НУКУССКИЙ ФИЛИАЛ ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИИ ИМЕНИ МУХАММАДА АЛ-ХОРАЗМИЙ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ПРАКТИЧЕСКИМ РАБОТАМ

по предмету

СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ. ВНЕШНИЕ СИСТЕМЫ.

Составил: **Ф.М.Зарипов**

Нукус - 2019

ОГЛАВЛЕНИЕ

ПРАКТИЧЕСКАЯ РАБОТА №1. УПРАВЛЕНИЕ ФАЙЛОВОЙ СИСТЕМОЙ И СБОРКА ПРОЕКТА.

Цель работы: Научиться использовать функции для управление файловой системой, и научиться использовать утилиту make.

Функции управления файловой системой

Смена корневого каталога

Процесс может изменить свой корневой каталог с помощью системного вызова:

#include<unistd.h>

int chroot(const char *path);

Функция chroot() делает каталог path корневым каталогом. После этого поиск файлов с абсолютными именами, начинающимися с '/', будет производиться, начиная с каталога, указанного аргументом path. Заметим, однако, что пользовательский текущий каталог сохраняется.
Для изменения корневого каталога значение эффективного пользовательского ID процесса (EUID) должно соответствовать системному администратору. Системная жесткая ссылка "**..**", входящая в корневой каталог, указывает на него самого. В связи с этим жесткая ссылка "**..**" не может быть использована для доступа к файлам за пределами поддерева, входящего в корневой каталог.
При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Смена текущего каталога

Процесс может изменить текущий каталог с помощью системного вызова:

#include<unistd.h>

int chdir(const char *path);

Функция chdir() изменяет текущий рабочий каталог на path, который может быть относительным или абсолютным именем.Так как с процессом связывается только один текущий каталог, то в многонитиевых приложениях любая нить, вызвавшая chdir(), изменит текущий каталог для всех нитей в этом процессе.
При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

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

int main( int argc, char* argv[] )
{
   if( argc != 2 ) {
      fprintf( stderr, "Use: cd <directory>\n" );
      return EXIT_FAILURE;
   }

   if( chdir( argv[1] ) == 0 ) {
```

```
        printf( "Directory changed to %s\n", argv[1] );
        return EXIT_SUCCESS;
    } else {
        perror( argv[1] );
        return EXIT_FAILURE;
    }
}
```

Создание каталога

Новый каталог можно создать с помощью вызова:

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

int mkdir(const char *path, mode_t mode);
```

Функция mkdir() создает новый пустой каталог, специфицированный в path с разрешениями доступа, заданными в mode в виде комбинации флагов разрешения, определенных в заголовочном файле <sys/stat.h>. ID владельца каталога устанавливается равным эффективному ID пользователя процесса. ID группы каталога устанавливается равным ID группы родительского каталога (если установлен флаг использования ID группы родительского каталога) или эффективный ID группы процесса.
При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

Создается новый каталог с именем /src in /hd:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    mkdir( "/hd/src",
           S_IRWXU |
           S_IRGRP | S_IXGRP |
           S_IROTH | S_IXOTH );

    return EXIT_SUCCESS;
}
```

Удаление каталога

Для удаления каталога используется вызов:

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

int rmdir(const char* path);
```

Функция rmdir() удаляет каталог, специфицированный в path, если его счетчик связей равен 0 и он не открыт ни каким процессом. Каталог должен быть пустым.
При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/*Удаляет каталог с именем /home/terry*/
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void ){
   rmdir( "/home/terry" );

   return EXIT_SUCCESS;
}
```

Создание жесткой связи

Создать связь к существующему файлу можно с помощью вызова:

```
#include <unistd.h>

int link(const char* pname, const char* new);
```

Функция link() создает новый элемент каталога с именем new (путь доступа для новой связи), являющийся жесткой ссылкой на существующий файл с именем pname (путь доступа к существующему файлу), и увеличивает счетчик связей для указанного файла на 1. При этом файл не может быть каталогом или находится на другом устройстве.
При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Создание символической связи

Создать символическую связь с файлом можно с помощью вызова:

```
#include <unistd.h>

int symlink(const char* pname, const char* slink);
```

Функция symlink() создает символическую связь с именем slink, которая содержит абсолютное имя файла pname (slink является именем создаваемой символической связи, pname есть абсолютное имя, содержащееся в символической связи).
Контроль прав доступа к файлу pname не выполняется и отсутствует необходимость существования файла. Символическую связь можно создать к файлу и каталогу даже на другом устройстве.
При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/* Создание символической связи для "/usr/nto/include" в текущем каталоге */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
  {
    if( symlink( "/usr/nto/include", "slink" ) == -1) {
      perror( "slink -> /usr/nto/include" );
      exit( EXIT_FAILURE );
    }
    exit( EXIT_SUCCESS );
  }
```

Чтение символической связи

Содержимое символической связи можно прочитать и поместить в буфер с помощью вызова:

#include <unistd.h>

int readlink(const char* path, char* buf, size_t bufsiz);

Функция readlink() помещает содержание символической связи с именем path в буфер buf. Если readlink() завершается успешно, то bufsiz байтов содержания символической связи помещается в buf. Возвращаемый набор символов размером bufsiz не является строкой (не имеет в конце нуля).
При успешном завершении функция возвращает количество байтов, помещенных в buf. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/* В качестве аргумента программа принимает имя символической связи */
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char buf[PATH_MAX + 1];

int main( int argc, char** argv ){
  int nread, fd;
/* Чтение содержимого символической связи */

if(( nread = readlink( argv[1], buf, PATH_MAX )) == -1) {
     perror( argv[1] );
     exit(EXIT_FAILURE);
}
buf[nread] = '\0';
printf( "Символическаясвязь %s -> %s\n", argv[1], buf );
exit( EXIT_SUCCESS );
}
```

Переименование файла

Для переименования файла или связи используют вызов:

#include <stdio.h>

int rename(const char* old, const char* new);

Функция rename() меняет имя файла, специфицированного в old, на имя, специфицированное в new. Если файл (или пустой каталог) с именем new существует, он заменяется.
При успешном завершении функция возвращает 0. В случае ошибки возвращается значение отличное от 0 и устанавливается errno.

Пример:

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

int main( void )
{
```

```
   if( rename( "old.dat", "new.dat" ) ) {
      puts( "Ошибка переименования old.datв new.dat." );

      return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```

Удаление файла

Удалить файл (или связь) можно с помощью вызова:

```
#include <unistd.h>

int unlink( const char * path );
```

Функция unlink() удаляет файл (связь), чье имя указано в path. Если указана жесткая связь, то она удаляется, а счетчик связей соответствующего файла уменьшается на 1. Если указан файл или символическая связь, то реальное удаление объекта произойдет в тот момент, когда счетчик связей окажется равным 0. До этого момента реальное удаление откладывается.
Эта функция эквивалентна функции remove().
Если каталог, содержащий файл, перезаписываем и у каталога установлен бит S_ISVTX, то процесс может удалять или переименовывать файлы внутри такого каталога только, если выполняется одно или большее количество следующих условий:
- EUID процесса равен UID файла;
- EUID процесса равен UID каталога, содержащего файл;
- процесс имеет право на перезапись файла;
- пользователь является системным администратором (UID = 0).

При успешном завершении функция возвращает 0. В случае ошибки возвращается значение отличное от 0 и устанавливается errno.

Пример:

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

int main( void )
{
   if( unlink( "vm.tmp" ) ) {
      puts( "Error removing vm.tmp!" );

      return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
```

Управление программными проектами с помощью утилиты *make*

qcc –Vgcc_ntox86 hello.c –o hello
qcc hello.c –o hello

./hello

Рассмотрим быть может, глуповатый, но вполне удобный для учебных целей проект. Функция *main()*1 определена в файле main.c:

```
#include "defs.h"
int main()

{
printf("I'm main\n");
aaa();
return 0;
}
```

Все, что делает функция *main(),* — выводит на экран фразу "I'm main", затем вызывает функцию *aaa()* и завершает работу с кодом возврата 0.
В файле main.c есть директива для вставки заголовочного файла defs.h1, который содержит объявления функций *aaa(), bbb()* и *ccc()*:

```
#ifndef _MY_DEFS_
#define _MY_DEFS_
#ifndef _EXT
#define _EXT extern
#endif // _EXT
_EXT void aaa();
_EXT void bbb();
_EXT void ccc();
#endif // _MY_DEFS_
```

Примечание

Директивы препроцессора (все они начинаются с символа #), используемые в defs.h, нужны для избежания многократного включения файла defs.h и конфликтов при объявлении функций в разных комбинациях.
Каждая из этих функций определена в отдельном файле, содержащем директиву включения того же самого заголовочного файла defs.h. Функция *aaa()* определена в файле aaa.c. Она выводит на экран текст "I'm aaa" и вызывает функцию *bbb()*:

```
#include "defs.h"
void aaa()
{
printf("I'm aaa\n");
bbb();
}
```

Функция *bbb()* определена в файле bbb.c. Она выводит на экран текст "I'm bbb" и вызывает функции *ccc()*:

```
#include"defs.h"
void bbb()
{
printf("I'm bbb\n");
```

```
ccc();
}
```

Функция *ccc()* определена в файле ccc.c и выполняет всего одно действие — выводит на экран текст "I'm ccc":

```
#include"defs.h"
void ccc()
{
printf("I'm ccc\n");
}
```

Для того чтобы из этих пяти файлов получить программу **program**, следует выполнить такие команды:

qcc –c main.c
qcc –c aaa.c
qcc –c bbb.c
qcc –c ccc.c
qcc –o program main.o aaa.o bbb.o ccc.o

Выполнение такой цепочки команд называют *сборкой проекта*. Если изменить файл ccc.c, то для получения свежей версии программы **program** потребуется вновь выполнить две последние команды, а если изменить файл main.c, то потребуется повторить первую и последнюю команды. Хорошо еще, что мы не используем многочисленные дополнительные опции, которые могут быть разными для компиляции разных файлов. Конечно, можно было задать одну команду:

qcc –o program main.o aaa.o bbb.o ccc.o

Однако в этом случае сборка проекта будет выполняться как одна, если можно так выразиться, транзакция и при сборке свежей версии проекта будут перекомпилироваться все файлы.
Короче говоря, проще написать некий конфигурационный файл, описывающий зависимости между файлами проекта и то, какую последовательность команд надо выполнить, чтобы пересобрать проект при внесении изменений, не затрагивая при этом те файлы, на которые изменения не повлияли. Таким файлом управления сборкой проекта является Makefile, а анализируется этот файл утилитой **make**. Точнее, при запуске утилита **make** ищет сначала файл с именем makefile, затем, если не нашла, — файл с именем Makefile. В принципе, можно задать любое имя, указав его утилите **make** с помощью опции **–f**.

