

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ «МАМИ»

А.А .Иванов, В.В.Матросова, Е.Г. Мурачев, Ю.А. Савостьянок.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по выполнению курсовых работ
по дисциплине
«Системное программное обеспечение»

для студентов, обучающихся по специальностям 220201.65, 220301.65 и
направлению 220200.62

Одобрено методической комиссией
по специальности

Москва
2010

УДК _____

ББК _____

(только для учебных пособий)

Разработано в соответствии с Государственным образовательным стандартом ВПО 2000 г. для направления (специальности) подготовки _____ на основе примерной (рабочей) программы дисциплины «Системное программное обеспечение»

Рецензенты: профессор кафедры «Автотракторное электрооборудование» МГТУ «МАМИ» Р.А. Малеев

профессор кафедры «Автомобили» им. Е.А. Чудакова
В.В. Селифонов

Работа подготовлена на кафедре «Автоматика и процессы управления»

Исследование систем автоматического регулирования: методические указания / А.А. Иванов, В.В. Матросова, Е.Г. Мурачев, Ю.А. Савостьянок, – 1-е изд. – М. : МГТУ «МАМИ», 2010. – 55 с.

В методических указаниях рассматриваются общие принципы построения операционных систем, построение и использование библиотек системных программ в среде операционной системы Linux, принципы построения программ с помощью интерпретатора PostScript, а также с помощью алгоритмического языка C++, способы хранения и вызов системных программ.

СОДЕРЖАНИЕ

Введение	4
1 Цель и задачи курсового проектирования	5
2 Организация и последовательность выполнения курсовой работы	5
2.1 Задание на курсовую работу	5
2.2 Объем и содержание курсовой работы	5
2.3 Последовательность выполнения работы	6
3 Варианты заданий	7
4 Информация для выполнения курсовой работы	16
4.1 С-функции работы с файлами	19
4.2 Управление оперативной памятью	24
4.3 Операции new и delete	27
4.4 Демоны	29
4.3 Язык PostScript	28
4.6 Командный интерпретатор	30
Приложения	49
Литература	54

Введение

Курсовая работа по дисциплине «Системное программное обеспечение» выполняется студентами специальности 210100, 210200 и направления 550200 в 6 семестре. В рамках курсовой работы должно быть разработано приложение для операционной системы UNIX, при выполнении курсовой работы используются знания, полученные студентами при изучении дисциплины «Программирование и основы алгоритмизации», а также дисциплины «Информатика».

1 ЦЕЛЬ И ЗАДАЧИ КУРСОВОГО ПРОЕКТИРОВАНИЯ

1.1 Целью курсового проектирования является приобретение практических навыков по разработке структуры приложения, алгоритмов и программ для их реализации с использованием языков C++ и PostScript для операционной системы UNIX/Linux.

1.2 Задачей курсовой работы является разработка приложения по заданным исходным данным:

- разработка программы-демона;
- построение грамматики для заданного языка и автомата для его распознавания
- построение лексического анализатора;
- разработка программы обработки запроса ядром UNIX для символьного устройства;
- разработка программы обработки запроса ядром UNIX для блочного устройства;
- разработка программы вывода графиков на языке PostScript;
- разработка программ, использующих системный вызов;
- внесение составленной программы в библиотеку системных программ;
- разработка системного вызова для обращения к созданной программе;
- разработка системы сообщений о результатах, полученных в ходе Выполнения программы.

2. ОРГАНИЗАЦИЯ И ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ КУРСОВОЙ РАБОТЫ

Курсовое проектирование является формой самостоятельной работы студента и выполняется по индивидуальному заданию.

Задание на курсовую работу выдается преподавателем на первом занятии по курсу СПО 6 семестра, защита проводится в конце того же семестра перед экзаменом. На защите демонстрируется выполнение программы с соответствующими пояснениями. В ходе выполнения курсовой работы студент консультируется с руководителем, назначенным кафедрой.

За правильность проектных решений, качество оформления работы, своевременность выполнения отдельных этапов и представления к защите отвечает студент.

2.1 Задание на курсовую работу

Задание на курсовую работу выбирается студентом по номеру группы и порядковому номеру студента в журнале.

2.2 Объем и содержание курсовой работы

Работа состоит из расчетно-пояснительной записки (РПЗ) и программы, представленной на CD-диске.

Техническое задание включает общие и специальные требования к программе.

Объем пояснительной записки составляет 30-40 машинописных страниц (формат А4), РПЗ должна быть написана четко и кратко, содержать пояснения к разработанному приложению, обоснование принятых решений. РПЗ должна включать следующие разделы:

- 1) титульный лист (приложение А),
- 2) бланк задания, подписанный преподавателем и студентом (приложение Б),
- 3) содержание,
- 4) перечень условных обозначений и сокращений в алфавитном порядке в виде списка, в котором слева приводится сокращение, справа – его расшифровка,
- 5) основная часть РПЗ:

Краткие теоретические сведения: анализ существующих программ подобного класса; особенности создания приложений для ОС UNIX/Linux;

Разработка структуры приложения;

Разработка алгоритма решения задачи;

Разработка программы;

Проектирование интерфейса приложения (если это предусмотрено заданием на КР);

Заключение;

Список использованных источников;

Приложения:

- спецификация программного обеспечения (приложение В),
- текст программы (приложение Г),
- руководство пользователя (системного программиста) (приложение Д).

2.3 Последовательность выполнения работы

Курсовая работа разрабатывается в последовательности, соответствующей содержанию РПЗ (п. 2.2).

Расчетно-пояснительная записка и графический материал оформляются в соответствии с требованиями ЕСКД и ЕСПД (Единая система конструкторской документации, Единая система программной документации).

Подготовленная и оформленная работа, прошедшая экспертизу на выполнение требований ЕСКД и ЕСПД представляется преподавателю не позднее, чем за неделю до защиты.

Защита работы происходит на 16 или 17 неделе семестра.

3 ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

Написать и отладить программу на языке C++, использующую системные вызовы для управления файлами в среде UNIX (тип файла - обычный, каталоги).

Вариант 2

Написать и отладить программу на языке C++, использующую системные вызовы для управления файлами в среде UNIX (тип файла - специальные, FIFO).

Вариант 3

Разработать на языке PostScript программу вывода графика функции $f()$, которая строит n периодов, соединяя точки, расположенные с интервалом m° , отрезками прямых. Размер поля рисования 10x4 см, график расположен в центре листа и обведен рамкой.

$$a) f = \sin(2a+x), \quad n=4 \quad m=5^\circ$$

$$б) f = \cos(x), \quad n=3, \quad m=10^\circ$$

$$в) f = \sin(x)/2, \quad n=4; \quad m=8^\circ$$

Вариант 4

Написать процедуру на языке PostScript, которая вдоль ранее созданного графического пути строит штрих - пунктирную линию, состоящую из:

- 1) прямоугольных штрихов со срезанными краями и кружков между ними;
- 2) двукратное проведение штриховой линии с разными параметрами,...

Вариант 5

Разработать на языке PostScript программу, позволяющую строить различные графические изображения с использованием функций $\cos()$ и $\sin()$, не используя при этом встроенные функции языка. Разработка данной программы предполагает вывод изображений функций $\cos()$ и $\sin()$ одновременно в различных проекциях и пространствах (двумерном,

трехмерном), различным цветом, со смещенной системой координат на 90° . Изображения строятся по точкам, расположенным друг от друга с интервалом 5° , 10° , 36° , 126° , 181° , 359° . Для работы с цветом использовать процедуру изменения цвета изображений (по RGB- матрице).

Вариант 6

Написать на языке PostScript файл для печати рекурсивных геометрических объектов:

- а) ковра Серпинского;
- б) кривых Гильберта;
- в) кривых Коха;
- г) кривых Пеано.

Вариант 7

Разработать на языке PostScript программу построения и вывода на печать графика заданной функции:

- а) $f = \text{tg}(l+2a)/(a+b)$;
- б) $f = \exp(3a+2)$;
- в) $f = \log(1+a/4)/3$

Вариант 8

Используя системный вызов *stat*, напишите программу, определяющую тип файла: обычный файл, каталог, устройство, FIFO-файл.

Вариант 9

Напишите программу удаления файлов и каталогов, заданных в **argv**. Делайте системный вызов *stat*, чтобы определить тип файла (файл/каталог). Программа должна отказываться удалять файлы устройств. Для удаления пустого каталога (не содержащего иных имен, кроме "." и "..") следует использовать системный вызов

rmdir(**имя_каталога**) ;

(если каталог не пуст - **errno** получит значение *EEXIST*); а для удаления обычных файлов (не каталогов)

unlink(**имя_файла**) ;

Программа должна запрашивать подтверждение на удаление каждого файла, выдавая его имя, тип, размер в килобайтах и вопрос "удалить ?".

Вариант 10

Напишите функцию печати текущего времени в формате ЧЧ:ММ:СС ДД-МЕС-ГГ. Используйте системный вызов *time()* и функцию *localtime()*. Рекомендуется использование стандартной функции *ctime()*.

Вариант 11

Структура *stat*, заполняемая системным вызовом *stat()*, кроме прочих полей содержит поля типа *time_t* **st_ctime**, **st_mtime** и **st_atime** - время последнего изменения содержимого I-узла файла, время последнего изменения файла и время последнего доступа к файлу.

- Поле **st_ctime** изменяется (устанавливается равным текущему астрономическому времени) при применении к файлу вызовов *creat*, *chmod*, *chown*, *link*, *unlink*, *mknod*, *utime**, *write* (т.к. изменяется длина файла); Это поле следует рассматривать как время модификации прав доступа к файлу;
- **st_mtime** - *write*, *creat*, *mknod*, *utime*; Это поле следует рассматривать как время модификации содержимого файла (данных);
- **st_atime** - *read*, *creat*, *mknod*, *utime*; Это поле следует рассматривать как время чтения содержимого файла (данных).

Модифицируйте функцию *typeOf()*, чтобы она печатала еще и эти даты.

```
utime(имяФайла, NULL);
```

Он используется для взаимодействия с программой *make* - в команде *touch*. Изменить время можно только своему файлу.

Вариант 12

Напишите аналог команды *ls -l*, выдающий имена файлов каталога и их коды доступа в формате **rw-rw-r--**. Для получения кодов доступа используйте вызов *stat*

```
stat( имяФайла, &st);  
кодыДоступа = st.st_mode & 0777;
```

Для изменения кодов доступа используется вызов

```
chmod(имя_файла, новые_коды);
```

Можно изменять коды доступа, соответствующие битовой маске

```
0777 | S_ISUID | S_ISGID | S_ISVTX
```

(смотри <sys/stat.h>). Тип файла (см. функцию *typeOf*) не может быть изменен. Изменить коды доступа к файлу может только его владелец.

Вариант 13

Вот программа, которая каждые 2 секунды проверяет - не изменилось ли содержимое текущего каталога:

```
#include <sys/types.h>
#include <sys/stat.h>
extern char *ctime();
main(){
    time_t last; struct stat st;
    for( stat(".", &st), last=st.st_mtime; ;
sleep(2)){
        stat(".", &st);
        if(last != st.st_mtime){
            last = st.st_mtime;
        printf("Был создан или удален какой-то файл: %s",
                ctime(&last));
        }
    }
}
```

Модифицируйте ее, чтобы она сообщала **какое** имя (имена) было удалено или создано (для этого надо при запуске программы прочитать и запомнить содержимое каталога, а при обнаружении модификации - перечитать каталог и сравнить его с прежним содержимым).

Вариант 14

Вот программа, которая каждые 2 секунды проверяет - не изменилось ли содержимое текущего каталога:

```
#include <sys/types.h>
#include <sys/stat.h>
extern char *ctime();
main(){
    time_t last; struct stat st;
    for( stat(".", &st), last=st.st_mtime; ;
sleep(2)){
        stat(".", &st);
        if(last != st.st_mtime){
            last = st.st_mtime;
        printf("Был создан или удален какой-то файл: %s",
                ctime(&last));
        }
    }
}
```

Модифицируйте ее, чтобы она выдавала сообщение, если указанный вами файл был кем-то прочитан, записан или удален. Вам следует отслеживать изменение полей `st_atime`, `st_mtime` и значение `stat() < 0` соответственно. Если файл удален - программа завершается.

