких функций можно вывести сообщение об ошибке в системном вызове?