Примечание
Кстати, интегрированные среды разработки также используют утилиту **make**, скрывая это от программиста за графическим интерфейсом.
Простейший файл Makefile содержит только два типа синтаксических конструкций — цели и макросы.
Цель — это имя файла, который следует сгенерировать. Описание цели меет такой вид:
ЦЕЛЬ: ЗАВИСИМОСТЬ-1 ЗАВИСИМОСТЬ-2 ... ЗАВИСИМОСТЬ-N
КОМАНДА-1
КОМАНДА-1
...

КОМАНДА-N

ЦЕЛЬ-*i* — непустой список файлов, которые предполагается создать.
ЗАВИСИМОСТЬ-*i* — список файлов, из которых строится цель.
В качестве зависимости может использоваться другая цель. Имя цели и список зависимостей записываются в одну строку и разделяются двоеточием. Они составляют *заголовок цели*. Ниже следует список *команд*, каждая команда — в отдельной строке.

Примечание
Каждая команда ОБЯЗАТЕЛЬНО начинается с символа табуляции.
Утилиту **make** можно запускать или с указанием имени конкретной цели, или без имени цели. Если цель не указана, то **make** строит первую цель, заданную в файле Makefile, имя которой не начинается с точки.
Построение цели заключается в выполнении команд, заданных для цели. Перед тем как строить цель, **make** сравнивает время последней модификации цели и зависимостей. Если какая-либо из зависимостей была изменена после последней сборки цели, то цель пересобирается.
Если какая-то зависимость сама является целью (т. е. "подцелью"), то сначала собирается "подцель".
Итак, с учетом сказанного, напишем файл управления сборкой нашего проекта с именем mf1:

```
program: main.o aaa.o bbb.o ccc.o
qcc -o program main.o aaa.o bbb.o ccc.o

main.o : main.c defs.h
qcc –c main.c
aaa.o : aaa.c defs.h
qcc –c aaa.c

bbb.o : bbb.c defs.h
qcc –c bbb.c

ccc.o : ccc.c defs.h
qcc –c ccc.c
```

Выполним сборку:
make –f mf1
Примечание
Если вы выполняли все описанные команды в порядке изложения материала, то программа (цель) **program** уже существует в самой свежей, так сказать, версии. Из-за этого утилита **make** не будет выполнять сборку, а выведет сообщение:

```
make: 'program' is up to date
```

Для верстки: следующий абзац - перевод предыдущей строки, сделать такой же отступ, а шрифт сделать такой же, как в предыд. Переводах (make: исходный код программы program не изменялся со времени последней компиляции)
Поэтому сначала удалите файл program или сделайте, например, файл defs.h более "старым", чем program:

touch defs.h

Поскольку утилита **make** вызвана без указания цели, будет вызвана первая цель — program. Для цели program заданы четыре зависимости, каждая из которых является подцелью.

Утилита **make** сначала соберет подцели, а затем и цель. Можно было задать произвольную цель:

make –f mf1 ccc.o

Тогда бы собиралась только цель ccc.o, т. к. среди ее зависимостей нет подцелей.
На самом деле утилита **make** достаточно умна, поэтому если в качестве цели задан объектный файл (т. е. файл с расширением .o) и существует одноименный C-файл, то **make** сама вызовет **qcc** с опцией **–c**. Поэтому мы можем записать наш пример так (назовем новый файл mf2):

```
program: main.o aaa.o bbb.o ccc.o
qcc -o program main.o aaa.o bbb.o ccc.o

main.o aaa.o bbb.o ccc.o: defs.h
```

Заметьте, что мы один и тот же список объектных файлов в трех строчках записали три раза. Чтобы этого не делать, **make** поддерживает макросы, имеющие синтаксис:

ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ

ЗНАЧЕНИЕ может являться произвольной последовательностью символов, включая пробелы и обращения к значениям ранее определенных переменных. После объявления переменной мы можем обратиться к ней так: $(ПЕРЕМЕННАЯ). Вновь откорректируем наш пример (получим файл mf3):

```
OBJS=main.o aaa.o bbb.o ccc.o
program: $(OBJS)
qcc -o program $(OBJS)
$(OBJS): defs.h
```

Некоторые переменные уже предопределены. К ним относятся, например:
- AR=ar — программа-архиватор;
- AS=as — ассемблер;
- CC=cc — компилятор C (**cc** — это ссылка на **qcc**);
- CXX=g++ — компилятор C++;
- CPP=cpp — препроцессор;
- LEX=lex — лексический анализатор C-программ;
- RM=rm -f — команда удаления файлов.

Есть также специальные встроенные макросы, значение которых изменяется в процессе выполнения утилиты **make**:
- **$@** — имя текущей цели;
- **$?** — список зависимостей, более свежих, чем цель;
- **$^** — полный список зависимостей для данной цели;
- **$*** — имя цели без расширения.

Примечание
Полный перечень всех предопределенных макропеременных и зависимостей можно получить с помощью команды:

make –p </dev/null 2>/dev/null

Вставим служебные переменные в наш пример (получим файл mf4):

```
OBJS=main.o aaa.o bbb.o ccc.o
program: $(OBJS)
$(CC) -o $@ $(OBJS)
$(OBJS): defs.h
```

Для удобства, да и для корректности, следовало бы добавить служебную цель, позволяющую удалять результаты компиляции. Такая цель имеет общепринятое название — clean (получим файл mf5):

```
OBJS=main.o aaa.o bbb.o ccc.o
program: $(OBJS)
$(CC) -o $@ $(OBJS)
$(OBJS): defs.h
clean:
$(RM) program $(OBJS)
```

Иногда добавляют и другие цели, например, install.
Поскольку имена целей задаются фактически произвольно, могут возникать некорректные совпадения имен целей и имен файлов проекта. Для обработки возникающих при этом конфликтов предназначена переменная .PHONY, в качестве значений которой указывают имена служебных целей. Проиллюстрируем это в файле mf6:

```
OBJS=main.o aaa.o bbb.o ccc.o
.PHONY=clean
program: $(OBJS)
$(CC) -o $@ $(OBJS)
$(OBJS): defs.h
clean:
$(RM) program $(OBJS)
```

Ну и последний, пожалуй, штрих. В файле Makefile могут присутствовать комментарии. Началом комментария считается символ #, а окончанием — конец строки. Остается только направить читателя к документации утилиты **make** для получения дополнительной информации.

Задачи:

1. Напишите программу для создание каталога. (минимум 3 пример).
2. Напишите программу для удаление каталога. (минимум 3 пример).
3. Напишите программу для переименование файла. (минимум 3 пример).
4. Напишите программу для удаление файла. (минимум 3 пример).
5. Напишите программу 4 функции на отдельных файлах и сделайте сборку проекта.
6. Напишите программу 4 функции на отдельных файлах и реализуйте сборку с командой make.

ПРАКТИЧЕСКАЯ РАБОТА №2. РАБОТА С ФАЙЛАМИ.

Цель работы: Научиться использовать функции работы с файлами.

Стандартные функции работы с файлами

В среде программирования QNX существуют два основных интерфейса для работы с файлами:

1. Интерфейс системных вызовов, предлагающий системные функции низкого уровня, непосредственно взаимодействующие со средствами операционной системы.
2. Стандартная библиотека ввода/вывода, предлагающая функции буферизированного ввода/вывода.

Второй интерфейс является надстройкой над интерфейсом системных вызовов и предлагает более удобный (упрощенный) способ работы с файлами. Функции этого интерфейса определены стандартом ANSIязыка С как стандартная библиотека ввода/вывода. Поэтому использование этих функций обеспечивает программе наибольшую мобильность. В то же время они не обеспечивают всех возможностей по управлению вводом/выводом, предоставляемых операционной системой QNX, которые могут потребоваться при создании приложений реального времени. Выбор между функциями интерфейса системных вызовов и стандартной библиотеки зависит от необходимой степени контроля ввода/вывода и требованием переносимости программы.

Стандартные функции ввода/вывода обеспечивают так называемый *буферизированный ввод/вывод*. Для работы с файлом в системе создается указатель на специальную структуру данных (системного типа FILE), называемую *потоком*. При создании потока в оперативной памяти автоматически создаются специальные буферы, участвующие в процессе ввода/вывода данных и позволяющие минимизировать число системных вызовов ввода/вывода и, следовательно, обращений к физическому устройству. Структура типа FILEсодержит в себе:

- указатель на буфер;
- указатель позиции файла, подлежащий чтению или записи;
- число занятых байтов в буфере;
- флаги состояния потока.

Открытие файла

Для открытия файла используется функция

```
#include<stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Функция fopen открывает файл с именем filename, связывает его с потоком и возвращает указатель на поток.
Строка mode задает режим доступа к файлу и может принимать одно из следующих значений:

r - Открыть только для чтения.

w - Создать новый файл для записи. Если файл с таким именем уже существует, то он будет переписан.

a - Открыть существующий файл для записи только в конец файла или создать файл для записи, если файл не существует.

r+ - Открыть существующий файл для изменения (перезапись и чтение).

w+ - Создать новый файл для записи и чтения. Если файл с таким именем уже существует, то он будет переписан.

a+ - Открыть (или создать, если файл не существует) файл для чтения и записи только в конец файла.

Файл можно открыть в текстовом или двоичном режиме. Если содержимое файла текстовое и работа с файлом будет осуществляться исключительно как с текстом, то можно указать текстовый режим открытия файла. Для этого нужно добавить символ t к строке mode (rt, w+t, и т.д.). При открытии файла в текстовом режиме операционная система учитывает это таким образом, что может осуществлять фильтрацию некоторых управляющих символов при чтении данных из файла или включать определенные управляющие символы в файл при записи. Поэтому работа с содержимым файлов, открытых в текстовом режиме, должна осуществляться с использованием специальных функций чтения/записи символов или строк.

Если содержимое файла трактуется как набор байтов, то необходимо задать двоичный режим открытия. Для задания двоичного режима надо добавить символ b к строке mode (wb, a+b, и т.д.).

Можно помещать символы t и b между буквой и символом +. Например, использование rt+ эквивалентно r+t.

Если t и b не указаны в строке mode - режим определяется значением глобальной переменной _fmode. Если _fmode установлена в O_BINARY, то файлы открываются в двоичном режиме. Если _fmode установлена в O_TEXT - файлы открываются в текстовом режиме. Эти константы определены в файле fcntl.h.

Когда файл открывается для изменения, по отношению к нему могут быть произведены как ввод, так и вывод данных. Однако следует знать, что вывод не может сразу следовать за вводом без использования функции fseek() или rewind() (см. ниже), как и ввод не может сразу следовать за выводом без вмешательства функций fseek() или rewind(), или за вводом, который встречает конец файла.

При успешном завершении функция fopen возвращает значение указателя на вновь открытый поток. В случае ошибки - возвращает NULL.

При запуске программного модуля на выполнение (процесса) автоматически становятся доступными стандартные потоки ввода/вывода, определённые следующим образом:

