МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА
Факультет
вычислительной математики
и кибернетики

М. А. Казачук, И. В. Машечкин,
И. С. Попов, А. Н. Терехин,
В. В. Тюляева

ПРОГРАММИРОВАНИЕ В ОС UNIX НА ЯЗЫКЕ СИ

Учебно-методическое пособие

2-е издание, исправленное и дополненное

МАКС ПРЕСС

МОСКВА – 2020

УДК 681.3.06(075.8)
ББК 32.973-018.2я73
П78

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ имени М. В. Ломоносова

Рецензенты:
доцент *Е. А. Кузьменкова;*
профессор *А. Н. Томилин*

П78 **Программирование в ОС UNIX на языке Си** : учебно-методическое пособие / М. А. Казачук, И. В. Машечкин, И. С. Попов, А. Н. Терехин, В. В. Тюляева. – 2-е изд. испр. и доп. – Москва : Издательский отдел ВМК МГУ имени М. В. Ломоносова; МАКС Пресс, 2020. – 108 с.

ISBN 978-5-89407-605-5 (ВМК МГУ). – ISBN 978-5-317-06374-0 (МАКС Пресс)

Пособие подготовлено авторами на основании практического опыта использования и преподавания языка Си на факультете ВМК. В данном пособии предложены методические рекомендации по проведению семинарских занятий по программированию, поддерживающих курс лекций «Операционные системы», читаемый в третьем семестре на факультете ВМК МГУ.

При подготовке пособия использованы задачи из «Сборника задач и упражнений по языку Си» Т. В. Руденко.

УДК 681.3.06(075.8)
ББК 32.973-018.2я73

ISBN 978-5-89407-605-5 (ВМК МГУ)
ISBN 978-5-317-06374-0 (МАКС Пресс)

Введение

Пособие подготовлено авторами на основании практического опыта использования и преподавания языка Си на факультете ВМК. В данном пособии предложены методические рекомендации по проведению семинарских занятий по программированию, поддерживающих курс лекций «Операционные системы», читаемый в третьем семестре на факультете ВМК МГУ.

При подготовке ряда примеров и упражнений использованы задачи из «Сборника задач и упражнений по языку Си» Руденко Т.В.([5]).

ТЕМА 1. Краткая характеристика языка. Базовые типы данных и операции с ними

Язык Си – язык программирования высокого уровня, разработанный Д.Ритчи и его коллегами в 70х гг прошлого века. Он представляет собой минималистичный процедурный язык программирования, в котором упор сделан на эффективность компиляции. Многие возможности вынесены из базовых средств языка и предоставляются стандартной библиотекой, при этом прототипы и семантика функций стандартной библиотеки фиксированы и не зависят от конкретного компилятора, вместе с которым она поставляется. Си часто называют «высокоуровневым ассемблером», поскольку он предоставляет возможность писать эффективные программы с весьма низким уровнем абстракции. В то же время Си не привязан к конкретной аппаратуре или операционной системе и предназначен для написания машинно-независимых и легко переносимых программ.

Базовые типы данных

В языке Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах. Язык Си обеспечивает разнообразие типов данных. Кроме базовых типов: **char**, **int**, **float** и **double** имеется возможность получать целую иерархию выводимых типов данных из указателей, массивов, структур и объединений.

Базовые типы данных могут быть использованы с квалификаторами: **short**, **long**, **signed** и **unsigned**. Всегда выполняется соотношение – размер **short** не больше размера **int**, а **int**, в свою очередь, не больше размера **long**. Конкретные размеры типов определяются реализацией компилятора.