Вариант 15

Современные *UNIX*-машины имеют встроенные таймеры (как правило несколько) с довольно высоким разрешением.

Некоторые из них могут использоваться как "будильники" с обратным отсчетом времени: в таймер загружается некоторое значение; таймер ведет обратный отсчет, уменьшая загруженный счетчик; как только это время истекает - посылается сигнал процессу, загрузившему таймер.

Вот как, к примеру, выглядит функция задержки в микросекундах (миллионных долях секунды). Примечание: эту функцию не следует использовать вперемежку с функциями *sleep* и *alarm*.

```
#include <sys/types.h>
#include <signal.h>
#include <sys/time.h>

void do_nothing() {}

/* Задержка на usec миллионных долей секунды
(микросекунд) */
void usleep(unsigned int usec) {

    struct itimerval      new, old;
    /* struct itimerval  содержит поля:
       struct timeval    it_interval;
       struct timeval    it_value;

       Где struct timeval содержит поля:
       long              tv_sec;      -- число целых
секунд
                                long              tv_usec;      -- число
микросекунд

    */
    struct sigaction      new_vec, old_vec;

    if (usec == 0) return;
```

```

/* Поле tv_sec содержит число целых
секунд.
Поле tv_usec содержит число
микросекунд.

it_value - это время, через которое
В ПЕРВЫЙ раз таймер "прозвонит",
то есть пошлет нашему
процессу сигнал SIGALRM.

Время, равное нулю,
немедленно остановит таймер.

it_interval - это интервал времени,
который будет загружаться
в таймер после каждого
"звонка"
(но не в первый раз).

Время, равное нулю,
остановит таймер
после его первого
"звонка".

*/
new.it_interval.tv_sec = 0;
new.it_interval.tv_usec = 0;
new.it_value.tv_sec = usec / 1000000;
new.it_value.tv_usec = usec % 1000000;

/* Сохраняем прежнюю реакцию на сигнал
SIGALRM в old_vec,
записываем в качестве новой реакции
do_nothing()
*/
new_vec.sa_handler = do_nothing;
sigemptyset(&new_vec.sa_mask);
new_vec.sa_flags = 0;

sighold(SIGALRM);
sigaction(SIGALRM, &new_vec, &old_vec);

```

```

/* Загрузка интервального таймера
значением new, начало отсчета.
* Прежнее значение спасти в old.
* Вместо &old можно также NULL - не
спасать.
*/
setitimer(ITIMER_REAL, &new, &old);

/* Здесь вам необходимо ждать прихода
сигнала SIGALRM */

/* Здесь вам необходимо восстановить
реакцию на SIGALRM */

/* Здесь вам необходимо восстановить
прежние параметры таймера */
}

```

Задание: в теле функции `usleep()` опишите функцию `setitimer()`, которая будет ждать прихода сигнала `SIGALRM`, восстанавливать реакцию на `SIGALRM`, восстанавливать прежние параметры таймера.

Вариант 16

Напишите "часы", выдающие текущее время каждые 3 секунды.

Вариант 17

Системный вызов `fork()` (вилка) создает **новый** процесс: **копию** процесса, издавшего вызов. Отличие этих процессов состоит только в возвращаемом `fork`-ом значении:

0 - в новом процессе.
pid нового процесса - в исходном.

Задание: сделайте так, чтобы вызов `fork` завершился неудачей, если таблица процессов переполнена.

Вариант 18

Перепишите следующий алгоритм при помощи `longjmp`. Используйте нелокальный переход вместо цепочки `return`-ов

```

#define FOUND      1 /* ответ найден      */
#define NOTFOUND  0 /* ответ не найден */
int value;          /* результат */
main(){      int i;
    for(i=2; i < 10; i++){
        printf( "попыаем i=%d\n", i);
        if( test1(i) == FOUND ){
            printf("ответ %d\n", value); break;
        }
    }
}
test1(i){      int j;
    for(j=1; j < 10 ; j++ ){
        printf( "попыаем j=%d\n", j);
        if( test2(i,j) == FOUND ) return FOUND;
        /* "сквозной" return */
    }
    return NOTFOUND;
}
test2(i, j){
    printf( "попыаем(%d,%d)\n", i, j);
    if( i * j == 21 ){
        printf( "    Годятся (%d,%d)\n", i, j);
        value = j; return FOUND;
    }
    return NOTFOUND;
}

```

Обратите внимание, что при возврате ответа через второй аргумент *longjmp* надо прибавить 1, а при печати ответа эту единицу необходимо отнять. Это надо сделать на случай ответа *j==0*, чтобы функция *setjmp* не вернула бы в этом случае значение 0 (признак **установки** контрольной точки).

Вариант 19

Написать программу на C++, выполнение которой позволяет:

создать файл,

считать данные с клавиатуры в буфер;

поместить данные из буфера в созданный файл;

закрыть этот файл;

открыть этот файл и вывести его содержимое на экран монитора, закрыть файл.

Вариант 20

Написать программу на C++, выполнение которой позволяет:

проверить наличие файла на диске;

если файл существует, открыть его. Если файл не существует создать его;

считать данные с клавиатуры в буфер;

поместить данные из буфера в созданный файл;

закрыть этот файл.

Отобразить содержимое файла на экране

Вариант 21

Написать программу на C++, выполнение которой позволяет:

проверить наличие двух файлов на диске;

если файлы существуют, открыть их. Если файлы не существуют, создать их, ввести данные;

перенаправить информацию из одного файла в другой;

закрыть файлы.

Отобразить содержимое файлов на экране.

Вариант 22

Написать программу, в которой инициализация переменных различного типа осуществляется с помощью указателей. Значения переменных выводятся на экран.

Вариант 23

Написать программу, определяющую минимальное, максимальное значения элементов одномерного массива. Элементы массива вводятся с клавиатуры с использованием указателей.

Вариант 24

Вывести на экран номер элемента одномерного массива, его значение, адрес ячейки памяти с использованием указателей. Элементы массива вводятся с клавиатуры.

Вариант 25

Написать программу, суммирующую элементы одномерного массива. Элементы массива вводятся с клавиатуры. Ввод данных, обращение к элементам массива осуществляется с помощью указателей.

Вариант 26

Отсортировать элементы массива. Ввод данных осуществляется с использованием указателей.

Вариант 27

Написать программу с использованием переменной типа структура. Ввод/вывод, обращение к элементам структуры осуществляется с помощью указателей.

Вариант 28

Написать программу с использованием операций new, delete (например, просуммировать элементы массива, память под массив распределить динамически).

4 ИНФОРМАЦИЯ ДЛЯ ВЫПОЛНЕНИЯ КУРСОВОЙ РАБОТЫ

Системное программирование (или **программирование систем**) — род деятельности, заключающийся в работе над системным программным обеспечением.

Основная отличительная черта системного программирования по сравнению с прикладным программированием заключается в том, что результатом последнего является выпуск программного обеспечения, предлагающего определённые услуги пользователям (например, текстовый процессор). В то время как результатом системного программирования является выпуск программного обеспечения, предлагающего сервисы по взаимодействию с аппаратным обеспечением (например, дефрагментация жёсткого диска), что подразумевает сильную зависимость таких программ от аппаратной части. В частности выделим следующее:

- программист должен учитывать специфику аппаратной части и другие свойства системы в которой функционирует программа, использовать эти свойства, например, применяя специально оптимизированный для данной архитектуры алгоритм.

- обычно используется низкоуровневый язык программирования или такой диалект языка программирования, который
 - позволяет функционирование в окружении с ограниченным набором системных ресурсов.
 - работает максимально эффективно и имеет минимальное запаздывание по времени завершения.
 - имеет маленькую библиотеку времени выполнения (RTL) или не имеет её вообще.
 - позволяет прямое управление (прямой доступ) к памяти и управляющей логике.
 - позволяет делать ассемблерные вставки в код.
- отладка программы может быть затруднена при невозможности запустить её в отладчике из-за ограничений на ресурсы, поэтому может применяться компьютерное моделирование для решения этой проблемы.

Системное программное обеспечение служит для обеспечения эффективной работы аппаратуры компьютера.

К группе системных программ относятся операционные системы, операционные оболочки, утилиты, драйверы, архиваторы, антивирусные и некоторые другие программы.

- Операционные системы представляют собой пакеты программ, которые обеспечивают эффективную работу всех аппаратных средств компьютера, а также возможность управления всеми его ресурсами (см. раздел 3.5).

- Операционные оболочки представляют собой дополнительные программы, которые предназначены для повышения удобства управления работой операционных систем (см. раздел 3.7.2).

- Утилиты (utility — полезность) представляют собой небольшие, но очень полезные программы, которые выполняют разнообразные вспомогательные функции по управлению работой аппаратных средств, по повышению эффективности их работы, проверки работоспособности, обслуживания и настройки.

- Для выполнения операций по обмену данными между программой и различными внешними устройствами в состав операционной системы включается ряд специализированных программ, которые принято называть

драйверами (drive — управлять). Отсутствие или применение не соответствующего устройству драйвера делает бесполезным это устройство.

- Программы-архиваторы служат для создания архивных копий важных программ и наборов данных. Архиваторы также минимизируют объем, который нужен для размещения архива на внешнем носителе.

- Антивирусные программы обеспечивают пользователя необходимыми средствами борьбы с упоминавшимися выше компьютерными вирусами.

Разрабатываются, настраиваются и поддерживаются в рабочем состоянии системные программы специалистами, которых принято называть системными программистами. Они должны обладать высокой квалификацией, в деталях знать аппаратное обеспечение компьютера и способы работы с данными на машинном уровне. Рядовому пользователю приходится постоянно сталкиваться с системными программами, но уже с точки зрения их эксплуатации, использования их возможностей для решения своих задач. С некоторыми программами этой группы мы познакомимся в следующих главах пособия.

Системный вызов в программировании и вычислительной технике — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции. Современные операционные системы (ОС) предусматривают разделение времени между выполняющимися вычислительными процессами (многозадачность) и разделение полномочий, препятствующее исполняемым программам обращаться к данным других программ и оборудованию. Ядро ОС исполняется в привилегированном режиме работы процессора. Для выполнения межпроцессной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова. С точки зрения программиста системный вызов обычно выглядит как вызов подпрограммы или функции из системной библиотеки. Однако системный вызов как частный случай вызова такой функции или подпрограммы следует отличать от более общего обращения к системной библиотеке, поскольку последнее может и не требовать выполнения привилегированных операций.

Структура — это набор из одной или более переменных, возможно различных типов, сгруппированных под одним именем для удобства обработки. (В некоторых языках, самый известный из которых паскаль, структуры называются "записями").

Традиционным примером структуры является учетная карточка работающего:

"служащий" описывается набором атрибутов таких, как фамилия, имя, отчество (ф.и.о.), адрес, код социального обеспечения, зарплата и т.д.

Некоторые из этих атрибутов сами могут оказаться структурами: ф.и.о. Имеет несколько компонент, как и адрес, и даже зарплата. Структуры оказываются полезными при организации сложных данных особенно в больших программах, поскольку во многих ситуациях они позволяют сгруппировать связанные данные таким образом, что с ними можно обращаться, как с одним целым, а не как с отдельными объектами.

Процессы могут обмениваться между собой информацией через файлы. Существуют файлы с необычным поведением - так называемые **FIFO-файлы** (**first in, first out**), ведущие себя подобно очереди. У них указатели чтения и записи разделены. Работа с таким файлом напоминает проталкивание шаров через трубу - с одного конца мы вталкиваем данные, с другого конца - вынимаем их. Организация FIFO-файлов идет по принципу «первый пришёл — первый вышел». Операция чтения из пустой "трубы" приостановит вызов *read* (и издавший его процесс) до тех пор, пока кто-нибудь не запишет в **FIFO-файл** какие-нибудь данные. Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация событий в Windows. Когда пользователь оказывает какое-то действие на приложение, то в приложении не вызывается соответствующая процедура (ведь в этот момент приложение может совершать другие действия), а ему присылается сообщение, содержащее информацию о совершенном действии, это сообщение ставится в очередь, и только когда будут обработаны сообщения, пришедшие ранее, приложение выполнит необходимое действие.