```
externFILE *stdin;//стандартный поток ввода
externFILE *stdout;//стандартный поток вывода
externFILE *stderr;/*стандартный поток системных сообщений об ошибках*/
```

Открытие временного файла

Если файл необходим для временного хранения данных в период выполнения программы, то можно открыть временный файл без явного указания его имени. Для этого используется функция

```
#include <stdio.h>

FILE *tmpfile(void);
```

Функция tmpfile() создает и открывает для модификации временный двоичный файл в режиме "w+b". Этот файл автоматически удаляется при его закрытии или завершении работы программы.

Функция tmpfile() возвращает указатель на поток созданного временного файла. Если файл не может быть создан, то tmpfile()возвращает NULL.

Пример:

```
#include <stdio.h>
#include <process.h>

int main(void)
{
  FILE *tempfp;

  tempfp = tmpfile();
  if (tempfp)
    printf("Временныйфайлсоздан\n");
  else
  {
    printf("Невозможно открыть временный файл\n");
    exit(1);
  }

  return 0;
```

Переназначение потоков

Дескриптор потока открытого файла можно переназначить (связать) с другим файлом. Это позволяет перенаправить поток ввода/вывода с одного файла на другой. Дляэтогоиспользуетсяфункция:

```
#include<stdio.h>

FILE *freopen(const char *filename,
              const char *mode,
              FILE *stream);
```

Функция freopen() закрывает текущий файл, открывает файл, с именем, указанным в filename и связывает дескриптор потока streamc вновь открытым файлом. Эта функция закрывает исходный файл не зависимо от того, успешно ли состоялось открытие нового файла или нет. Строка mode, определяет режим открытия и доступа к новому файлу так же, как и в функции fopen().

При успешном завершении freopen() возвращает значение аргумента stream. В случае ошибки - возвращает NULL.

Функцию freopen() удобно использовать для переключения стандартных потоков stdin, stdout или stderr с файлов устройств на обычные файлы.

Пример:

```
#include <stdio.h>

int main(void)
{
  /* перенаправить стандартный вывод в файл */
  if (freopen("OUTPUT.FIL", "w", stdout) == NULL)
```

```
    fprintf(stderr, "ошибкапереадресации stdout\n");

  /* этот вывод будет осуществляться в файл */
  printf("Это будет выведено в файл.");

  /* закрыть стандартный поток вывода */
  fclose(stdout);

  return 0;
}
```

Доступ к файлу в текстовом режиме

Чтение символа из потока

Для чтения символа из потока используется семейство функций: fgetc(), fgetchar(), getc(), getch(), getchar(), getche(). Рассмотрим некоторые из них.

Функция

```
#include<stdio.h>
intfgetc (FILE *stream);
```

получает текущий символ из входного потока stream, перемещает указатель потока на следующий символ, возвращает код символа в виде положительного целого значения типа int или системную константу EOF, если обнаруживается конец файла или возникает ошибка ввода.

Для получения символа из стандартного потока stdin можно использовать функцию

```
intfgetchar(void);
```

Для ввода символа с клавиатуры без эхо-отображения на экране используется функция

```
#include<conio.h>
int getch(void);
```

Запись символа в поток

Для записи символа в поток используется семейство функций: fputc(), fputch(), fputchar(), putch(), putchar(). Рассмотрим некоторые из них.

Функция

```
#include<stdio.h>
intfputc (intch, FILE * stream);
```

выводит символ chв поток stream, перемещает указатель потока на следующую позицию и возвращает код этого символа в виде положительного целого значения типа int или системную константу EOF, если обнаруживается конец файла или возникает ошибка вывода.

Функция

```
int fputchar(int ch);
```

выводит символ ch в поток stdout. При успешном выполнении функция возвращает значение символа ch. При ошибке fputchar()возвращает EOF.

Чтение строки из потока

Для чтения строки из потока используется функция

```
#include<stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Функция fgets() читает символы из потока в строку s. Функциязаканчивает чтение, когда она либо прочтет n-1 символ, либо встретит символ \n, который заносит в конец строки s. Байт \0 добавляется к s для идентификации конца строки. Если функция выполняется со значением n равным 1, то в s формируется пустая строка. Функция при успешном выполнении возвращает значение указателя на строку, а при обнаружении ошибки или конца файла - NULL.

Запись строки в поток

Для записи строки в поток используется функция

```
#include<stdio.h>
int fputs(const char *s, FILE *stream);
```

Функция fputs() выводит в поток строкуs, ограниченную байтом \0, который в поток не выводится. При успешном выполнении функция возвращает значение последнего записанного символа строки, либо значение 0, если строка пуста. В противном случае - значение EOF.

Пример:

```
#include <stdio.h>

intmain(void)
{
  /* пишет строку в стандартный выходной поток */
  fputs("Здравствуй, мир\n", stdout);
  return 0;
}
```

Контроль события EOF

Ряд функций чтения/записи данных из файла могут возвращать значение EOF, которое означает либо событие конца файла, либо ошибку ввода/вывода. Для выяснения, какое из указанных событий реально возникло, служат функции:

```
#include<stdio.h>
int feof(FILE *stream);
intferror(FILE *stream);
```

Признак конца файла сбрасывается при каждой операции ввода/вывода. Функция feof() проверяет поток на наличие признака конца файла и возвращает ненулевое значение, если обнаружен признак конца файла во время выполнения последней операции ввода/вывода в потоке, или 0 - если не был обнаружен конец файла. Как только этот признак в потоке установлен, последующие выполнения функций чтения/записи файла возвращают значение этого признака до тех пор, пока не будет вызвана функция rewind() или файл не будет закрыт.

Функция ferror() проверяет поток на наличие признака ошибки чтения/записи и возвращает ненулевое значение, если признак обнаружен во время выполнения последней

операции ввода/вывода в потоке, или 0 - если признак не обнаружен. Если установлен признак ошибки потока stream, то он сохраняется до вызова функции

void clearerr(FILE *stream);

или

void rewind(FILE *stream);

или до закрытия потока. Функция clearerr() принудительно очищает признак ошибки потока. Функция rewind() перемещает указатель позиции файла в начало файла и очищает признак конца файла и признак ошибки в потоке.

Пример:

```
#include <stdio.h>

int main(void)
{
  FILE *stream;

  /* открыть файл для записи */
  stream = fopen("DUMMY.FIL", "w");

  /* попыткой чтения вызвать условие для ошибки */
  (void) getc(stream);
  if (ferror(stream))    /* проверка наличия ошибки в потоке */
  {
    /* вывести сообщение об ошибке */
    printf("Ошибка при чтении DUMMY.FIL\n");

    /* сбросить признак ошибки и конца файла EOF */
    clearerr(stream);
  }
  fclose(stream);
  return 0;
}
```

Форматный доступ к файлу в текстовом режиме

Форматные преобразования и запись данных в поток

#include<stdio.h>
int fprintf(FILE *stream,const char *format[,аргумент,...]);

Функция fprintf() позволяет использовать переменное число аргументов. Функция выполняет вывод данных в поток stream. Формат вывода (шаблон представления) для каждого аргумента задается в строке формата, определяемой указателем format. Число задаваемых в строке формата спецификаций форматов должно совпадать с количеством аргументов или быть меньше. Если их будет больше, то результат будет непредсказуемым (возможно, катастрофическим). Лишние аргументы (если их число больше, чем спецификаций формата) просто игнорируются.

Функция fprintf() принимает набор аргументов, применяет к каждому из аргументов соответствующую спецификацию формата из строки формата format и записывает символьное представление значения аргумента в поток stream. Форматная строка управляет

преобразованием, форматированием и записью символьных представлений значений аргументов в поток.

Форматная строка - это строка символов, содержащая два типа объектов - простые символы и спецификации форматных преобразований. Простые символы записываются в выходной поток без изменений. Спецификации форматных преобразований применяются для форматирования значений аргументов из списка.

Спецификаторы формата имеют следующую форму:

%[флаги][ширина][.точность][модификатор_размера]type

Каждая спецификация преобразования начинается с символа процента <%>, после чего следуют признаки в приведенном ниже порядке:

флаги - необязательная последовательность символов флагов;

ширина - необязательный спецификатор ширины;

точность - необязательный спецификатор точности;

модификатор_размера - необязательный модификатор принимаемого размера;

type- форматный код преобразования.

Ниже приводятся основные правила управления форматом вывода, включая необязательные символы, спецификаторы и модификаторы форматной строки.

Флаги управляют выравниванием выводимых символов, выводом знаковых символов ('+' и '-') числа, пробелов, десятичной точкой, восьмеричными и шестнадцатеричными префиксами.

Ширина – целое число, которое определяет минимальное число выводимых символов, с учетом пробелов и нулей.

Точность – целое число, которое определяет максимальное число выводимых символов; или минимальное число выводимых цифр для целых чисел. Если не задана, то устанавливается по умолчанию.

Модификатор размера, принимающий значения h, **l** и L. Используется как префикс с форматными кодами *type*. Модификатор влияет на то, как функция интерпретирует тип соответствующего аргумента. Он заменяет тип аргумента по умолчанию для заданного кода типа на другой, соответствующий модификатору, следующим образом:

h– Используется с кодами типов d, i, o, u, хи X. Определяет, что аргумент является shortint.

l - Используется с кодами типов d, i, o, u, хи X. Определяет, что аргумент является longint; также используется с кодами типов c, e, f, gили G, чтобы показать, что аргументы имеют тип double, а не float.

L - Используется с кодами типов e, E, f, g или G. Аргумент интерпретируется как long double.

Форматный код *type* специфицирует с помощью односимвольного кода тип аргумента и способ символьного представления значения аргумента в поле файла. В следующей таблице приводится список кодов, соответствующие им типы аргументов и представление значения аргумента в поле файла после преобразования. Приводимая информация основана на предположении, что в спецификацию формата не включены флаговые символы.

Таблица

Код типа	Тип аргумента	Представление значения аргумента в поле файла
d	int	Десятичное целое со знаком
i	int	Десятичное целое со знаком
o	int	Беззнаковое восьмеричное целое
u	int	Беззнаковое десятичное целое
x	int	Беззнаковое шестнадцатиричное целое с цифрами a,b,c,d,e,f

X	int	Беззнаковое шестнадцатиричное целое с цифрами A,B,C,D,E,F
f	double	Число с фиксированной точкой в форме [-]ddd.dddddd, где число цифр после десятичной точки соответствует заданной в формате точности
e	double	Число с плавающей точкой в форме [-]d.ddde[+/-]ddd, где десятичной точке предшествует одна цифра; количество цифр после десятичной точки соответствует заданной в формате точности; степень всегда содержит по крайней мере 2 разряда
E	double	Аналогично e, но с E для экспоненты ([-]d.dddE[+/-]ddd)
g	double	Значение со знаком в форме e, Еили f, в зависимости от заданного в формате значения и точности. Конечные нули и десятичная точка выводятся только в случае необходимости
G	double	Аналогично g, но с G для экспоненты в случае использования формата e.
c	char	Одиночный символ
s	char *	Последовательность символов до признака конца строки '\0', либо до достижения количества символов, равного явно заданному в формате значению точности
%	нет	Выводится символ %
n	int *	в ячейке, на которую указывает аргумент, сохраняется число, означающее количество символов, выведенных в выводной поток к данному моменту выполнения функции
p	Указатель (type *)	Значение аргумента выводится как значение указателя void*, представленное в виде 16-ричного целого

К символам-флагам относятся: минус(-), плюс(+), диез(#) и пробел ( ). Их можно указывать в любом порядке; допустимы комбинации флагов. Влияние флагов на представление значения аргумента в поле файла следующее:

(-) - Выравнивание результата по левому краю поля и заполнение правого края пробелами. Если не задан, то осуществляется выравнивание справа, и левый край заполняется пробелами или нулями.