Переменные и константы

Имена переменных могут состоять из букв (знак подчеркивания считается буквой) и цифр. Первая литера обязательно буква. Большие и маленькие буквы различаются. Для внутренних имен значимыми являются первые 31 литера, а для внешних – 6 литер.
Все переменные должны быть описаны до использования, например,

```
int a;
long b,c,d;
unsigned char s;
double m[100]; /* m - массив из 100 элементов
                  типа double, индексы принимают
                  значения от 0 до 99 */
```

К любой переменной в декларации может быть применен квалификатор **const**, указывающий на то, что ее значение далее изме-

няться не будет (для массива это означает, что ни один из его элементов не будет меняться)

```
const char warnmsg[] = "warning: ";
```

Особенности записи констант: целая константа, например, 123 имеет тип **int** (или **long int**, если она столь велика, что тип **int** не может вместить ее значение). Целое значение может также иметь восьмеричное или шестнадцатеричное представление. Восьмеричное начинается с нуля (**017**), а шестнадцатеричное начинается с **0x** или **0X** (**0x1a** или **0X1a**). Константа типа **long** завершается буквой **l** или **L**, например, **123L**. Беззнаковые константы заканчиваются буквой **u** или **U**. Константы типа **double** имеют десятичную точку или экспоненциальную часть, например, **123.4** или **1.234e2**. Константы типа **float** заканчиваются буквой **f** или **F**. Константы типа **long double** имеют десятичную точку или экспоненциальную часть и заканчиваются буквой **l** или **L**, например, **123.4L**.

Константа типа **char** записывается в виде литеры, обрамленной одиночными кавычками (**'*', 's'**). Некоторые литеры записываются с помощью заранее определенных эскейп-последовательностей (**'\n'**-новая строка, **'\t'** – табуляция, **'\a'** – сигнал «звонок» и т.д.). Кроме того, произвольный восьмеричный код можно задать в виде **'\ooo'**, где **ooo** – одна, две или три восьмеричные цифры, или **'\xhh'**, где **hh** – одна или две шестнадцатеричные цифры (например, **'\23'** или **'\x12'**).

Строковая константа (строка) – это ноль или более литер, заключенных в двойные кавычки (например, **"asdf"**). Во внутреннем представлении в конце строки присутствует **'\0'**, поэтому, фактически – это массив типа **char** из **n+1** элементов, где **n** – число значащих символов строковой константы.

В Си имеется возможность описания перечислимого типа, для этого используется ключевое слово **enum.** При описании перечислимого типа полностью задается набор его значений:

```
enum palette{white, black=10, red, green=-1,
   blue};
/* white, black, red, green и blue - константы
перечислимого типа palette */
enum palette color;
/*color - переменная типа palette, ей может быть
присвоена любая из констант этого типа*/
color = white;
```

Именование констант перечислимого типа уникально в пределах области видимости. Данные константы ассоциированы с целым типом Си и могут использоваться везде, где используются констан-

ты типа **int**. По умолчанию, константам перечислимого типа присваиваются последовательные значения (0,1,2, ….). Если для константы указано "**= значение**", то ей присваивается это значение, а следующая за ней константа получает значения на 1 больше и т.д. Таким образом, в приведенном примере константы перечислимого типа **palette** имеют следующие значения:

```
white       -    0
black       -   10
red         -   11
green       -   -1
blue        -   0
```

Операции над базовыми типами

В языке Си определены операции следующих типов:

1. Арифметические операции (**+, -, *, /, %**):
   - унарные операции **+** и **-** (изменяет знак стоящей справа величины) имеют более высокий приоритет, чем бинарные операции **+** и **-** ;
   - бинарные операции **+**(сложение) и **–**(вычитание) имеют одинаковый приоритет, он ниже приоритета операций *****(умножение), **/**(деление) и **%**(остаток от деления);
   - операция **/** с операндами целых типов выполняется с усечением дробной части;
   - операция **%** определена только для операндов целых типов;
   - арифметические операции с одинаковым приоритетом выполняются **слева направо.**
2. Логические операции (**!, &&, ||**)
   - результатом унарной операции **!**(логическое НЕ) будет 0 для ненулевого операнда и 1 для нуля;
   - приоритет бинарной операции **&&**(логическое И) выше приоритета бинарной операции **||**(логическое ИЛИ).
3. Операции отношения(**<, <=, >, >=, ==, !=**)
   - операции **<, <=, >, >=** имеют одинаковый приоритет, он ниже приоритета операций сравнения на равенство **==** и **!=** ;
   - операции отношения имеют более низкий приоритет, чем арифметические операции, но более высокий, чем логические операции **&&** и **||**.
4. Побитовые операции (**&, |, ^, <<, >>, ~**)
   - все побитовые операции можно применять только к операндам целых типов (знаковым и беззнаковым **char**, **short**, **int** и **long**);