4.1 С-функции работы с файлами

Изучение и использование С-функций работы с файлами.

Рассмотрим основные системные вызовы, связанные с проведением операций ввода-вывода, которые наиболее часто используются в операционной системе UNIX. Ввод-вывод в ОС UNIX является крайне простой операцией и налагает на пользователя минимум ограничений.

Системный вызов open

Системный вызов **open** (открыть файл) имеет следующий формат:

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

```
open(name, flags, mode)
```

```
char *name;
```

```
int flags, mode;
```

Системный вызов **open** открывает файл с именем **name** для чтения и/или записи. Режим открытия файла определяется значением параметра **flags**. Это значение может быть задано как результат логического сложения следующих признаков (в любой комбинации):

`O_RDONLY` - открыть только для чтения.

`O_WRONLY` - открыть только для записи.

`O_RDWR` - открыть для чтения и записи.

`O_NDELAY` - не блокировать при открытии. Если операция открытия задерживается по каким-либо причинам, например, при отсутствии готовности линии связи, процесс не приостанавливается. Возвращается код ошибки.

`O_APPEND` - открыть для дозаписи. Обычно, при открытии файла, указатель текущей позиции чтения/записи устанавливается на его начало, но, если задан режим `O_APPEND`, этот указатель устанавливается на конец файла.

`O_CREAT` - создать файл, если он не существует.

`O_TRUNC` - сократить размер файла. В режиме `O_TRUNC`, если указанный файл существует, его размер усекается до нуля.

`O_EXCL` - выдавать ошибку при попытке создания существующего файла. (Этот признак используется в сочетании с признаком `O_CREAT`). Режим может применяться для организации простого механизма блокировки.

В случае успешного завершения вызова **Open**, возвращается дескриптор открытого файла, иначе - значение -1 и в переменную **errno** записывается код ошибки.

При неудачном завершении, переменная **errno** может принимать следующие значения:

[ENOTDIR] - указанное имя, содержит компоненту, которая не является справочником;

[ENOENT] - указанный файл не существует и режим `O_CREAT` не был определен;

[EPEKM] - указанное имя содержит символ, отсутствующий в коде ASCII*);

[ELOOP] - число косвенных ссылок в указанном имени превышает максимально допустимое значение;

[EROFS] - указанный файл содержится в файловой системе закрытой по записи и не может быть модифицирован;

[ETXTBSY] - попытка открыть на запись файл, содержащий загрузочный модуль выполняющейся реентерабельной программы;

[EACCES] - режим доступа указанного файла не соответствует запросу;

[EFAULT] - адрес параметра системного вызова выходит за границы адресного пространства процесса;

[EISDIR] Попытка открыть на запись справочник;

[EMFILE] - переполнена таблица дескрипторов открытых файлов процесса;

[ENXIO] - указанный файл является внешним устройством, которое в данный момент не готово к работе.

Системный вызов creat

Системный вызов **creat** (создать файл) имеет следующий формат:

creat(name,mode)

char *name;

int mode;

Функция **creat** создает новый (или подготавливает к повторной записи уже существующий) файл с именем **name**. Если файл еще не существует, значение параметра **mode** используется для формирования режимов доступа создаваемого файла, при этом учитывается значение маски режимов процесса. Значение параметра **mode** составляется по правилам, приведенным в описание системного вызова **chmod**. Если указанный файл существует, то его владелец и режим доступа остаются прежними, а файл усекается до нулевой длины. Кроме того, файл открывается для записи и возвращается его дескриптор.

При неудачном завершении, операции **creat** возвращается значение -1, в остальных случаях возвращается дескриптор открытого файла (целое положительное число).

После неудачного завершения системного вызова **creat** переменная **errno** может принимать следующие значения:

[ENOTDIR] - указанное имя содержит компоненту, которая не является справочником;

[ENOENT] - задано слишком длинное или пустое имя файла, или указанный файл не существует или какой-либо из справочников, входящих в имя файла, не доступен для просмотра;

[EPEKM] - указанное имя содержит символ, отсутствующий в коде ASCII;

[ELOOP] - число косвенных ссылок в указанном имени превышает максимально допустимое значение;

[EROFS] - попытка создания файла в файловой системе, закрытой на запись;

[ETXTBSY] - попытка сократить длину файла, содержащего загрузочный модуль реентерабельной программы (разделяемый текстовый сегмент), которая в настоящий момент выполняется;

[EACCES] - режим доступа указанного файла не соответствует запросу;

[EFAULT] - адреса аргументов системного вызова выходят за границы памяти, доступной данному процессу;

[EISDIR] - указанное имя файла является именем справочника;

[EMFILE] - переполнилась таблица открытых файлов процесса;

[ENXIO] - указано имя специального файла, для которого нет соответствующего устройства.

Параметр **mode** задается произвольно, в нем не обязательно должно быть разрешение на запись. Эта возможность используется программами, которые работают с временными файлами с фиксированными именами. Создание

производится с режимом, запрещающим запись. Затем, если другая программа пытается выполнить вызов **creat**, возвращается ошибка и программе становится известно, что в данный момент это имя использовать нельзя. Все действия, перечисленные для вызова **create**, можно выполнить с помощью системного вызова **open**. Вызов **create** оставлен для совместимости с ранними версиями ОС Unix.

Системный вызов read

Системный вызов **read** (чтение файла) имеет следующий формат:

read (fd, buf, nbytes)

char *buf;

int fd, nbytes;

Системный вызов **read** обеспечивает считывание **nbytes** данных в массив **buf** из файла с дескриптором **fd**. Информация читается из файла по текущему указателю позиции чтения/записи. После завершения пересылки данных значение указателя увеличивается на число считанных байт. Для некоторых файлов значение указателя позиции чтения/записи не имеет смысла (например, для терминала), тем не менее, данные передаются. При успешном завершении вызова, возвращается число считанных байт, в случае ошибки - значение -1, при достижении конца файла в процессе чтения - число 0.

При возникновении ошибки чтения, переменная **errno** может принимать следующие значения:

[EBADF] - указанный дескриптор не является дескриптором файла, открытого для чтения;

[EFAULT] - адрес параметра системного вызова не входит в адресное пространство процесса;

[EINTR] - чтение с медленного устройства прервано до передачи данных.

Системный вызов write

Системный вызов **write** (запись в файл) имеет следующий формат:

write (fd, buf, nbytes)

char *buf;

int fd, nbytes;

Системный вызов **write** записывает **nbytes** данных из массива **buf** в файл с дескриптором **fd**. Информация записывается в файл по текущему указателю позиции чтения/записи. После завершения пересылки данных, значение указателя увеличивается на число записанных байт. Для некоторых файлов значение указателя позиции чтения/записи не имеет смысла, (например, для терминала), тем не менее, данные передаются.

Если статус файла, в который записывается информация, содержит признак set-UID и процесс работает в непривилегированном режиме, данный признак удаляется (в целях защиты информации). При успешном завершении вызова возвращается число записанных байт, в случае ошибки - значение -1.

При возникновении ошибки, переменная **errno** может принимать следующие значения:

[EBADF] - указанный дескриптор не является дескриптором файла, открытого для записи;

[EPIPE] - попытка записи в программный канал, который никто не читает;

[EPIPE] - запись в файл типа "гнездо" в режиме SOCK_STREAM, при отсутствии соединения;

[EFBIG] - при записи в файл превышает допустимый размер файла;

[EFAULT] - адрес параметра системного вызова не входит в адресное пространство процесса.

Системный вызов close

Системный вызов **close** (закрыть файл) имеет следующий формат:

ans = close (fd)

int fd;

Системный вызов **close** удаляет дескриптор fd из таблицы дескрипторов открытых файлов процесса. Если удаленный дескриптор был последним ссылающимся на данный файл, то весь контекст работы с файлом теряется. Для обычного файла это указатель позиции чтения/записи и режим блокировки. Хотя, при завершении процесса, все открытые им файлы автоматически закрываются, число одновременно открытых файлов ограничено, поэтому данный вызов может оказаться необходимым для программ, работающих с большим количеством файлов.

При порождении нового процесса (см описание вызова **fork**) все его дескрипторы указывают на те же объекты, что и дескрипторы процесса-предка. После выполнения вызова **execve** в порожденном процессе, новая программа также наследует все активные дескрипторы. Для того, чтобы сделать недоступными новой программе уже открытые файлы, соответствующие дескрипторы можно переопределить с помощью **dup2** или удалить с помощью системного вызова **unlink**. Однако бывают ситуации, в которых уже открытые файлы могут потребоваться при неудачном завершении системного вызова **execve**. В таких случаях, применение вызова **fcntl** обеспечивает закрытие определенных файлов после успешного старта новой программы. В случае успешного завершения, системный вызов **close** возвращает значение 0, иначе - значение -1 и код ошибки в переменной **errno**. Код ошибки:

[EBADF] - указанный дескриптор не является дескриптором открытого файла.

Системный вызов lseek

Системный вызов **lseek** (установка указателя чтения/записи) имеет следующий формат:

```
#define L_SET 0 /* установка * /
# define L_INCR 1 /* смещение */
# define L_XTND 2 /* увеличение размера файла */
```

long lseek (fd, offset, whence)

int fd, whence;

long offset;

Системный вызов **lseek** изменяет значение указателя позиции чтения/записи дескриптора **fd** следующим образом: если значение параметра **whence** равно **L_SET**, то указателю присваивается значение параметра, если значение параметра **whence** равно **L_INCR**, значение указателя увеличивается на значение **offset**, если значение параметра **whence** равно **L_XTND**, то указателю присваивается значение (**offset + fsize**), где **fsize**- размер файла.

Следует отметить, что если установить указатель текущей позиции за конец файла, а затем записать что-либо, в файле получается промежуток, который физически не занимает места, а при чтении дает нули.

В случае успешного завершения, вызов **lseek** возвращает значение указателя текущей позиции чтения/записи (целое положительное число), определяющее смещение от начала файла (в байтах).

При возникновении ошибки, возвращается значение -1 и код ошибки в переменной **errno**, которая может принимать следующие значения:

[EBADF] - некорректный дескриптор файла;

[ESPIPE] - дескриптор относится не к файлу, а к программному каналу или файлу типа "гнездо";

[EINVAL] - недопустимое значение параметра.

4.2 Управление оперативной памятью

Изучение возможностей языка программирования C/C++ для обращения к ячейкам оперативной памяти, управления ОП.

Указатели на простые переменные

Указатель - это адрес памяти, распределяемой для размещения идентификатора (в качестве идентификатора может выступать имя переменной, массива, структуры, строкового литерала). В том случае, если переменная объявлена как указатель, то она содержит адрес памяти, по которому может находиться скалярная величина любого типа. При объявлении переменной типа указатель, необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек). Формат объявления указателя: спецификатор-типа [модификатор] * описатель

Спецификатор-типа задает тип объекта и может быть любого основного типа, типа структуры, смеси. Задавая вместо спецификатора-типа ключевое слово **void**, можно своеобразным образом отсрочить спецификацию типа, на который ссылается указатель. Переменная, объявляемая как указатель на тип **void**, может быть использована для ссылки на объект любого типа. Однако для того, чтобы можно было выполнить арифметические и логические операции

над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такие определения типов может быть выполнено с помощью *операции приведения типов*.

В качестве модификаторов при объявлении указателя могут выступать ключевые слова `const`, `near`, `far`, `huge`. Ключевое слово `const` указывает, что указатель не может быть изменен в программе. Размер переменной объявленной как указатель, зависит от архитектуры компьютера и от используемой модели памяти, для которой будет компилироваться программа. Указатели на различные типы данных не обязательно должны иметь одинаковую длину.