(+) - Результаты знакового преобразования всегда начинаются со знаков плюс(+) или минус(-). флаг плюс (+) имеет приоритет над пробелом при одновременном их использовании.

пробел - Если значение неотрицательно, то вывод начинается с пробела вместо знака плюс; отрицательные значения - всегда со знака минус.

(#) - Определяет, что аргумент преобразуется с использованием "альтернативной формы".

Альтернативные формы

Флаг # влияет на представление значения аргумента для заданного кода типа следующим образом:

c,s,d,i,u - Не влияет.

o - Ненулевое значение аргумента будет начинаться с нуля.

x или X - Значение аргумента будет начинаться с 0x (или 0X).

e, E или f - Результат всегда будет содержать десятичную точку, даже если за ней не следуют цифры. Обычно десятичная точка появляется только в случае, когда после нее следуют цифры.

g или G - Аналогично e и E, но конечные нули не удаляются.

Спецификатор ширины устанавливает минимальную ширину поля для выводимого значения. Ширина определяется одним из двух способов: непосредственно с помощью

строки десятичных цифр, или косвенно - с помощью знака (*). Если используется звездочка в качестве спецификатора ширины, то следующий аргумент (который должен быть целым) определяет минимальную ширину выводимого поля.

Отсутствие спецификации ширины или его недостаточное значение не вызывают усечение выводимого значения. Если результат преобразования больше, чем позволяет ширина поля, то поле просто расширяется до размеров результата.

Спецификатор точности влияет на точность вывода значения аргумента. Если точность явно не указана, то по умолчанию установлена следующая точность: 1 - для d, i, o, u, x, X типов; 6 - для e, E, f типов; все значащие цифры – для g и G типов; вывод до первого нулевого байта '\0' - для типов s; не влияет - на тип c. Если указана точность 0, то для типов d, i, o, u, x точность устанавливается по умолчанию. Для типов e, E, f не печатается десятичная точка. Если явно задана точность N, то выводится N символов строки или N десятичных знаков числа. Если выводимое значение содержит свыше N символов, то оно может быть усечено или округлено (случится это или нет, зависит от кода типа). Если в качестве точности используется символ звездочка (*), то спецификатор точности задается как аргумент в списке аргументов, причем он предшествует форматируемому аргументу. Если явно указана нулевая точность и для вывода задан один из целочисленных форматов (т.е. d, i, o, u, x), а выводимое значение равно нулю, то ни одной цифры не будет выведено в этом поле (т.е. поле будет представлено пробелом).

Функция fprintf() возвращает количество выведенных байтов. В случае ошибки возвращается EOF.

Замечания:

Частным вариантом функции fprintf() для вывода в стандартный поток вывода stdout является функция:

int printf(const char *format[,аргумент,...]);

Кроме того, иногда требуется выполнить форматное преобразование данного в символьное представление, а результат вместо файла поместить в буфер памяти. Для этого можно использовать функцию:

intsprintf(constchar *buf,constchar *format [,аргумент,...]);

Пример 1

```
#include <stdio.h>

int main(void)
{
  FILE *stream;
  int i = 100;
  char c = 'C';
  float f = 1.234;

  /* открыть файла для обновления */
  stream = fopen("DUMMY.FIL", "w+");

  /* записать некоторые данные в этот файл */
  fprintf(stream, "%d %c %f", i, c, f);

  /* закрытьфайл */
  fclose(stream);
  return 0;
```

```
}
```

Пример 2

```
/* Программа для создания дубликата файла my_file.dat */

#include<stdio.h>

int main(void)
{
  FILE *in, *out;

  if ((in = fopen("/houme/my_file.dat", "rt")) == NULL);
  {
    fprintf(stderr, "Невозможно открыть входной файл.\n");
    return 1;
  }
  if ((out = fopen("/houme/my_file.bac", "wt")) == NULL);
  {
    fprintf(stderr, "Невозможно открыть выходной файл.\n");
    return 1;
  }
  while (!feof(in))
    fputc(fgetc(in), out);
  fclose(in);
  fclose(out);
  return 0;
}
```

Ввод и форматное преобразование данных из потока

```
#include<stdio.h>

int fscanf(FILE *stream, const char *format[,address,...]);
```

Функция fscanf() посимвольно сканирует набор вводимых полей, считывая их из потокаstream. Каждое введённое из потока поле форматируется в соответствии со спецификацией формата, которая передается в виде указателя на форматную строку format. Преобразованные данные присваиваются переменным, адреса которых передаются в качестве аргументов функции, следующих после аргумента format. Количество спецификаций формата должно совпадать с количеством адресов, переданных функции.

Форматная строка управляет вводом, преобразованием и запоминанием данных из полей ввода. При этом для заданных спецификаций формата должно быть достаточное количество адресных аргументов, в противном случае результат работы функции непредсказуем и может привести к катастрофическим последствиям. Лишние адресные аргументы (которым нет соответствующих форматов) просто игнорируются. Форматная строка является символьной строкой, которая содержит три типа объектов: пробельные символы, отображаемые символы и спецификации формата.

Пробельными символами являются пробел ( ), символ табуляции (\t) и символ перехода на новую строку (\n). Если функция fscanf()встречает пробельный символ в форматной строке, она будет считывать, но не сохранять всю последовательность символов вплоть до следующего отображаемого символа во входном потоке.

Отображаемыми символами являются все другие символы кода ASCII, за исключением символа процента (%). Если функция fscanf() встречает в строке форматов

отображаемый символ, то она ожидает такой же символ в потоке, прочитает соответствующий ему символ из потока, но не выполняет никаких присвоений.

Спецификации формата предписывают функции fscanf() осуществить чтение и преобразование символов из входного поля в значения определенного типа, затем запомнить их по адресу, указанному соответствующим адресным аргументом.

Завершающий (последний) пробельный символ не читается (включая символ перехода на новую строку), если только он не описан явно в форматной строке.

Спецификации формата функции fscanf() имеют следующий вид:

% [*] [widht] [type_length] <форматный_код>

Спецификация каждого формата начинается с символа процента <%>. После этого символа следуют символы спецификации в следующем порядке:

* - необязательный символ подавления назначения;

width - необязательная спецификация ширины поля;

type_length - необязательный модификатор длины типа аргумента;

<форматный код> - символ кода форматного преобразования.

Символ спецификации <*> отменяет присваивание следующего поля ввода. Спецификация ширины поля <width> задаёт максимальное число считываемых символов. Меньшее количество символов может быть считано в случае, если функция fscanf() встретит пробельный или непреобразуемый символ.

Форматные коды. Следующая таблица содержит список форматных кодов, типы переменных, указываемых аргументами, и вид представления вводимых значений в полях файла.

Таблица

Форматный код	Тип аргумента	Представление значения в поле файла
d	int*	Десятичное целое
o	int*	Восьмеричное целое со знаком
i	int*	Десятичное, восьмеричное или шестнадцатеричное целое со знаком
u	unsigned int*	Десятичное целое беззнака
x, X	int*	Шестнадцатиричное целое
a,A,e,E, f,F,g,G	float*	Число с плавающей точкой [+/-] ddddddddd [.] dddd [E|e] [+/-] ddd
s	char*,unsigned char*	Последовательность непробельных символов до достижения количества символов, равного заданной длине поля widht, или до первого пробельного символа, если длина явно не указана. Кроме того автоматически добавляется признак конца строки - '\0'.
c	char*	Любая последовательность символов в количестве widht или одиночный символ, если длина не указана. Пробельные (пробельные) символы не пропускаются. Чтобы прочитать следующий отображаемый символ, следует использовать спецификацию %1s.
%	нет	Вводится символ %, преобразования не осуществляется, символ % сохраняется.
n	int *	Ввода данных не происходит, а в переменную, на которую указывает соответствующий аргумент,

		заносится целое значение, равное количеству ранее успешно считанных символов.
p	Указатель (type *)	Шестнадцатеричное целое, соответствующее рассмотренному выше типу x, которое трактуется как значение указателя неопределенного типа void*. Значение присваивается преременной, на которую указывает аргумент.

Полями ввода могут быть:

- все символы до следующего пробельного символа (не включая его);
- все символы до первого непреобразуемого по указанной спецификации формата символа (например, числа "8" и "9" по восьмеричному формату);
- последовательность символов, в количестве, определяемом значением width.

Для некоторых спецификаций формата приняты определенные соглашения, которые приведены ниже:

%c - по этой спецификации читается следующий символ, включая пробельный символ. Для пропуска пробельных символов и чтения одного отображаемого символа следует использовать спецификацию %1s.

%<width>c - преобразование предполагает, что аргумент является указателем на массив символов; массив состоит не менее чем из widthэлементов (char arg[width]).

%s – преобразование предполагает, что аргумент является указателем на массив символов (char arg[]). Размер массива должен быть не менее (n+1) байт, где n - длина строки s (в символах). Поле ввода завершается пробельным символом или символом перехода на новую строку <\n>. Ограничитель строки <\0> автоматически добавляется в строку после считывания и хранится в последнем элементе массива.

Поле ввода является строкой, не разделенной пробельными символами. Функция fscanf()считывает соответствующее поле ввода до первого символа, не являющегося элементом допустимого для текущей спецификации формата набора символов.

Символ <*> является *символом подавления присваивания* при вводе значений с помощью функции fscanf(). Если этот символ следует в спецификации формата за символом <%>, то следующее входное поле будет считано, но не присвоено переменной, адресуемой следующим аргументом. Предполагается, что подавляемые входные данные специфицируются форматным кодом, который следует за символом <*>.

Спецификатор ширины<widht>, десятичное целое число, задает максимальное количество символов, которые будут считаны из текущего поля ввода. Если входное поле содержит менее, чем <widht> символов, функция считывает вначале в этом поле все символы, а затем обрабатывает следующее поле и спецификацию формата. Если пробельный или непреобразуемый символ встретился в пределах указанной ширины поля ввода, то функция считывает, преобразует и размещает по указанному адресу символы, находящиеся до пробельного или непреобразуемого символа, после чего функция обратится к следующей спецификации формата.

Непреобразуемыми считаются те символы, которые не могут быть преобразованы в соответствии с указанной спецификацией (такие как, например, символы "8" или "9" для восьмеричного формата и "J" либо "K", если указан шестнадцатиричный или десятичный формат).

Модификаторы длины типа аргументов позволяют уточнить длину типа переменных, указываемых аргументами, которым присваиваются вводимые значения. Могут использоваться следующие одно- или двухсимвольные модификаторы:

hh- для форматных кодовd, i, o, u, x, X или n означает требование преобразовать в значение типа signedcharили unsignedchar.

h- для форматных кодовd, i, o, u, x, X или n означает требование преобразовать в значение типа short или unsignedshort.

l- для форматных кодовd, i, o, u, x, X или n означает требование преобразовать в значение типа longили unsignedlong. Для форматных кодов a, A, e, E, f, F, g или G означает требование преобразовать в значение типа double.

L– для форматных кодов a, A, e, E, f, F, g или G означает требование преобразовать в значение типа longdouble.

z– для форматных кодов a, A, e, E, f, F, g или G означает требование преобразовать в значение типа size_t

Функция fscanf() возвращает количество аргументов, для которых было успешно выполнено присвоение введённых значений. В случае ошибки или достижения конца файла возвращается EOF.

Замечания:

Частным вариантом функции fscanf() для ввода из стандартного потока ввода stdin является функция:

int scanf(const char *format[,аргумент,...]);

Кроме того, иногда необходимо выполнить форматное преобразование данного, представленного в символьном виде, но находящегося не в файле , а в буфере памяти. Для этого можно использовать функцию:

intsprintf(constchar *buf,constchar *format [,аргумент,...]);

Пример1:

```
…
shortval;
…

/* Из стандартного потока ввода (буфер клавиатуры) ввести поле длиной не более 3-х
символов и преобразовать его в целое значение типа shortint */

fscanf(stdin, "%3hd", &val);
```

Пример2:

```
…
charArr[81];
…
/* Из стандартного потока ввода ввести строку символов до пробельного символа, длиной
не более 80 символов*/

scanf("%s", Arr);
…
```

Пример3:

```
…
int a, b, c;
…
/* Из стандартного потока ввода ввести три поля, длиной соответственно 6, 6 и 3
символа, содержащих три целых значения*/
```

```
scanf("%6d%6d%3d", &a, &b, &c);
```

Пример4:

```
…
int group;
char Initial;
char Famil[20];
…
        scanf("%s %c%*s №гр %d%*s", Famil, &Initial, &group);

        printf("%s%c. №гр %d", Famil, &Initial, &group);

…
```

Если в четвёртом примере положить, что в буфер клавиатуры введена последовательность символов <Иванов Пётр №гр 6405 очно>, то результат печати будет:

Иванов П. №гр 6405

Задачи:

1. Открыть временный файл.
2. Создать файл и написать текст в файл. (минимум 3 примера).
3. Написать программу которая выполняет следующие задачи:
    a. Создать файл,
    b. Запись строку в файл,
    c. Чтение строки из файла.

ПРАКТИЧЕСКАЯ РАБОТА №3. РАБОТА С ПРОЦЕССАМИ.

Цель работы: Научаиться работать с процессами, научиться использовать функции библиотеки process.h.

Процесс — это выполняющаяся программа. Грубо говоря, процесс состоит из образа процесса и метаданных процесса. *Образом процесса* будем называть совокупность кода (т. е. инструкций для процессора — выполнение этих инструкций и есть выполнение программы) и данных (как раз ими манипулируют с помощью инструкций). *Метаданные процесса* — это информация о процессе, которая хранится в структурах данных операционной системы и сопровождается операционной системой. Метаданными является, например, информация о физическом размещении кода и данных в оперативной памяти, а также атрибуты процесса, к которым относятся:

- идентификатор процесса (Process ID, PID) — уникальный номер, присваиваемый процессу при его порождении операционной системой;
- идентификатор родительского процесса (Parent PID, PPID) — PID процесса, породившего данный процесс, т. е. выполнившего запрос к операционной системе для создания данного процесса;
- реальные идентификаторы владельца и группы (User ID, UID и Group ID, GID) — номер, позволяющий механизмам защиты информации от несанкционированного доступа (НСД) определять, какому пользователю принадлежит процесс и к какой группе пользователей принадлежит этот пользователь. Эти идентификаторы присваиваются при регистрации пользователя в системе или командному интерпретатору (login shell), если выполнялась командно-стоковая регистрация через утилиту **login**, или графической оболочке Photon, если регистрация выполнялась в графическом режиме через утилиту **phlogin**. Процессы, запускаемые пользователем, наследуют UID и GID той программы, из которой они запускаются (т. е. *родительского процесса*);
- эффективные идентификаторы владельца и группы (Effective UID, EUID и Effective GID, EGID) — предназначены для повышения гибкости механизмов защиты информации от НСД. Пользователь при наличии соответствующих полномочийможет в ходе работы менять эффективные идентификаторы. При этом реальные идентификаторы не меняются. Механизмы, реализующие дискреционную защиту информации от НСД, для проверки прав доступа используют эффективные идентификаторы;
- текущий рабочий каталог — путь (разделенный слэшами список каталогов), который будет автоматически добавляться к относительным именам файлов. Выводится на экран командой **pwd**;
- управляющий терминал — терминал, с которым связаны потоки ввода, вывода и ошибок;
- маска создания файлов (umask) — атрибуты доступа, которые будут заданы для файла, созданного процессом;
- значение приоритета;
- дисциплина диспетчеризации.
- использование ресурсов процессора (статистика по времени выполнения программы) — включает: время выполнения программы в прикладном контексте (user time — время выполнения инструкций, написанных программистом), время выполнения в контексте ядра (system time — время выполнения инструкций ядра по запросу программы, т. е. системных вызовов), суммарное время выполнения всех дочерних процессов в прикладном контексте, суммарное время выполнения всех дочерних процессов в контексте ядра.

ПОЛУЧЕНИЕ МЕТА ДАННЫЕ ПРОЦЕССА

Программа начинает выполняться с *точки входа* — функции *main()*.

Проиллюстрируем, каким образом программа может получить информацию о значении своих атрибутов (файл process.c):

```
#include<stdlib.h>
#include <sys/resource.h>
int main(int argc, char **argv, char **env)
{
struct rusage r_usage;
printf("\nProcess Informaition:\n");
printf("Process name = \t\t%s\n", argv[0]);
printf("User ID = \t\t< %d >\n", getuid());
printf("Effective User ID = \t< %d >\n", geteuid());
printf("Group ID = \t\t< %d >\n", getgid());
printf("Effective Group ID = \t< %d >\n", getegid());
printf("Process Group ID = \t< %d >\n", getpgid(0));
printf("Process ID ( PID )= \t< %d >\n", getpid(0));
printf("Parent PID ( PPID )= \t< %d >\n", getppid(0));
printf("Process priority= \t< %d >\n", getprio(0));
printf("Processor utilisation:\n");
getrusage (RUSAGE_SELF, &r_usage);
printf("\t< user time = %d sec, %d microsec >\n",
r_usage.ru_utime.tv_sec, r_usage.ru_utime.tv_usec);
printf("\t< system time = %d sec, %d microsec >\n",
r_usage.ru_stime.tv_sec, r_usage.ru_stime.tv_usec);
return EXIT_SUCCESS;
}
```

Жизненный цикл процесса можно разделить на четыре этапа:

- создание процесса;
- загрузка образа процесса;
- выполнение процесса;
- завершение процесса.

СОЗДАНИЕ ПРОЦЕССА

"Предком" всех процессов является процесс *Администратор процессов* (процесс **procnto**), идентификатор PID которого равен 1. Остальные процессы порождаются в результате вызова соответствующей функции другим процессом, именуемым *родительским*. Таких функций несколько:

- семейство функций *exec()* — отличаются набором аргументов функций, заменяют образ вызвавшего процесса указанным исполняемым файлом;
- *fork()* — создает дочерний процесс путем "клонирования" родительского процесса;

Примечание

Долгое время в UNIX-подобных системах новые процессы порождались путем вызова сначала функции *fork()*, а затем из дочернего процесса-клона — функции семейства *exec()*. Рекомендуется использовать *fork()* только в однопоточных программах.

- *vfork()* ("виртуальный *fork()*") — используется как "облегченная" альтернатива паре вызовов *fork()-exec()*. В отличие от стандартной функции *fork()*, она не выполняет реального копирования данных, а просто блокирует родительский процесс, пока дочерний не вызовет *exec()*;
- семейство функций *spawn*()* — сразу порождает дочерний процесс, загрузив указанный исполняемый файл. Наиболее эффективный способ порождения процессов в QNX Neutrino;
- *system()* — порождает shell для выполнения командной строки, указанной в качестве аргумента.

Посмотрим, как применяют функцию *fork()* на примере файла fork.c. В этом примере видно, что функция *fork()* возвращает целое число, которое в родительском процессе равно нулю, а в дочернем — идентификатору процесса:

```
#include<stdlib.h>
#include <sys/resource.h>
int main(int argc, char **argv, char **env)
{
pid_t pid;
char *Prefix;
Prefix = (char *) malloc (sizeof(char));
pid = fork();
if (pid == 0) sprintf ( Prefix, "CHILD:");
else sprintf ( Prefix, "PARENT:");
printf("%s Process name = %s\n", Prefix, argv[0]);
printf("%s PID = %d\n", Prefix, getpid(0));
printf("%s PPID = %d\n", Prefix, getppid(0));
return EXIT_SUCCESS;
}
```

Примечание
На всякий случай напомню — применение функции *fork()* в многопоточных приложениях не рекомендуется.
На примере exec.c проиллюстрируем порождение нового процесса с помощью комбинации вызовов *vfork()* и *exec()*:

```
#include<stdlib.h>
#include <sys/resource.h>
int main(int argc, char **argv, char **env)
{
pid_t pid;
pid = vfork();
if (pid == 0)
{
execlp("process", "process", NULL);
perror("Child");
exit( EXIT_FAILURE);
}
waitpid(0, NULL, 0);
printf("Parants's PID = %d\n", getpid(0));
printf("Parants's PPID = %d\n", getppid(0));
return EXIT_SUCCESS;
```

```
}
```

В файле spawn.c представлен пример наиболее простого и быстрого способа порождения нового процесса:

```
#include<stdlib.h>
#include <process.h>
int main(int argc, char **argv, char **env)
{
spawnl ( P_WAIT, "process", "process", NULL );
printf("Parants's PID = %d\n", getpid());
printf("Parants's PPID = %d\n", getppid());
return EXIT_SUCCESS;
}
```

Загрузка процесса выполняется операционной системой (Администратором процессов) и заключается в помещении сегментов кода и данных исполняемого файла в оперативную память ЭВМ. После загрузки начинается выполнение функции *main()*.