Для *модификации размера указателя* можно использовать ключевые слова `near`, `far`, `huge`.

Примеры:

```
unsigned int * a; // переменная a представляет собой указатель
                  // на тип unsigned int (целые числа без знака)
double * x;      // переменная x указывает на тип данных с
                  // плавающей точкой удвоенной точности //
char * fuffer ; // объявляется указатель с именем fuffer,
                  // который указывает на переменную типа char
double nomer;
void *adres;
adres = & nomer;
(double *)adres ++;
```

Переменная `adres` объявлена как указатель на объект любого типа. Поэтому ей можно присвоить адрес любого объекта (& - операция вычисления адреса). Однако, как было отмечено выше, ни одна арифметическая операция не может быть выполнена над указателем, пока не будет явно определен тип данных, на которые он указывает. Это можно сделать, используя операцию приведения типа `(double *)` для преобразования `adres` к указателю на тип `double`, а затем увеличение адреса.

```
const * dr;
```

Переменная `dr` объявлена как указатель на константное выражение, т.е. значение указателя может изменяться в процессе выполнения программы, а величина, на которую он указывает, нет.

```
unsigned char * const w = &obj.
```

Переменная `w` объявлена как константный указатель на данные типа `char unsigned`. Это означает, что на протяжении всей программы `w` будет указывать на одну и ту же область памяти. Содержание же этой области может быть изменено.

Указатель на величину одного типа может быть преобразован к указателю на величину другого типа. Однако результат может быть не

определен из-за отличий в требованиях к выравниванию и размерах для различных типов.

Указатель на тип `void` может быть преобразован к указателю на любой тип, и указатель на любой тип может быть преобразован к указателю на тип `void` без ограничений. Значение указателя может быть преобразовано к целой величине. Метод преобразования зависит от размера указателя и размера целого типа следующим образом:

- если размер указателя меньше размера целого типа или равен ему, то указатель преобразуется точно так же, как целое без знака;
- если указатель больше, чем размер целого типа, то указатель сначала преобразуется к указателю с тем же размером, что и целый тип, и затем преобразуется к целому типу.

Целый тип может быть преобразован к адресному типу по следующим правилам:

- если целый тип того же размера, что и указатель, то целая величина просто рассматривается как указатель (целое без знака);
- если размер целого типа отличен от размера указателя, то целый тип сначала преобразуется к размеру указателя (используются способы преобразования, описанные выше), а затем полученное значение трактуется как указатель.

Массивы и указатели на массивы

C++ и C интерпретируют имя массива как адрес его первого элемента. Таким образом, если `x` – массив, то выражения `&x[0]` и `x` эквивалентны. В случае матрицы – назовем ее `mat` – выражения `&mat[0][0]` и `mat` также эквивалентны.

C++ позволяет использовать указатели для обращения к разным элементам массива. Когда Вы вызываете элемент `x[i]` массива `x`, то выполняются две задачи: получить базовый адрес массива, т.е. узнать где находится первый элемент массива и использовать `i` для вычисления смещения базового адреса массива. Фактически, если есть указатель `ptr` на базовый адрес массива, т.е.

`ptr = x;`

то можно записать :

адрес элемента `x[i]` = адрес `x` + `i * sizeof(базовый тип)`

или

адрес элемента `x[i]` = `ptr + i * sizeof(базовый тип)`

В C++ это записывается более коротким образом

адрес элемента `x[i]` = `ptr + i`

При последовательном обращении к элементам массива можно использован метод инкремента/декремента указателя, т.е. операцию `++` и `--`, например

`ptr++`

```
a = * ptr++
```

```
a = * (ptr--)
```

Указатели на структуры

C++ поддерживает объявление и использование указателей на структуры.

Для присвоения адреса структурной переменной указателю того же типа используется тот же синтаксис, что и в случае простых переменных. После того, как указатель получит адрес структурной переменной, для обращения к элементам структуры нужно использовать операцию ->.

Общий синтаксис для доступа к элементам структуры с помощью указателя имеет вид

```
structPtr -> aMember
```

где structPtr –указатель на структуру, aMember – элемент структуры.

Пример:

```
struct point
```

```
{
```

```
double x;
```

```
double y;
```

```
};
```

```
point p;
```

```
point* ptr = & p;
```

```
ptr->x = 23.3;
```

```
ptr->y = ptr->x + 12.3;
```

Динамическое распределение ОП

4.3 Операции new и delete

Существует много приложений, в которых необходимо создавать новые переменные и динамически распределять для них память во время выполнения программы. В языке C для этого имеются функции динамической памяти, такие как malloc, calloc, free. В C++ введены новые операции - new и delete, которые лучше контролируют тип создаваемых динамических данных. К тому же эти операции работают с конструкторами и деструкторами (раздел объектно-ориентированное программирование).

Операции new и delete имеют следующий синтаксис:

```
указатель = new тип;
```

```
delete указатель;
```

Операция new возвращает адрес динамически распределенной переменной. Операция delete освобождает динамически распределенную память, на которую ссылается указатель.

Если динамическое распределение памяти с помощью операции new потерпело неудачу, оно выбрасывает исключение типа xalloc, объявленное в заголовочном файле EXCERPT.H, поэтому часто динамическое распределение памяти осуществляется внутри блока try. Пример:

```

try
{
    int *pint;
    pint = new int;
    *pint = 33;
    cout << "Указателю выделена память и в ней хранятся данные" << *pint
<< endl;
    delete pint;
}
catch (xalloc&)
{
    cout << "Ошибка при выделении памяти" << endl;
}

```

Сигналы

Сигналы — это программные прерывания, предоставляющие механизм для обработки асинхронных событий. Такие события могут происходить из-за пределов системы — например, из-за введения пользователем символа прерываний (обычно Ctrl+C) — или возникать вследствие действий в программе или ядре, например, когда процесс исполняет код, в котором выполняется деление на ноль. В виде примитивной формы взаимодействия между процессами (interprocess communication, IPC) один процесс также может отправлять сигналы другому процессу.

Самое главное в сигналах — это не только то, что события происходят асинхронно, например пользователь может нажать Ctrl+C в любой момент выполнения программы, но и то, что программа асинхронно обрабатывает сигналы. Функции обработки сигналов регистрируются в ядре, которое асинхронно вызывает функции из оставшейся части программы, как только сигналы доставляются.

Сигналы существуют в Unix с самого рождения системы. Со временем, однако, они эволюционировали, наиболее значительно — в терминах надежности, поскольку когда-то сигналы могли теряться, и в терминах функциональности, поскольку теперь сигналы могут переносить данные, определяемые пользователем. Вначале в разных системах Unix вносились несовместимые друг с другом изменения в сигналы. К счастью, на помощь пришел стандарт POSIX, в котором обработка сигналов была стандартизирована. Именно этот стандарт обеспечивается в Linux и именно о нем мы будем говорить далее.

Наиболее значимые приложения взаимодействуют с сигналами. Даже если вы намеренно разработаете свое приложение таким образом, чтобы оно не полагалось на сигналы в своих коммуникационных нуждах — а это часто оказывается хорошей идеей! — в определенных случаях вам все равно придется работать с сигналами, например, при обработке завершения программы.

4.4 Демоны

Важную роль в работе операционной системы играют системные демоны. Демоны — это неинтерактивные процессы, которые запускаются обычным образом - путем загрузки в память соответствующих им программ (исполняемых файлов) и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы и обеспечивают работу различных подсистем UNIX; системы терминального доступа, системы печати, системы - сетевого доступа и сетевых услуг и т. д. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают пока тот или иной процесс запросит определенную услугу, например, доступ к файловому архиву или печать документа. Возможность терминального входа пользователей в систему, доступ по сети, использование системы печати и электронной почты, — все это обеспечивается соответствующими демонами. Некоторые демоны работают постоянно, пример такого демона — процесс `init()`, являющийся прародителем всех прикладных процессов в системе. Другими примерами являются `cron()`, позволяющий запускать программы в определенные моменты, времени, `inetd()`, обеспечивающий доступ к сервисам системы из сети, и `sendmail()`, обеспечивающий получение и отправку электронной почты.

На рисунке 2 приведен основной алгоритм неинтерактивной программы. В ней вызываются функции, необходимые для выполнения программы, отслеживающей запущенные процессы. Данная функция организывает собственную группу и сеанс, не имеющие управляющего терминала. Так как лидером группы и сеанса может стать процесс, если он еще не является лидером, а предыстория запуска данной программы неизвестна, необходима гарантия, что процесс не является лидером. Для этого порождает дочерний процесс. Так как его PID уникален, то ни группы ни сеансы с таким идентификатором не существует, а значит, нет и лидера. При этом родительский процесс немедленно завершает выполнение, поскольку он уже не нужен. Существует еще одна причина необходимости порождения дочернего процесса. Если демон был запущен из командной строки командного интерпретатора `shell` не в фоновом режиме, последний будет ожидать завершения выполнения демона, и таким образом, терминал будет заблокирован. Порождая процесс и завершая выполнение родителя, имитируется для командного интерпретатора завершение работы демона, после чего `shell` выведет свое приглашение. Далее закрываются открытые файлы, и выполняется запись в системном журнале. Для этого сначала устанавливаются опции "ведения журнала" каждая запись будет предваряться идентификатором PID демона. Далее закрываются открытые файлы, и выполняется запись в системном журнале. Для этого сначала устанавливаются опции "ведения журнала" каждая запись будет предваряться идентификатором PID демона. При

невозможности записи в журнал сообщения будут выводиться на консоль. Каждые десять секунд вызывается функция, реализующая запись запущенных процессов в файл, например, `sysmag.txt`

4.5 Язык PostScript

Язык PostScript не является языком программирования, он используется для описания страниц, подготовленных для печати на принтерах. Эти страницы могут включать не только текст, но и графические объекты, причем в самых разнообразных комбинациях. Вместе с тем PostScript обладает вычислительными возможностями, работает со строками текста, поддерживает ввод/вывод. Все графические объекты в языке PostScript, а так же шрифты являются векторными. В векторной графике положение и размеры объекта не зависят от размера точки изображения на конкретном устройстве печати и размера области изображения. Векторные изображения легко масштабируются, сдвигаются и поворачиваются. Изображения переводятся в растровый формат устройства вывода непосредственно перед печатью, что обеспечивают высокое качество печати. Координаты точек на странице задаются вещественными числами в полиграфических единицах длины - пунктах.

4.5.1 Элементы языка PostScript

PostScript-файл состоит из слов, которые отделяются друг от друга пробелами, символами табуляции, специальным символом конца строки и некоторыми другими специальными символами.

В словах кроме обычных, алфавитно-цифровых символов можно использовать специальные символы. Специальными символами являются скобки трех видов (,), [,], { и } символы <, >, / и X.

Строка может содержать любое количество слов и любое слово может быть первым в строке. Исходный текст PostScript-файла может располагаться (форматироваться) так, чтобы подчеркнуть его логическую структуру, сделать удобочитаемым либо уменьшить размер файла. В PostScript используется обратная (польская) запись, при которой сначала указываются операнды, а затем оператор, выполняющий с ними действия.

Использование польской записи объясняется тем, что все операции в PostScript выполняются над операндами в стеке. Стек представляет собой специальную временную область памяти, организованную по принципу «последний вошел — первый вышел». Команда выбирает последнее значение, удаляя его из стека. После этого текущим становится значение, бывшее ранее предпоследним.