ЗАДАНИЕ

1. Написать программу для получение мета данных процесса.
2. Создать новый процесс с помощью fork().
3. Создать новый процесс с помощью vfork().
4. Создать новый процесс с помощью spawn().

ПРАКТИЧЕСКАЯ РАБОТА №4. РАБОТА С БИБЛИОТЕКАМИ.

Цель работы: Научаиться работать с статических и разделяемых библиотеками.

Общие сведения о библиотеках

Почти во всех рассмотренных примерах для вывода сообщений на экран использовалась функция *pirntf()*. Эта функция объявлена в заголовочном файле stdio.h, а ее объектный код содержится в стандартной библиотеке libc.a. При компоновке объектный код функции *printf()* вставляется в исполняемый файл и становится его неотъемлемой частью. Такие библиотеки называют статическими. *Статическая библиотека* представляет собой обычный архив о- файлов. Создается статическая библиотека с помощью утилиты **ar**. Для удобства имена файлов статических библиотек начинаются с lib и имеют расширение a (например, libsocket.a).

Существуют также разделяемые библиотеки. *Разделяемая библиотека* представляет собой файл того же формата, что и приложение (ELF, Executable and Linking Format), но, в отличие от приложения, не имеет функции *main()*. Имена файлов разделяемых библиотек начинаются с lib и имеют расширение so (от англ. Shared Object — разделяемый объект), после которого может стоять номер версии (например, libmy.so.2).

Разделяемые объекты предоставляют следующие преимущества:

- экономия дискового пространства и оперативной памяти;
- возможность обновления программного обеспечения путем замены разделяемых объектов, без перекомпиляции использующих их программ;
- сокращение времени загрузки запускаемой программы.

Недостатки разделяемых объектов:

- требуется дополнительное время при первом вызове разделяемого объекта для загрузки его в оперативную память;
- непреднамеренное удаление разделяемого объекта приводит к неработоспособности всех использующих его программ.

Разделяемая библиотека должна быть обязательно загружена либо до запуска приложения, либо при запуске. Однако можно загружать и выгружать разделяемые объекты с помощью специальных функций *dlopen()* и *dlclose()* по мере надобности. Такие разделяемые объекты в документации QSS, удобства ради, называются принятым в Windows термином DLL (Dynamic Link Library — библиотека динамической компоновки). Другими словами, DLL отличается от Shared Library не физической реализацией, а только способом использования.

Создание статической библиотеки

Для иллюстрации создания библиотек будем использовать тот же проект, который рассматривался при обсуждении утилиты **make.** Этот "проект" состоит из файлов defs.h, main.c, aaa.c и bbb.c.
Напишем Make-файл static.mf для создания статической библиотеки:

```
OBJS = aaa.o bbb.o ccc.o
.PHONY=clean
libmy.a: clean $(OBJS)
ar -q libmy.a $(OBJS)
$(OBJS): defs.h
clean:
$(RM) *.o libmy.a
```

Создадим статическую библиотеку libmy.a, выполнив команду:

make –f static.mf

При этом сначала будет сгенерировано три объектных файла aaa.o, bbb.o и ccc.o, из которых с помощью утилиты **ar** будет создана статическая библиотека libmy.a. Теперь библиотеку можно использовать для создания других программ:

qcc –o my_static –L ./ main.c -lmy

С помощью опции **–l** мы указали компоновщику, в какой библиотеке искать нужные объектные модули, а с помощью опции **–L** — где искать статическую библиотеки кроме стандартных путей.

Примечание
Обратите внимание, что при указании опции **–l** мы отбросили префикс lib и постфикс .a. Компоновщик сам добавит их к указанному нами фрагменту имени my.

Создание разделяемой библиотеки
Теперь напишем Make-файл shared.mf для создания разделяемой библиотеки:

```
OBJS = aaa.o bbb.o ccc.o
CFLAGS+=-shared
.PHONY=clean
libmy.so: $(OBJS)
$(CC) -o $@ -shared $(OBJS)
$(OBJS): defs.h
clean:
$(RM) *.o libmy.so
```

Создадим разделяемую библиотеку libmy.so, предварительно удалив старые объектные файлы:

make –f shared.mf clean
make –f shared.mf

Сначала будет сгенерировано три объектных файла aaa.o, bbb.o и ccc.o, при этом компиляция этих файлов будет выполнятся с опцией **-shared.** Затем в результате компоновки из объектных файлов будет создана разделяемая библиотека libmy.so. Теперь библиотеку можно использовать для создания других программ:

qcc –o my_shared –L ./ main.c -Bdynamic –lmy

С помощью опции **–l** мы указали компоновщику, в какой библиотеке искать нужные объектные модули, а с помощью опции **–L** — в каких каталогах искать библиотеки в дополнение к стандартным путям поиска. Перед именем библиотеки мы опцией **-Bdynamic** указали компоновщику, что эта библиотека — разделяемая.
Примечание
Обратите внимание, что при указании опции **–l** мы отбросили префикс lib и постфикс .so. Компоновщик сам добавит их к указанному нами фрагменту имени my.

Чтобы можно было запускать программу **my_shared**, необходимо, чтобы динамический компоновщик знал, где искать разделяемую библиотеку libmy.so. Поиск выполняется в следующем порядке:

1. Динамический компоновщик ищет разделяемый объект в переменной окружения LD_LIBRARY_PATH программы, желающей загрузить этот объект.
2. Динамический компоновщик просматривает содержимое тега DT_RPATH динамической секции программы.
3. Динамический компоновщик ищет разделяемый объект в переменной окружения LD_LIBRARY_PATH Администратора процессов операционной системы QNX Neutrino.

Итак, запустим программу, добавив в переменную LD_LIBRARY_PATH текущий каталог, т. е. каталог ".":

LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
./my_shared

Теперь выполним команду **ls -l** и обратим внимание на размеры файлов my_shared и libmy.so. Сравним их с размерами файлов my_static и libmy.a.

Примечание

Для того чтобы жестко прописать в самом приложении путь поиска разделяемых объектов, надо заполнить тег DT_RPATH. Значение этого тега задается с помощью опции **–rpath** компоновщика **ld**. Чтобы задать эту опцию для компоновщика при компиляции с использованием утилиты **qcc**, как это и рекомендуется делать, нужно указать для **qcc** опцию **–Wl,-rpath** ***путь_к_разд_объекту***. То есть команда компиляции выглядела бы так:

**qcc –o my_shared –L ./ main.c -Bdynamic **
–lmy –Wl,-rpath ./

Использование DLL

Напишем простую программу dll.c, использующую функцию *aaa()* из разделяемой библиотеки libmy.so:

```
#include<dlfcn.h>
#include"defs.h"
int main()
{
void *dll;
void (*my_funk)();
printf("I'm main\n");
dll = dlopen("libmy.so", RTLD_NOW);
if (!dll)
{
perror("dll");
exit(1);
}
my_funk=dlsym(dll, "aaa");
(*my_funk)();
return 0;
}
```

Эта программа работает следующим образом. Сначала с помощью функции *dlopen()* загружаем разделяемый объект libmy.so. Затем с помощью функции *dlsym()* определяем

адрес нужной нам функции *aaa()* внутри модуля libmy.so. И, наконец, по полученному адресу вызываем функцию *aaa()*.

ЗАДАНИЕ

Написать программу для по ниже перечисленных требованием:

5. Создание статической библиотеки.
6. Создание разделяемой библотеки.
7. Написать программу использующую функции из разделяемой библиотеки.

ПРАКТИЧЕСКАЯ РАБОТА №5. POSIX-МЕХАНИЗМЫ ВЗАИМОДЕЙСТВИЯ МЕЖДУ ПРОЦЕССАМИ

Цель работы: Научиться работать механизмами взаимодействия между процессами, такие как разделямая память, очередь сообщений.

Процессы операционной системы QNX Neutrino, в каких бы "родственных" отношениях они ни находились, выполняются каждый в своем изолированном адресном пространстве и не могут обмениваться информацией без использования механизмов *межпроцессного взаимодействия* (IPC, InterProcess Communication). В QNX Neutrino реализован ряд таких механизмов, как стандартных, так и уникальных.
В этой главе мы рассмотрим некоторые из стандартных механизмов (читайте — POSIX-механизмов) IPC:

- именованные и неименованные программные каналы;
- разделяемая память;
- очереди сообщений POSIX.

Кроме того, мы поговорим об именованных и неименованных семафорах.

Программные каналы

Программные каналы достаточно широко используются для обмена данными между процессами. В нашем распоряжении два типа программных каналов — неименованные и именованные. Каждый из этих типов обладает особенностями, которые следует принимать во внимание при организации межпроцессного взаимодействия.

Неименованные программные каналы

Неименованный программный канал — это механизм, который командный интерпретатор использует для создания конвейеров *(см. главу 1)*. Этот механизм реализован в администраторе ресурсов **pipe** *(см. главу 3)* и обеспечивает передачу данных через промежуточный буфер в оперативной памяти (в адресном пространстве администратора **pipe**). Неименованный канал создается функцией *pipe()*, которой в качестве аргумента передается массив из двух целых чисел. В этот массив записывается два файловых дескриптора, один из которых в последующем используется для записи информации, другой — для чтения.

К достоинствам неименованных программных каналов можно отнести то, что они являются стандартным POSIX-механизмом IPC, а также то, что это — достаточно быстрый механизм. К недостаткам — то, что обмен данными возможен только между процессами-"родственниками", т. к. процесс может получить файловые дескрипторы для работы с неименованным каналом, только наследуя их от родительского процесса.

Именованные программные каналы

Именованные каналы — это POSIX-механизм, поддержка которого реализована в файловой системе Neutrino. В основе механизма лежит особый тип файла — FIFO *(см. главу 4)*, выполняющего функцию буфера для данных. Файл типа FIFO можно создать двумя способами:

- из командной строки — утилитой **mkfifo**;
- из программы — функцией *mkfifo()*.

Поскольку FIFO — это файл, имеющий имя, то, во-первых, работа с ним выполняется практически так же, как с обычным файлом (*open(), read(), write(), close()* и т. п.), во-вторых, именованный канал медленнее неименованного, но данные, записанные в именованный канал, сохраняются в случае сбоя (например, отключения питания).

Разделяемая память

Разделяемая память — чрезвычайно важный и широко используемый POSIX-механизм обмена данными большого объема (впрочем, не обязательно большого) между процессами. Использовать этот механизм можно, к примеру, выполнив на стороне каждого из взаимодействующих процессов такие действия:

1. С помощью функции *shm_open()* создается или открывается существующий регион памяти, который будет разделяться.
2. С помощью функции *shm_ctl()* задаются нужные атрибуты разделяемого региона.
3. С помощью функции *mmap()* на разделяемый регион памяти отображается фрагмент адресного пространства процесса, после чего с этим фрагментом выполняются необходимые операции (например, чтения или записи данных).
4. Для отмены отображения адресного пространства процесса на разделяемый регион используется функция *unmap()*, для закрытия разделяемого региона — функция *shm_close()*, для уничтожения — функция *shm_unlink()*.