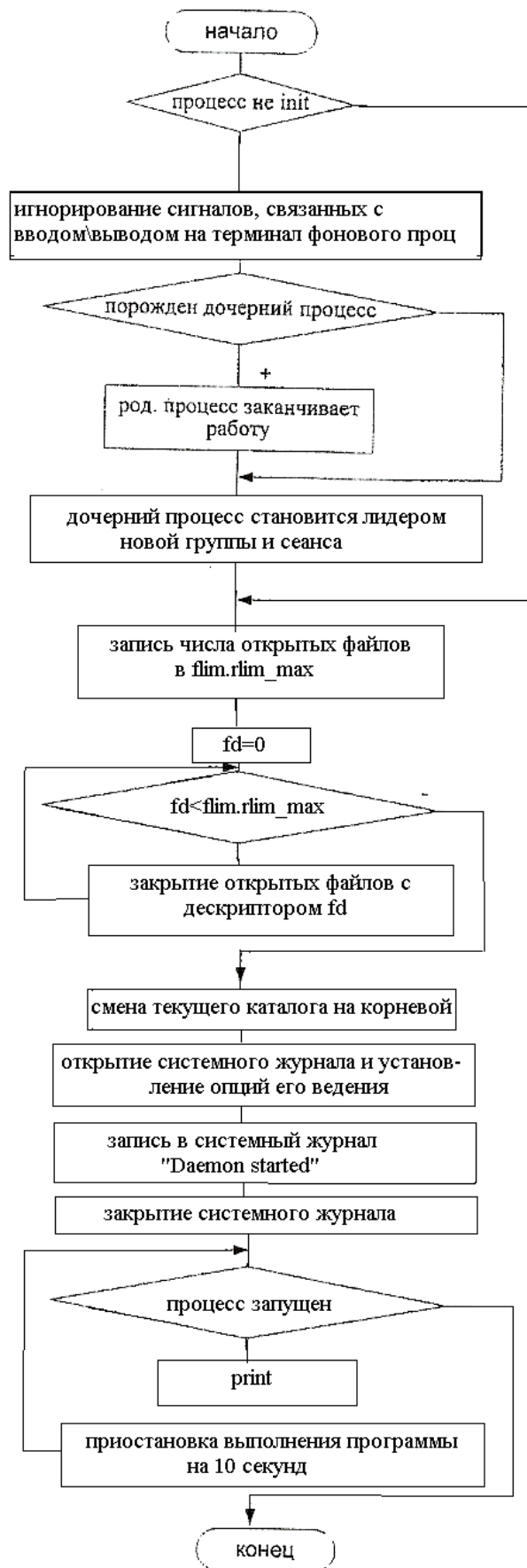


Рисунок 2 Основной алгоритм программы

Таким образом, находящиеся в стеке значения извлекаются из него в обратной последовательности. Интерпретатор PostScript последовательно считывает файл, интерпретируя очередное слово. Если слово представляет собой команду, она немедленно выполняется, а если это некоторый объект данных, он заносится в стек.

4.5.2 Типы данных

Основные типы данных языка PostScript:

- целое число со знаком;
- вещественное число;
- логическое значение;
- строка символов;
- процедура;
- массив произвольных объектов, в том числе и разнотипных;
- словарь.

Целые константы задаются в десятичном формате или в формате без знака \wedge с указанием основания системы счисления. Шестнадцатеричные константы записываются в угловых скобках.

Вещественные **константы** содержат десятичную точку или указатель десятичного порядка, без указателя порядка и с указателем порядка.

Логические константы принимают только два значения, true и false («истина» и «ложь»).

Строка записывается в круглых скобках. Скобки играют роль ограничителей. Строка может содержать пробелы, специальный символ конца строки, любые другие символы, а также экранированные последовательности, начинающиеся символом \.

Процедура — это группа команд, которая решает какую-то частную задачу или используется достаточно часто. В PostScript у процедуры нет ни имени, ни параметров. Процедура имеет тело и ссылку на него, этим она похожа на объекты других сложных типов. Текст процедуры содержится в фигурных скобках. При интерпретации процедуры ее операторы записываются в тело процедуры, а ссылки на нее — в стек операндов. В дальнейшем мы будем говорить, что в стек записывается, процедура. Процедуру, записанную в стек, можно исполнить или записать в переменную, которая в таком случае будет указывать на тело процедуры, играя роль ее имени.

При исполнении процедуры в стек записывается значение, а затем выполняется умножение двух последних чисел из стека. Результат заносится в стек. Процедура записывает в стек операндов одно число, а использует для вычислений два. Первое число должно перед исполнением процедуры находиться в стеке операндов, оно играет роль аргумента. После выполнения процедуры в стеке операндов остается вещественное значение, оно и

возвращается. Процедура может записать в стек несколько значений разных типов.

Во время исполнения интерпретатор PostScript помещает тело процедуры в стек исполнения, а исполнив, удаляет его из этого стека. Если из одной процедуры вызывается другая, первая – приостанавливается, а тело второй заносится в стек исполнения и она становится текущей. После ее окончания и удаления из стека текущей становится вызывавшая процедура, которая продолжает работу с того места, где она была приостановлена. Это обстоятельство, а также метод хранения данных в стеке операндов позволяет использовать рекурсию, когда процедура вызывает на исполнение саму себя.

Массивом в программировании называют совокупность однотипных данных, которой присвоено общее имя. Массив в PostScript может состоять и из разнотипных объектов. Значения элементов массива записываются в квадратных скобках: [1 (String) 1.0].

Символы квадратных скобок являются командами. Команда [записывает в стек операндов специальную метку, а команда] создает массив, содержащий элементы этого стека от текущего значения до ближайшей метки. Элемент, находящийся в стеке сразу после метки, имеет нулевой индекс. Между командами [и] можно исполнять любые команды со значениями, записанными в этот стек.

Словарь является сложным типом данных. Рассмотрим только применение словаря для хранения шрифта и изменения его кодировки. Каждый словарь имеет определенную емкость — максимальное число словарных записей, которое определяется при его создании. Словарная запись состоит из имени и значения. Значением может быть объект любого типа.

Команды PostScript хранятся в специальных словарях, находящихся в стеке словарей с самого начала, которые не могут быть удалены. Это словарь ошибок, системный и пользовательский словари. Данные в системном словаре и словаре ошибок нельзя изменить, эти словари закрыты для записи. Пользовательский словарь изначально является текущим. Можно создавать новые словари, помещать их в стек словарей и удалять из него. Текущим - является последний словарь, записанный в этот стек.

4.5.3 Структура PostScript-файла

PostScript-файл состоит из трех частей: пролога, тела и эпилога. В прологе содержатся описания подпрограмм, и данные необходимые для печати документа. В теле PostScript-файла содержатся только команды, формирующие страницы документа. Эпилог не содержит никаких команд.

Также важную роль при написании программы имеют комментарии, так как по комментарию, расположенному в первой строке файла, ОС UNIX и прикладные программы определяют его тип.

Структурные комментарии используются для выделения элементов логической структуры PostScript-файла и интерпретируются программами просмотра/преобразования.

Структурные комментарии делятся на три группы:

- комментарии в затловке программы (перед прологом);
- комментарии в теле программы;
- завершающие комментарии (в эпилоге).

Для принтера, печатающего файл, комментарии не играют никакой роли, подобно тому, как комментарии в обычном языке программирования не обрабатываются компилятором. Комментарий начинается символом % и продолжается до конца строки.

Для того чтобы подсистема печати или прикладная программа смогли правильно определить тип файла, его первая строка должна начинаться комментарием: %!.

Данный комментарий является обязательным. После символов %! в той же строке может следовать текст PS-Adobe-N.M. Цифры N.M (например, 1.0) определяют версию соглашения о структурных комментариях (это специальный документ, содержащий правила оформления структурных комментариев).

Структурные комментарии начинаются символами %%. Если программа содержит такие комментарии, она соответствует договоренностям о структуре PostScript-файлов. После %%- без пробела следуют зарезервированные (ключевые) слова. Регистр букв в них имеет значение. При необходимости с ключевым словом может быть указано значение, которое отделяется от ключевого слова двоеточием и пробелом.

Комментарий %% DocumentFonts позволяет программам обработки загружать в принтер необходимые шрифты, если они еще не содержатся в его памяти. В комментарии %% PageFonts перечисляются только те шрифты, которые используются при печати данной страницы.

Комментарий %%BoundingBox определяет положение изображения на странице. Четыре целых числа — это координаты левого нижнего и правого верхнего углов прямоугольника, содержащего изображение. В многостраничном документе прямоугольник должен охватывать все изображения. Этот комментарий важен для программ, которые включают изображение из одного файла в другой файл. Если точно определить размер изображения невозможно, следует указать заведомо большие значения, например соответствующие размеру листа бумаги формата A4:

```
%%BoundingBox: 0 0 595 842
```

Комментарием %%EudProlog завершается раздел описаний PostScript-файла и начинается тело документа.

Комментарий %%Page открывает ту часть файла, которая описывает печать очередной страницы. У него два аргумента — метка страницы и ее номер.

Метка представляет собой последовательность любых символов, кроме пробела, и на печать не выводится. Номер страницы задается числом, начиная с 1.

Комментарий `%%PageFonts`, если он имеется, должен идти сразу после `%%Page`. Очередной комментарий `%%Page` обозначает конец предыдущей страницы и начало следующей. Конец последней страницы отмечается комментарием `%%Trailer`.

В языке PostScript нет четкого деления файла с описанием документа на раздел описания и раздел операторов (тело программы), как, например, в языках программирования Pascal или Фортран. Определения переменных в PostScript могут чередоваться с командами формирования изображения.

Текст в PostScript тоже является изображением. Важно лишь, что правильно составленная программа должна иметь правильную структуру. Определения общих для всех страниц документа переменных и процедур следует выносить в пролог.

На рисунке 3 представлена структура программы, написанная на языке описания страниц – PostScript

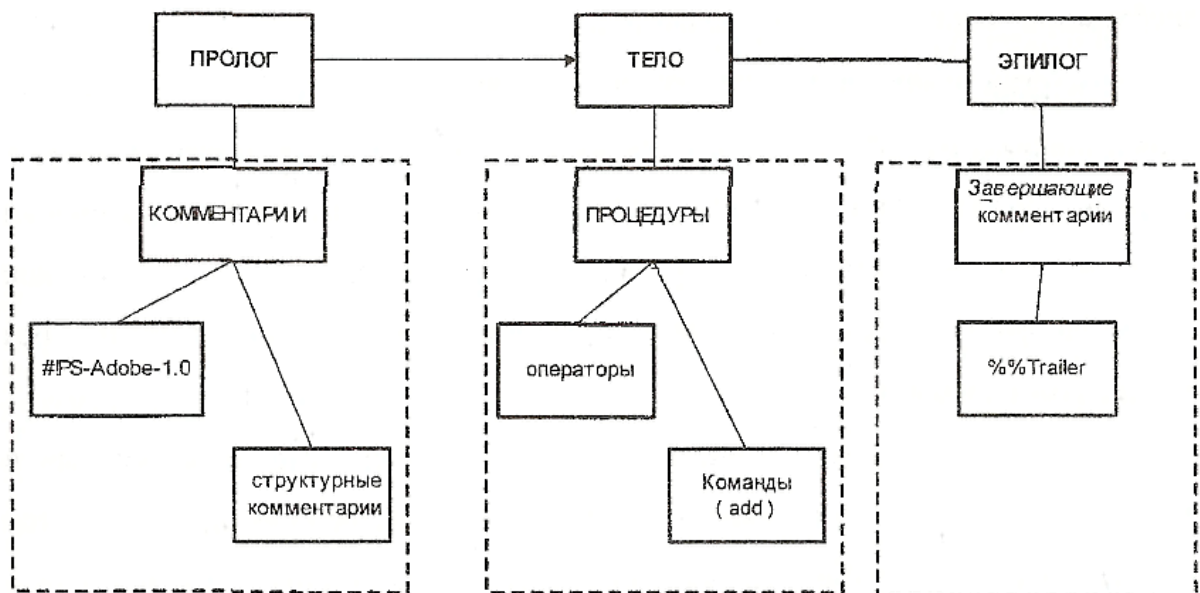


Рисунок 3 Структура программы

4.5.4 Обзор универсальных команд

Для описания команд используется следующая нотация:

*аргумент1, аргумент2, ... команда @ результат1, результат2, ...
%комментарий*

Имя команды выделяется специальным шрифтом. Слева от имени описывается состояние стека до исполнения команды, а справа от стрелки — после ее исполнения. Минус вместо аргумента или результата обозначает отсутствие значения. Данные обозначаются словами, производными от названии типов или математических обозначений, характеризующих

операцию. Такая запись приближена к записи текста программы. Например, описание операции сложения выглядит так:

p q add -> p+q

Запись команды в тексте PostScript-файла может быть такой:

4M 5.0 add

или

1710 add

Целые числа обозначаются буквами *i, j, n*, вещественные числа — *x, y, z*, логические (булевы) значения — буквой *b*, координаты — *x, y*. Буквами *p, q, г, s* обозначаются любые числа, как целые, так и вещественные если их тип не имеет значения. Строки, массивы и процедуры указываются вместе со скобками, в которых они определяются.