Рассмотрим пример из двух программ — **shm_creator** и **shm_user**.
Пример работает так:

1. Запускается программа **shm_creator**, которая создает регион разделяемой памяти, задает его параметры и отображает на него некий буфер, содержащий текстовую строку.
2. Запускается программа **shm_user**, которая отображает регион разделяемой памяти, созданный программой **shm_creator**, на свой буфер и печатает содержимое этого буфера.

Приведем исходные тексты обеих программ. Текст файла shm_creator.c выглядит так:

```
#include<stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <inttypes.h>
int main()
{
int fd, status;
void* buffer;
fd = shm_open("/swd_es", O_RDWR | O_CREAT, 0777);
if( fd == -1 ) {
perror("shm_creator");
return EXIT_FAILURE;
}
status = ftruncate(fd, 100);
if (status!=0) {
perror("shm_creator");
return EXIT_FAILURE;
}
buffer=mmap(0,100,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
if (buffer == MAP_FAILED) {
perror("shm_creator");
```

```
return EXIT_FAILURE;
}
sprintf(buffer, "It's a nice day today, isn't it?");
printf("shm_creator: %s\n", buffer);
return EXIT_SUCCESS;
}
```

Итак, сначала вызываем функцию *shm_open()*:

```
fd = shm_open("/swd_es", O_RDWR|O_CREAT, 0777);
```

Первый аргумент "/swd_es" — имя региона разделяемой памяти (или, как еще говорят, разделяемого объекта памяти);

Примечание

Обратите внимание, что когда имя разделяемого объекта начинается с символа /, объект будет помещен в служебный "каталог" /dev/shmem. То есть реальное имя создаваемого нами региона — /dev/shmem/swd_es.

Второй аргумент представляет собой битовую маску из нескольких флагов, к этим флагам относятся:

- O_RDONLY — открыть объект только для чтения;
- O_RDWR — открыть объект для чтения и записи;
- O_CREAT — создать разделяемый объект с режимом доступа, заданным третьим аргументом функции *shm_open()*. Если объект уже существует, то флаг O_CREAT игнорируется, за исключением случаев, когда указан еще и флаг O_EXCL;
- O_EXCL — этот флаг используют совместно с флагом O_CREAT.
  В результате, если разделяемый объект памяти уже существует, то функция *shm_open()* завершится с ошибкой;
- O_TRUNC — этот флаг работает, когда объект уже существует и успешно открыт для чтения/записи. При этом размер объекта становится равным нулю (режим доступа и идентификатор владельца сохраняются).

Функция вернет файловый дескриптор fd, который в последующем и будет использоваться для доступа к данному разделяемому объекту. Теперь нужно сделать, чтобы разделяемый объект имел нужный размер и параметры. Для этого используем функцию *shm_ctl()*:

```
ftruncate(fd, 100);
```

В качестве первого аргумента используется тот самый идентификатор объекта разделяемой памяти fd, который вернула функция *shm_open()*;

Примечание

Иногда вместо функции *ftrancate()* используют функцию *shm_ctl()*. Теперь созданный объект, имеющий нужный размер, необходимо отобразить на виртуальное адресное пространство нашего процесса: buffer = mmap(0, 100, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

Первый и последний аргументы в нашем случае не нужны — они требуются при работе с физической памятью. Второй аргумент (100) указывает, какой величины фрагмент разделяемого объекта стоит отобразить на адресное пространство процесса (мы отобразили весь объект). Третий аргумент представляет собой битовую маску, которая может содержать несколько флагов:

- PROT_EXEC — разделяемый объект доступен для исполнения;
- PROT_NOCACHE — не кэшировать разделяемый объект;
- PROT_NONE — разделяемый объект недоступен;

- PROT_READ — разделяемый объект доступен для чтения;
- PROT_WRITE — разделяемый объект доступен для записи.

Четвертый аргумент определяет режим отображения региона. В нашем случае лучше задать MAP_SHARED (остальные режимы используются для работы с физической памятью).

Пятый аргумент — идентификатор разделяемого объекта.

Функция *mmap()* возвращает указатель на область виртуальной памяти процесса, на который отображен разделяемый объект (buffer). С полученым указателем мы можем поступать, как нам вздумается. Все, что мы запишем по этому адресу, будет отображено на разделяемый объект (конечно же, столько байт, сколько мы задали функции *mmap()*).
Запишем в буфер текстовую строку: sprintf(buffer, "It's a nice day today, isn't it?");
Теперь посмотрим на исходный текст программы shm_user.c:

```
#include<stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
int main()
{
int fd;
char *buffer;
fd = shm_open("/swd_es", O_RDONLY, 0777);
if( fd == -1 ) {
perror("shm_user");
return EXIT_FAILURE;
}
buffer = mmap(0, 100, PROT_READ, MAP_SHARED, fd, 0 );
if (buffer == MAP_FAILED) {
perror("shm_user");
return EXIT_FAILURE;
}
printf("shm_user: %s\n", buffer);
munmap(buffer, sizeof (buffer));
return EXIT_SUCCESS;
}
```

Как видно из текста программы, для получения доступа к разделяемому объекту снова используется функция *shm_open()*. Для того чтобы отобразить разделяемый регион на адресное пространство процесса, используется функция *mmap()*. В результате получаем указатель buffer на область виртуальной памяти процесса, который можно использовать. Распечатаем содержимое разделяемого объекта:

```
printf("shm_user: %s\n", buffer);
```

По аналогии мы можем передавать между процессами любые структуры данных. Но помните о том, что за правильность интерпретации данных, содержащихся в разделяемой памяти, отвечаете вы сами.

Хорошо если бы вы догадались спросить: а как процесс, считывающий данные из разделяемой памяти, определяет, что запись данных уже закончена и данные готовы для чтения? Ответ: никак. Чтобы избежать нарушений целостности данных, нам нужно использовать механизмы *синхронизации*.

Основным POSIX-механизмом синхронизации процессов являются *семафоры*.

ЗАДАНИЕ

1. Написать программу для создание разделямой памяти.
2. Написать программа для чтение данных из области разделямой памяти.

ПРАКТИЧЕСКАЯ РАБОТА №6. ФУНКЦИИ ОБМЕНА СООБЩЕНИЯМИ

Цель работы: Научиться использовать функции обмена сообщениями между процессами.

Передача сообщений

Посылку сообщения выполняет клиент. Предварительно вызовом ConnectAttach()он создает соединение coid с каналом сервера (предполагается, что необходимые для этого значения nd, pid и chid сервера ему известны). Посылка сообщения осуществляется с помощью функции:

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

int MsgSend(int     coid,
                    const   void* smsg,
                    int     sbytes,
                    void*   rmsg,
                    int     rbytes);
```

Посылаемые данные берутся из буфера, указанного smsg. Предполагается, что сервер, приняв сообщение, выполняет соответствующее действие и шлет ответное сообщение, ожидаемое клиентом. Ответное сообщение сервера размещается в буфере, указанном rmsg. Число посланных байтов, задается в sbytes, а число байтов в ответе задается в rbytes.

Количество переданных байтов определяется минимальным размером буферов, используемых клиентом и сервером. Это гарантирует от переполнения буферов при приеме сообщения сервером и получении ответа клиентом.

Если процесс-сервер имел нить, которая ожидала прихода сообщения (была RECEIVE-блокирова на этом канале), то перенос сообщения в адресное пространство сервера осуществляется, немедленно, а принимающая сообщение нить сервера становится готовой для выполнения. Посылающая сообщение нить клиента при этом становится REPLY-блокированной. Если нити, ожидающей приема сообщения из данного канала, в сервере нет, то пославшая сообщение нить клиента становится SEND-блокированной и ставится в очередь к каналу в порядке приоритета вместе с другими нитями, так же пославшими сообщение в этот канал. Фактический перенос данных из адресного пространства клиента в адресное пространство сервера не происходит до тех пор, пока принимающая нить сервера не выполнит функцию получения данных из канала. После этого нить клиента, пославшая данные, становится REPLY-блокированной (ждет ответного сообщения).

В случае успешного выполнения функция возвращает значение статуса, заданного в аргументе status функции MsgReply(), которую выполняет нить сервера для посылки ответа. Если возникает ошибка, товозвращается -1 и в errno устанавливается код ошибки или, если нить сервера вместо MsgReply() использовала функцию MsgError(), errno получает значение ошибки от MsgError().

Функция MsgSend() принадлежит семейству функций MsgSend*() и семантически связана с функциями ConnectAttach(), TimerTimeout() и функциями семейства MsgReceiv*().

Рассмотрим пример передачи сообщения процессу с IDпроцесса равным pid в канал с IDканала равным chid.

```
#include <sys/neutrino.h>
#include <strino.h>
#include <stdio.h>
```

```
int main(void)
{
char *smsg="Это содержимое буфера сообщения";
char *rmsg[200]; //Это буфер ответа
int coid;
int pid=540704;
int chid=1;
/* pid – IDпроцесса, chid – IDканала */
//Установить соединение
coid=ConnectAttach(0,pid,chid,_NTO_SIDE_CHANNEL,0);
if(coid==-1)
{
fprintf(stderr,"Ошибка соединения\n")
exit(EXIT_FAILURE);
}

//Послатьсообщение
if(MsgSend(coid,smsg,strlen(smsg)+1, rmsg,sizeof(rmsg))==-1)
{
fprintf(stderr,"Ошибка MsgSend\n");
exit(EXIT_FAILURE);
}

if(strlen(rmsg)>0) printf("Сервер ответил \n%s\n",rmsg);
}
```

Прием сообщения

Процесс-сервер должен принять сообщение и послать ответ клиенту. Для приема сообщения сервером используется функция:

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

int MsgReceive(int chid,void *msg,int bytes,struct _msg_info *info);
```

Эта функция ожидает сообщение в канале chid. Принятое сообщение размещается в буфере, адрес которого указан в msg. Размер буфера msg задается в bytes в байтах. Число принятых байтов не может превысить размера буфера (контролируется ядром).

Если сообщение уже было в канале, когда нить сервера вызывает MsgReceive(), то оно немедленно копируется ядром в адресное пространство сервера. Если сообщения в канале нет, то принимающая сообщение нить сервера переходит в RECEIVE-блокированное состояние, ожидая пока сообщение от клиента не поступит в канал. При получении сообщения нить переходит в состояние готовности к выполнению (READY).

Если значение infoотлично от NULL, то в нем сохраняется дополнительная информация относительно сообщения и нити клиента, которая послала его. Если значение info равно NULL, то эта информация, при необходимости, может быть получена, с помощью функции MsgInfo() (описание этой функции содержит описание структуры _msg_info).

Если при выполнении функции возникает ошибка, то возвращается -1 и errno присваивается код ошибки. В случае успеха возвращается идентификатор rcvid, специфицирующий нить клиента, пославшую сообщение.