PostScript имеет команды общего назначения для работы с данными (присваивание, работа со стеком)^операторы вызова подпрограмм, ветвления и циклы. Операторов безусловного перехода и меток нет. Имеются команды ввода/вывода> но при печати на принтере они теряют смысл, так как PostScript-файл передается в принтер и обрабатывается в нем автономно, без связи с компьютером.

При интерпретации команды интерпретатор PostScript ищет ее имя в стеке словарей. Если имя найдено, соответствующее значение либо заносится в стек операндов (если это не процедура), либо исполняется (если это процедура). Если имя не найдено ни в одном из словарей, возникает ошибка и интерпретация PostScript-файла завершается. Имена стандартных команд представляют собой имена процедур, записанные в системном словаре. При необходимости эти имена можно переопределить. Команда *load* выполняет поиск указанного имени, но в любом случае заносит найденное значение в стек. Эту команду можно применять для того, чтобы использовать описанную ранее процедуру в качестве тела цикла или условного оператора.

Условный оператор *if* удаляет из стека операндов процедуру и логическое значение, после чего, либо исполняет процедуру,, либо нет. Процедура может использовать значения в стеке операндов, записанные до того, как туда было записано логическое значение. Условный оператор *ifelse* удаляет из стека операндов обе процедуры и логическое значение, после чего исполняет одну из процедур. В этом случае обе процедуры должны одинаково использовать старые значения в стеке и оставлять там одинаковое количество новых значений.

Оператор цикла *for* удаляет из стека операндов все свои 4 аргумента и устанавливает значение параметра цикла равным $p_{нач}$. После этого он циклически выполняет следующие действия: проверяет, не вышло ли текущее значение параметра за допустимую границу $s_{конj}$ помещает значение параметра цикла в стек операндов, выполняет процедуру, увеличивает значение параметра на величину $q_{шаг}$. Шаг может быть как положительным, так и отрицательным. Если начальное значение больше конечного при положительном шаге ИЛИ меньше

конечного при отрицательном шаге, процедура не выполняется. Процедура может использовать предыдущие значения в стеке, а также значение параметра цикла. После окончания цикла стек не восстанавливает автоматически свое прежнее состояние, что может привести к его переполнению, поэтому, в частности, необходимо удалять из стека операндов очередное значение параметра цикла.

В таблице 1 приведен обзор некоторых универсальных команд.

4.5.5 Графический контекст

В операторах вывода графики используются неявные параметры, влияющие на их выполнение. Совокупность этих параметров называется графическим контекстом. Графический контекст составляют положение, ориентация и масштаб системы координат, толщина и стиль рисования линии, область рисования, текущий путь построения линии и некоторые другие. В PostScript имеются команды для работы с графическим контекстом.

Среди них команды, которые заносят в специальный стек графических контекстов текущий графический контекст и могут восстановить ранее записанное состояние. Это позволяет изменять параметры вывода одной части рисунка и возвращаться к предыдущему состоянию перед выводом другой его части. Команда `gsave` выполняет запись в стек графических контекстов, а команда `grestore` считывает из этого стека графический контекст, делая его текущим. Таким образом, все изменения параметров рисования, произведенные с момента сохранения контекста, перестают действовать на последующие команды вывода графических объектов. Команда `grestoreall` восстанавливает самое первое состояние, записанное в стек командой `gsave`. Более «мощные» команды `save` и `restore` кроме сохранения графического контекста сохраняют еще и область памяти виртуальной ЭВМ, содержащую все переменные, и восстанавливают как графический контекст так и значения переменных. Эти команды можно использовать для предотвращения побочных эффектов при печати страниц, если среди команд печати встречаются команды изменения каких-либо объектов. Команда `grestoreall` восстанавливает последний графический контекст, сохраненный командой `save`, или, если таких команд не было, самый первый (верхний в стеке), сохраненный командой `gsave`. Образ памяти, создаваемый командой `save`, рекомендуется хранить в стеке операндов и использовать команду `restore`, предварительно убрав из стека все записанные чуда .позже операнды.

Команды `save/restore` рекомендуется использовать в начале и в конце описания каждой страницы.

Такой стиль программирования позволит в начале каждой страницы получать тот графический контекст и переменные, которые были определены в прологе программы. Использование в начале/конце описания страниц команд `gsave/grestore` позволяет сохранять графический контекст, но не освобождает память от определенных, но уже не используемых значений. Эти

команды выполняются быстрее, чем save/restore, поэтому при печати больших файлов они могут оказаться более эффективными. В таблице 2 приведены команды для работы с графическими объектами.

Таблица 1 Перечень команд

Описание команды	Примечание
$p \ q \ \text{add-} \rightarrow p+q$	-
$p \ q \ \text{sub-} \rightarrow p-q$	-
$p \ q \ \text{mul-} \rightarrow p*q$	-
$p \ q \ \text{div-} \rightarrow p/q$	-
$ij \ \text{idiv-} \rightarrow [i/j]$	Целая часть от деления. Аргументы только целого типа.
$i \ j \ \text{mod-} \rightarrow i \bmod j$	Остаток от деления. Аргументы только целого типа.
$p \ \text{neg-} \rightarrow -p$	Изменение знака
$p \ \text{abs-} \rightarrow p $	Модуль числа -
$p \ \text{cv.i-} \rightarrow i$	Преобразование к целому типу. У вещественного числа отбрасывается дробная часть.
$p \ \text{cvt-} \rightarrow x$	Преобразование к вещественному типу.
$p \ \text{round-} \rightarrow q$	Округление числа. Тип результата совпадает с типом аргумента.
(строка) $\text{cvi-} \rightarrow i$	Преобразование символьного представления числа Γ^1 в целое
(строка) $\text{cvt-} \rightarrow x$	Преобразование символьного представления числа в вещественное
	Функции
$a^\theta \ \sin \rightarrow \sin(o:D)$	Угол задается в градусах
$P \ \text{sqrt-} \rightarrow \sqrt[p]{p}$	Квадратный корень
$p \ \ln \rightarrow \ln(p)$	Натуральный логарифм
$p \ \log \rightarrow \log(p)$	Десятичный логарифм
$\text{-rand-} \rightarrow i$	Генератор псевдослучайных чисел в диапазоне от 0 до 231-1
Отношения	
$P \ q \ \text{eq-} \rightarrow p=q$	В стек помещается логическое значение true или false
$p \ q \ \text{ne-} \rightarrow p \neq q$	
$p \ q \ \text{le-} \rightarrow p \leq q$	
$p \ q \ \text{lt-} \rightarrow p < q$	

P-q #->p>q	
Логические операции	
bTb2and- >blAb2	Логическое И
bl b2or- >blvb2	Логическое ИЛИ
Ы Ь2 хог- >M=Ь2	Исключающее-ИЛИ
bnot->£	Логическое отрицание

4.5.6 Рисование и закрашка фигур

Отличительным аспектом PostScript является то, что даже текст - это разновидность графики. Первой задачей будет рисование линий для создания изображения.

Основные шаги рисования и закрашки фигур:

- Начать путь оператором `newpath`;
- Собрать путь из отрезков и кривых (не обязательно смежных);
- Нарисовать линию оператором `stroke` или закрасить оператором `fill`.

Эта последовательность действий может быть изменена для получения более сложных результатов.

Рисование прямоугольника

Нарисуем прямоугольник на расстоянии в дюйм от сторон левого нижнего угла страницы. Начнем с функции, переводящей дюймы в единицы измерения PostScript - пункты (один пункт равен 1/72 дюйма). Осуществить такое преобразование просто - достаточно умножить число дюймов на 72:

```
/inch {72 mul} def
```

Начинаем новую линию и помещаем текущую точку на расстояние в дюйм от границ:

```
newpath
```

```
/ inch / inch rnoveto
```

К этому моменту линия состоит из одной точки с координатами (-72, 72). Добавим стороны с помощью оператора `lineto`. Этот оператор добавляет к пути отрезок, соединяющий текущую точку и точку, координаты которой находятся на стеке. Координаты конца отрезка становятся новыми координатами текущей точки.

Итак, добавим три стороны квадрата:

```
2 inch I inch lineto
2 inch 2 inch lineto
1 inch 2 inch lineto
```

Получившуюся линию можно замкнуть кратчайшим отрезком. Это делается оператором `closepath`. Этот оператор особенно полезен при закрашке фигур.

Теперь полученную линию можно нарисовать оператором `stroke`. Оператор `showpage` закончит вывод страницы на печать:

Closepath

Stroke

showpage

Закраска фигур

Таблица 2 Команды для работы с графическими объектами

Описание команд	Примечание
Сохранение и восстановление графического контекста	
<code>-gsave-></code>	Запись текущего графического контекста в стек графических контекстов
<code>-grestore-></code>	Восстановление текущего графического контекста из сохраненного командой <code>gsave</code>
<code>-save-></code> образ памяти метка	Сохранение текущего графического контекста в стеке и запись в стек операндов образа памяти
образ памяти <code>restor-></code>	
Изменение системы координат	
<code>x y translated</code>	Перенос начала координат
<code>a°rotate-></code>	Поворот координатных осей
<code>Sx Sy scale-></code>	Изменение масштаба осей координат
Создание графического пути	
<code>-newpath-></code>	инициализация нового графического пути
<code>-closepath-></code>	Замыкание участка графического пути из текущей точки в начальную точку пути
<code>x y moveto-></code> .	Установка текущей точки без проведения линии ;
<code>-currentpoint-></code> <code>x y</code>	Определение координат текущей точки
<code>x y lineto-></code>	Добавление отрезка прямой линии
Использование графического пути для вывода фигур	
<code>-stroke-></code>	Построение линии вдоль графического пути

-fill->	Закрашивание текущим цветом фигуры, ограниченной замкнутым графическим путем
-pathbbox->	Определение координат прямоугольника, охватывающего текущий путь
-clippath->-	Создание нового графического пути вокруг всех элементов

Сначала создается путь, но вместо вызова оператора stroke вызывается оператор fill, который заполняет путь текущим цветом. Применение fill вместо stroke в приведенном примере даст закрашенный квадрат вместо контура.

Вставка текста

Вставка текста состоит из следующих основных шагов:

- Выбрать необходимый шрифт;
- Сделать текущей точку, в которую будет помещен левый нижний угол текста;
- Передать строку для печати оператору 'show'.

Оператор 'show' - это простейший оператор для вывода строки. Его аргументом является строка, которую он выводит текущим шрифтом. Вывод происходит, начиная с текущей точки, которая становится левой нижней точкой по отношению к тексту. После того как текст был выведен, текущей становятся точка соответствующая нижнему правому краю строки.

Ниже приведен текст программы для вывода графика функции $f=\sin(x)^2$

```
% IPS-ADOBE-1.0
%%Title: график функции  $f=\sin(x)^2$ 
%%Creator: Иванова Анна
%%Pages: 1
%%BoundingBox: 0 0 595 842
%%EndComments
/cm { 72.0 mul 2.54 div } def %перевод см в пункты /x0 21.0 2 div 5.0 sum -cm def
^координаты нижнего /y0 29,7 2 div 2.0 sum era def %левого угла графика
%%EndProlog
%%Page: 1 1
gsave
x0 y0 translate ^сдвигаем начало координат к рамке
newpath
0 0 moveto %выводим рамку размером 10см x 4см
10 cm 0 rlineto
0 4 cm rlineto
```

```

~10 cm 0 rlineto
closepath
.3 setlinewidth %толщина рамки 0.3 пункта
stroke %рисуем линию вдоль сторон рамки
newpath
0 8 1440 %заголовок цикла: 0a? 0°до8
{
/a exch def %локальная переменная
a 144.0 div %стек: это число в диапазоне 0...10см ст %с^ек: x координата
лежит в диапазоне %0.10см
a sin
a 2 div 1.0 add
2-0 mul -
см %стек: x y %координата у лежит в %интервале G...4 см
a 0 eg
{ moveto } %переходишь в первую точку % стек:x y
{ lineto } ifelse
} for % оператор цикла
1 setlinewidth % график выводится жирной линией
stroke -% рисует линию вдоль прямой
grestore
showpage
%%Trailer

```

На рис. 4 приведен результат выполнения программы.

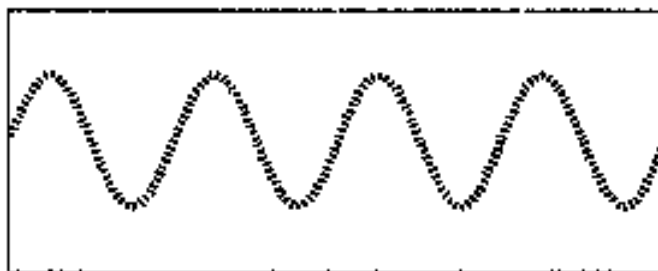


Рисунок 4 – Вид окна с результатами выполнения программы

4.6 Командный интерпретатор

Командный интерпретатор является одной из важнейших программ, обеспечивающих диалог пользователя с системой. Он запрашивает у пользователя команду и анализирует ее. Если команда является внутренней по отношению к командному интерпретатору, то он реализует ее своими средствами (например, команда смены директории - `cd` - реализуется функцией `cd()`). Если же введенная команда не является внутренней, он запускает эту команду на выполнение (функция `execvp()`). В случае некорректной команды, выводится сообщение об ошибке.

Все современные системы UNIX поставляются по крайней мере с тремя командными интерпретаторами: Bourne shell (`/bin/sh`), C shell (`/bin/csh`) и Korn shell (`/bin/ksh`). Существует ещё несколько интерпретаторов, например Bourne-Again shell (`bash`), со сходными функциями.

В UNIX реализуется следующий сценарий работы в системе:

- При включении терминала активизируется процесс `getty` (M), который является сервером терминального доступа и запускает программу `login(1)`, которая, в свою очередь, запрашивает у пользователя имя и пароль.
- Если пользователь зарегистрирован в системе и ввёл правильный пароль, `login(1)` запускает программу, указанную в последнем поле записи пользователя в файле `/etc/passwd`. В принципе это может быть любая программа, но в нашем случае - это командный интерпретатор `shell`.
- `Shell` выполняет соответствующий командный файл инициализации, и выдаёт на терминал пользователя приглашение. С этого момента пользователь может вводить команды.
- `Shell` считывает ввод пользователя, производит синтаксический анализ введённой строки, подстановку шаблонов и выполняет действие, предписанное пользователем (это может быть запуск программы, выполнение внутренней функции интерпретатора) или сообщает об ошибке, если программа или функция не найдены.
- По окончании работы пользователь завершает работу с интерпретатором, вводя команду `exit`, и выходит из системы.

Основной алгоритм программы, реализующей функции командного интерпретатора, представлен на рисунке 5. В ней осуществляется вывод на экран строка с - текущей директорией и приглашением командного интерпретатора, запрашивающим команду. После ввода пользователем команды, вызывается функция (`translate`), которая делит введенную команду на имя команды и ее аргументы, возвращая при этом константу в соответствии с именем команды. Потом с помощью оператора `switch` анализируется возвращенная константа и выполняются соответствующие действия.

`Translate` - функция разбора строки команды, введенной пользователем с клавиатуры, на имя команды и ее аргументы. Функции передается параметр ~

строка команды, возвращаемые значения - имя команды в переменной *command*, список аргументов в массиве *p* и константа, определяющая команду.

Алгоритм функции *translate* представлен на рисунке 6.

Sozd - функция, выполняющая запуск внешней команды. Функция создает дочерний поток, в котором выполняется команда, возвращает результат выполнения команды. В случае если команда не найдена, выдается сообщение об ошибке.

Алгоритм функции *sozd* представлен на рисунке 7.

F - функция выполняет действия аналогичные команде *cat>1.txt*. являющейся внешней, т.е, производит запись введенной информации в файл. Весь ввод с консоли направляется в файл *1.txt*. Для вызова функции следует нажать клавишу «f», для завершения записи в файл нажать сочетание клавиш «Ctrl+Z», Файл создается с именем *1.txt* в текущей директории.

Алгоритм функции *F* представлен на рисунке 8.