Посылка ответа

Получив сообщение от клиента сервер должен послать ему ответное сообщение с помощью функции:

```
#include <sys/neutrino.h>
int MsgReply(int rcvid,int status,const void* msg,int size);
```

rcvid - ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive().
status - статус завершения, который возвращается нити клиента, пославшей сообщение, при успешном завершении функции MsgSend().
msg - указатель буфера, содержащего ответное сообщение.
size - размер сообщения в байтах.

Функция посылает ответ с сообщением нити клиента, идентифицированной rcvid. При этом нить должна находится в REPLY-блокированном состоянии. Ответ может послать любая нить сервера. Важно только, чтобы на каждое принятое сервером сообщение следовал бы ответ и только один. Выполнение функции MsgSend(), вызванной нитью клиента, которой соответствует rcvid, завершается разблокированием нити и возвратом функцией MsgSend()значения статуса, заданного сервером в аргументе status при выполнении функции MsgReply().

Ядро контролирует, чтобы число байтов ответного сообщения, принимаемых клиентом, не превышало объема буфера клиента, предназначенного для приема ответа.

Функция MsgReply() не блокирует нить сервера, передача ответа выполняется немедленно. Нет никаких особых требований к порядку ответа, но в конечном счете серверу необходимо ответить на каждое принятое сообщение, чтобы разблокировать нить клиента, пославшую сообщение.

Заметим, что в функциях MsgSend() и MsgReceive() указывается количество посылаемых и принимаемых байт. Если они не совпадают, то выбирается минимальное из указанных значений с целью согласования размеров буферов передачи и приема. Аналогичная ситуация и с функцией MsgReply(). То есть, при необходимости сообщение будет "урезано" и лишние байты "отброшены".

Если возникает ошибка, то возвращается -1 и errno присваивается код ошибки.

Пример.

```
#include <sys/neutrino.h>
#include <stdlib.h>
#incude <stdio.h>
int main(void)
{
        server();
        return 0;
}
void server(void)
{
int rcvid;
int chid;
char message[512];
//Создать канал
chid=ChannelCreate(1);
printf(“Start server…\n”);
//Бесконечный цикл
while(1){
//Получить и вывести сообщение
rcvid=MsgReceive(chid,message,sizeof(message),NULL);
printf("Получил сообщение, rcvid = %X\n",rcvid);
printf("Сообщение такое: %s\n", message);
```

```
/*Подготовить отправить ответ - используем тот же буфер, что и для приема сообщения*/
strcpy(message,"Это ответ");
MsgReply(rcvid,EOK,message,sizeof(message));
}
}
```

Сценарии ответов

Использование MsgReply() достаточно прозрачно. Но за внешней простотой скрывается принципиальная возможность управления активностью клиента со стороны сервера. Во-первых, сервер совершенно не обязан посылать ответ клиенту как можно быстрее, и вообще никак не ограничен во времени с ответом клиенту. Поэтому сервер, при необходимости, может целенаправленно управлять временем нахождения клиента в REPLY-блокированном состоянии. При этом сервер может принимать и обрабатывать сообщения по тому же каналу от других клиентов (возможно, от других нитей того же процесса-клиента) и отсылать им ответы. Во-вторых, ответ сервера может быть пустым и преследовать цель только разблокировать клиента, например, MsgReply(rcvid,EOK,NULL,0). Если серверу необходимо проинформировать клиента о проблемах, возникших при обработке сообщения, полученного от клиента, то в этом случае для посылки клиенту ответа более удобно воспользоваться специальной функцией:

```
#include <sys/neutrino.h>
int MsgError(int rcvid,int error);
```

rcvid -ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive(), когда сообщение получено сервером.

error - код ошибки, отсылаемый клиенту.

При вызове сервером этой функции, функция MsgSend*() на стороне клиента завершается, возвращая -1 и присваивая errno значение error. Никаких данных клиенту не пересылается. Значение error, равное коду ERESTART, заставляет клиента немедленно повторить вызов MsgSend*() (этот код нельзя использовать после вызова MsgWrite()).

Если возникает ошибка, то возвращается -1 и errno присваивается код ошибки. В случае успеха - значение отличное от -1.

Управление сообщениями

Функции MsgSend(), MsgReceive(), MsgReply() должны указывать буфер фиксированной длины для приема/передачи сообщения. Однако не всегда имеется возможность заранее знать предельную длину сообщений. Сервер, например, может не знать, какие по длине сообщения ему придется принимать от клиентов, а также какие по длине ответы будут формироваться в результате обработки принятых сообщений. В таких случаях серверу может потребоваться осуществлять прием/передачу сообщений и ответов по частям, используя при приеме сообщения наряду с функцией MsgReceive() еще и вызовы функции MsgRead(), а при посылке ответа - вызовы функции MsgWrite(), прежде чем будет выполнена функция MsgReply().

Управление приемом сообщений

Если сервер не располагает буфером, способным всегда целиком разместить поступающие сообщения, то он должен действовать осторожно, а именно - предварительно выяснять длину посланного клиентом сообщения. Такую информацию функция MsgReceive() предоставляет серверу посредством аргумента info, если при создании канала был установлен флаг _NTO_CHF_SENDER_LEN.

Аргумент info является структурой _msg_info:

```
struct _msg_info{
int nd;//ID принимающего узла
```

```
int srcnd; //ID передающего узла
pid_t pid; //ID клиента
int32_t chid; //ID канала
int32_t scoid; //внутренний системный ID, используемый ядром
int32_t coid; //ID соединения
int32_t msglen; //количество принятых сервером байтов сообщения
int32_t tid; //ID нити, пославшей сообщение
int16_t priority; //приоритет нити, пославшей сообщение
int16_t flags; //дополнительные информационные флаги
int32_t srcmsglen; /*длина посланного сообщения в байтах (это поле актуально, если при
                                    выполнении функции ChennelCreat() был установлен флаг
                                    _NTO_CHF_SENDER_LEN)*/
}
```

Чтобы выяснить, целиком ли принято посланное клиентом сообщение, серверу достаточно проанализировать значения полей msglen и srcmsglen аргумента info. Если msglen < srcmsglen, то сервер принял только часть посланного сообщения, которая уместилась в буфере сервера. Остальная часть осталась в буфере клиента. Используя функциюMsgRead()сервер имеет возможность по частям дополучить сообщение.

ФункцияMsgRead()имеетпрототип:

```
#include <sys/neutrino.h>
int MsgRead(int rcvid,void* msg,int bytes,int offset);
```

Аргументы функции:

rcvid - ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive(),
msg - буфер сервера для приема части сообщения,
bytes - длина принимаемой сервером части сообщения,
offset - смещение относительно начала сообщения в буфере клиента.

Выполнение сервером функции MsgRead() оставляет соответствующую нить клиента в REPLY-блокированном состоянии. Поэтому сервер может многократно выполнять эту функцию для получения всего посланного нитью сообщения по частям, выделяя (например, динамически) буферы для каждой части или сразу обрабатывая принятую часть сообщения в одном и том же буфере. Когда прием и обработка сообщения полностью завершается, сервер, как и прежде, должен выполнить функцию MsgReply() для вывода нити клиента из REPLY-блокированного состояния.

Управление передачей ответа

Кроме приема по частям сообщения от клиента сервер может передавать по частям и ответ клиенту. При этом предполагается, что клиент располагает буфером, достаточным по величине для приема любого возможного ответа целиком. В противном случае ответ будет, все равно, принят только частично, заполнив буфер приема ответа, выделенный клиентом.

Для передачи клиенту ответа по частям сервер должен перед выполнением функции MsgReply() использовать функцию MsgWrite(), котораяимеетпрототип:

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

int MsgWrite(int rcvid,void* msg,int bytes,int offset);
```

Эта функция пишет данные в буфер ответа нити, ID которой указан в rcvid. Значение IDнити возвращается функцией MsgReceive() при получении сообщения сервером. Нить, по отношению к которой выполняется запись, должна быть в REPLY-блокированном состоянии. Выполнение MsgWrite() не выводит нить из REPLY-блокированного состояния.

Данные в количестве size байтов берутся из буфера, указанного в msg, и записываются в буфер ответа нити клиента, ожидающей ответ, начиная с места в буфере, отстоящего от начала буфера на величину offset байт (смещение от начала буфера).

Если размер ответа size превышает размер буфера ответа клиента, то переполнения буфера не произойдет, а превышающая размер буфера часть ответа не будет передана.

Чтобы закончить передачу ответа и вывести клиента из REPLY-блокированного состояния, серия вызовов MsgWrite() должна быть завершена вызовом функции MsgReply(). При этом ответ, отправляемый функцией MsgReply() не должен обязательно содержать какие-либо данные. Если он все-таки содержит данные, то они будут всегда записываться с нулевым смещением в буфере ответа нити назначения. Это - удобный способ записи заголовка ответа, когда он полностью передан.

Функция MsgWrite() возвращает число реально переданных байтов ответа. В случае ошибки возвращается -1 и устанавливается errno.

Передача сообщений с использованием векторов ввода/вывода

Если сообщение состоит из несвязных частей, то его передача может осуществляться с использованием так называемых векторов ввода/вывода. Под вектором ввода/вывода (IOV) понимается структура, которая содержит два поля - адрес и длину части сообщения. Для определения IOV используется системный тип iov_t, имеющий определение вида:

```
typedef struct iovec{
void *iov_base;
size_t iov_len;
} iov_t;
```

Для инициализации значения IOV удобно использовать системную макрокоманду:

SETIOV(_iov, addr, _len);

где:
_iov - имя вектора ввода/вывода;
addr - адрес части сообщения;
_len - длина части сообщения в байтах.

При передаче несвязного сообщения для каждой части такого сообщения формируется свой вектор ввода/вывода. Затем из них формируется массив векторов ввода/вывода, в котором вектора располагают в нужном порядке следования частей сообщения при передаче.

Для посылки клиентом сообщения с использованием IOV применяется функция:

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

int MsgSendv(int          coid,
                    const iov_t*  siov,//Массив IOV сообщения
                    int           sparts,//Количество IOV сообщения
                    const iov_t*  riov,//Массив IOV ответа
                    int           rbytes);//Количество IOV ответа
```

Для приема сервером сообщения в несвязную область памяти (буфер) с использованием IOV применяется функция:

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

int MsgReceivev(int           chid,
                        const iov_t*    riov,//Массив IOV буфера
                        int             sparts,//Количество IOV буфера
                        struct_msg_info*  riov);
```

Заметим, что структура буфера сообщения клиента и буфера приема сервера не обязаны совпадать. Кроме того, клиент и сервер не обязаны договариваться о способе передачи/приёма сообщения, т.е. клиент, например, может использовать для посылки сообщения функцию MsgSend(), а сервер – MsgReceivev() и наоборот.

ЗАДАНИЕ

3. Написать программу сервер для приема сообщений.
4. Написать программу клиента для отправки сообщение на серверную программу.