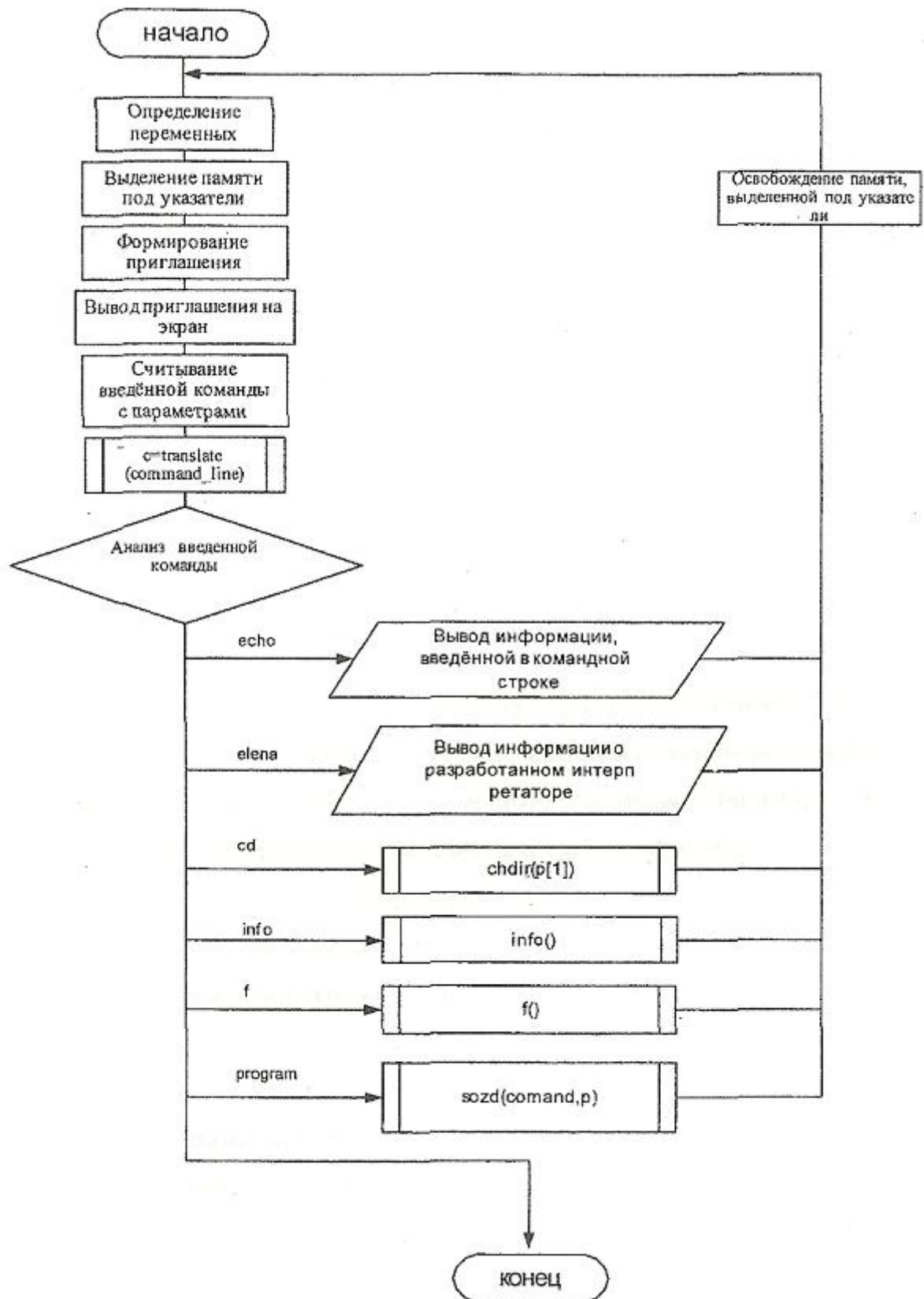


Рисунок 5. Основной алгоритм программы

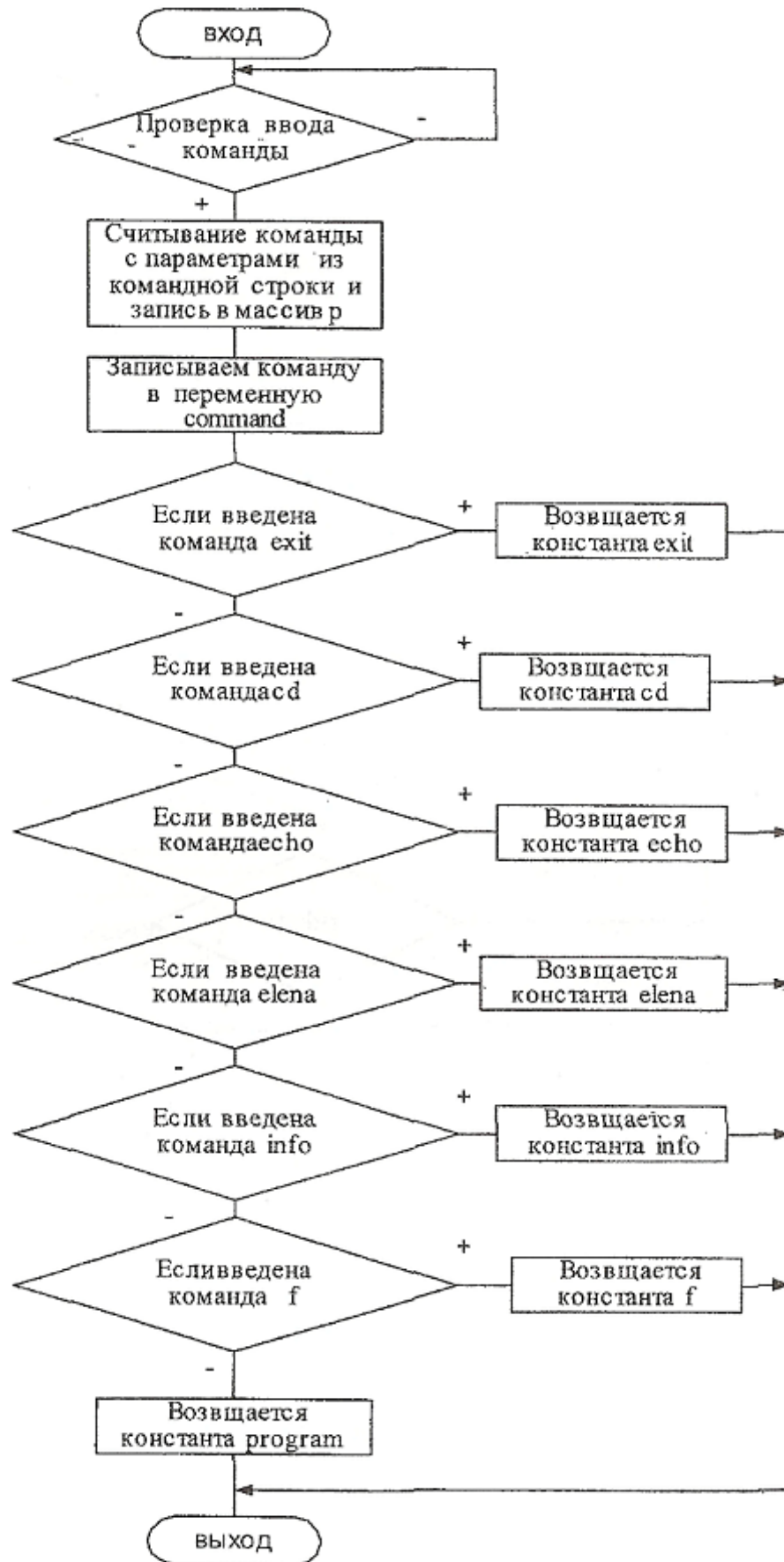


Рисунок 6 Алгоритм функции translate

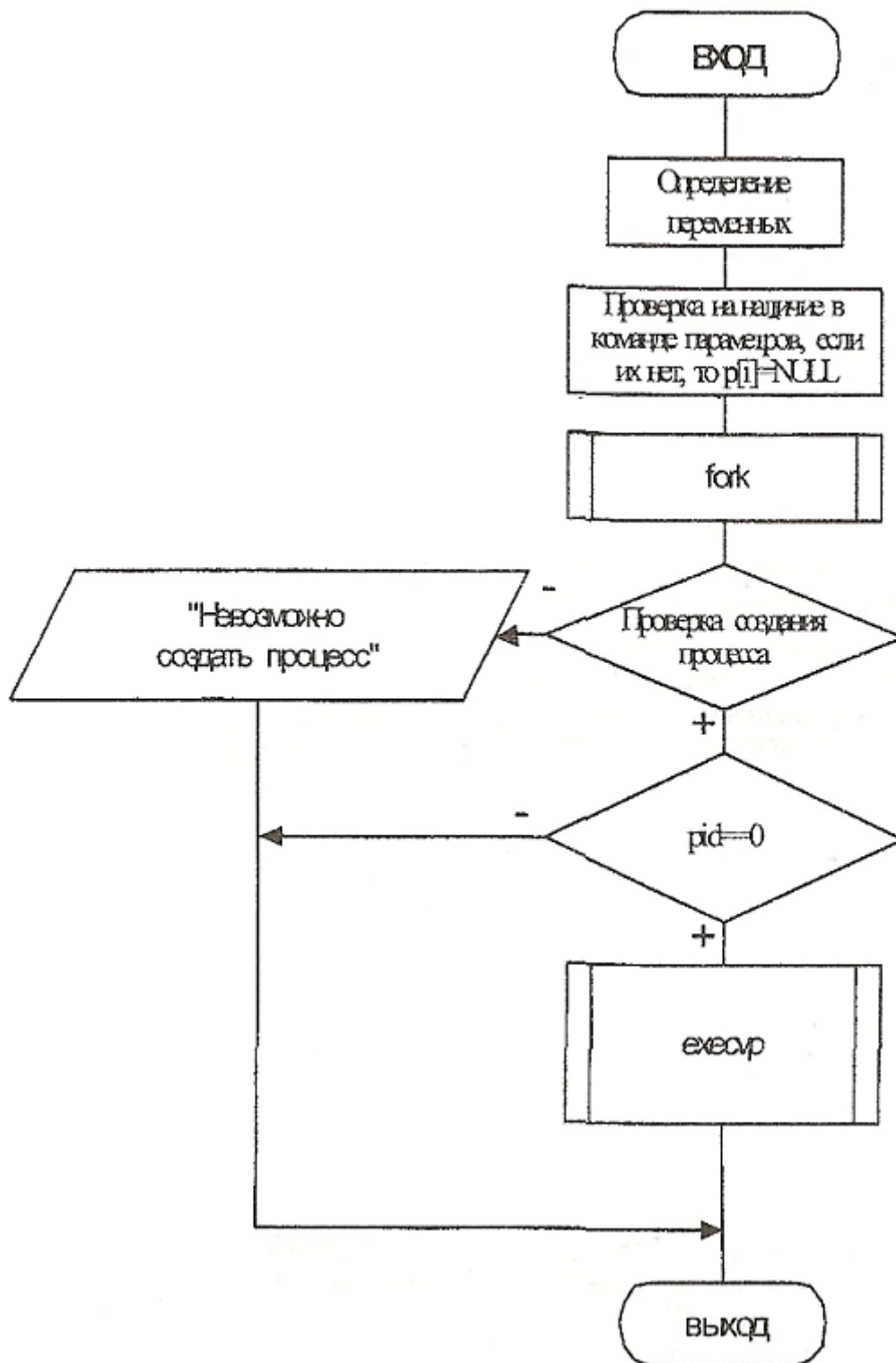


Рисунок 7. Алгоритм функции sozd

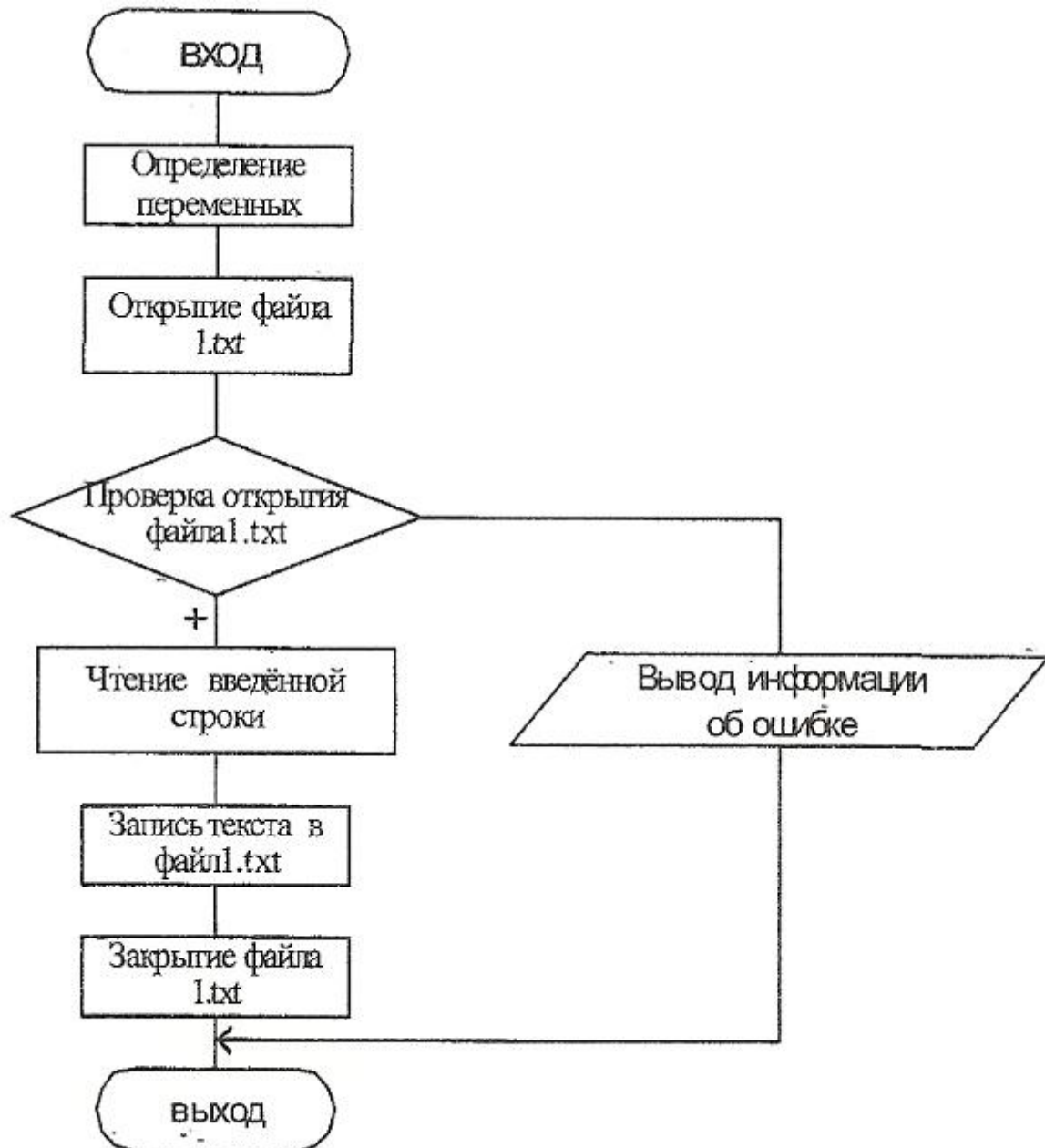


Рисунок 8. Алгоритм функции F

ПРИЛОЖЕНИЕ А

Министерство образования и науки
Российской Федерации
Московский Государственный Технический Университет
«МАМИ»

Кафедра «Автоматика и процессы управления»

Курсовая работа защищена с оценкой
(подпись преподавателя, дата)

КУРСОВАЯ РАБОТА
по дисциплине «Системное программное обеспечение»
Вариант №14

Тема: «Разработка командного интерпретатора»

Курсовая работа допущена к защите
(подпись преподавателя, дата)
Выполнила ст. группы

(Ф.И.О.)

Руководитель:
доцент, к. т. н., Мурачев Е.Г.
(звание, степень Ф.И.О.)

ПРИЛОЖЕНИЕ Б

Министерство образования и науки
Российской Федерации
Московский Государственный Технический Университет
«МАМИ»

Кафедра «Автоматика и процессы управления»

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине:
«Системное программное обеспечение»
вариант № 1

Задание

Разработать программу распознавания инфиксных выражений в постфиксную форму. Выражения разделены между собой точками с запятой и состоят из чисел, идентификаторов и операторов +, -, *, /, div, mod.

Задание выдано «___»_____2011 г. (подпись преподавателя)

Задание получил _____ (подпись студента)

ПРИЛОЖЕНИЕ В

Спецификация

Обозначени	Наименование	Примечани
	Документация	
КР021045	Разработка прмраммы	
	вывода графика функции	
	$f=\sin(x)/2$. Текст -	-
	программы.	
КР021045	Разработка программы	
-	вывода графика функции	
	$r=\sin(x)/2$. Руководство	
	оператора-	

ПРИЛОЖЕНИЕ Г

УТВЕРЖДЕН
КР021045 12

Разработка программы для вывода графика функции $f=\sin(x)/2$

Текст программы
К021045 12
Листов 12

МОСКВА 2011

ПРИЛОЖЕНИЕ Д

УТВЕРЖДЕН
КР021045 34

Разработка программы для вывода графика функции $f=\sin(x)/2$

Руководство оператора
К021045 34
Листов 18

ЛИТЕРАТУРА

- 1 Робачевский А. М. Операционная система Unix. - СПб: БХВ-Петербург, 2001.
2. Стахнов А. А. Linux. - СПб.: БХВ-Петербург, 2003.
3. Шоу А. Логическое проектирование операционных систем," Пер. с англ.- М: Мир, 2009
4. Грис Д. Конструирование компиляторов для ЦВМ.-М.: Мир, 2001
5. Бек Л. Введение в системное-программирование.-М.: Мир, 1988
6. Немнюгин С, Чаунин М., Камолкйн А. Эффективная работа: UNIX.- СПб.: Питер, 2003.
7. Андрей Богатырев. «Хрестоматия по программированию на Си в Unix»
8. Теренс Чан. Системное программирование на C++ для Unix. /Под ред. Коломыцева, Киев, 1997.
9. Лав. Linux. Системное программирование. Всё о системных вызовах в Linux. - СПб.: Питер, 2008.

Учебное издание

Иванов Антон Александрович
Матросова Владлена Валентиновна
Мурачев Евгений Григорьевич
Савостьянок Юрий Алексеевич

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Под редакцией авторов

*Оригинал-макет подготовлен редакционно-издательным отделом
МГТУ «МАМИ»*

По тематическому плану внутривузовских изданий учебной литературы
на 2010г., доп.

Подписано в печать 30.06.2010г. Формат 60х90 1/16. Бумага 80г/м
Гарнитура «Таймс». Ризография. Усл. печ. л. 3,5.
Тираж 55 экз. Заказ № 78-10.

МГТУ «МАМИ»
107023, г. Москва, Б. Семеновская ул., 38.