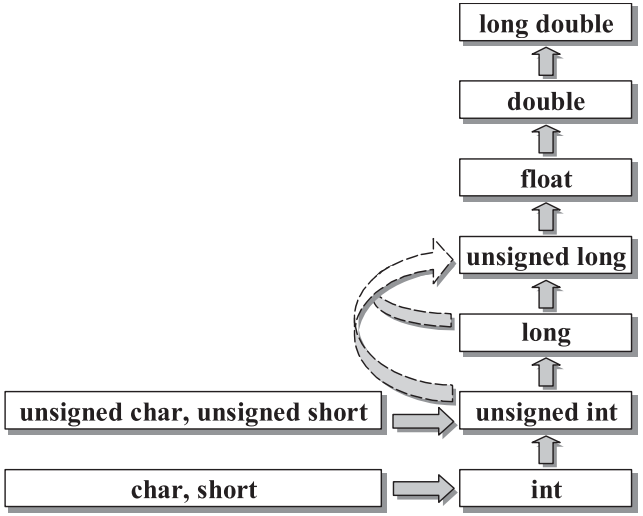
- результат поразрядных операций **&** и **|** не совпадает с результатом логических операций **&&** и **||**, например, для **x=2** и **y=5** результатом **x&y** будет 0, тогда как результатом **x&&y** будет 1;
- операции **<<**(сдвиг влево) и **>>**(сдвиг вправо) могут использоваться как эффективное умножение и деление на степени 2: **x<<n** умножает **x** на 2 в **n**-ой степени, а **x>>n** делит **x** на 2 в **n**-ой степени, если **x** и **n** – неотрицательные числа (в случае, если значение **x** или **n** отрицательно, результат операций сдвига не определен, т.е. зависит от реализации).

5. Операции присваивания (**=,+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=**)
   - типом и значением любой операции присваивания является тип и значение ее левого операнда после завершения присваивания.
6. Инкрементная (**++**) и декрементная(**--**) операции
   - могут быть как префиксными, так и постфиксными, так, например, **++x** увеличивает **x** на 1 до того, как его значение будет использовано, а **x++** – после;
   - эти операции можно применять только к переменным: запись **++(x+y)** не верна.
7. Условная (тернарная) операция (**?:**)
   - в выражении **x?y:z** первым вычисляется значение выражения **x**, и если оно не равно нулю, то результатом всей операции будет значение выражения **y**, иначе – значение выражения **z**.

Полная таблица операций с правилами их выполнения приведена в *Приложении 1*.

Приведение типов

В выражении могут быть использованы операнды различных типов. При вычислении любой бинарной операции над операндами различных типов предварительно осуществляется автоматическое приведение к объемлющему типу в соответствии со следующей схемой:



Примечание. Читать схему следует следующим образом: «если один из операндов имеет тип **long double**, то другой операнд приводится к **long double**» и т.д. Особого внимания требует случай, когда один из операндов имеет тип **long**, а другой – **unsigned int**: если представление типа **long** включает все значения типа **unsigned int**, то **unsigned int** приводится к **long**, если же это не так – оба операнда приводятся к **unsigned long** (эта особенность вызвана тем, что стандарт языка Си не накладывает ограничений на машинное представление переменных, кроме того, что машинное представление переменной **long** должно быть не короче машинного представления переменной **int**).

При выполнении операции присваивания тип правой части приводится к типу левой, например:

```
...
int x;
double y=2.9;
x=2+y;
```

Тип правой части – **double**, а переменной **x** будет присвоено значение 4.

Кроме автоматического приведения типов в выражении можно использовать операцию явного приведения типа:

```
(<тип>)<выражение>
```

Примечание. В языке Си нет логического типа, а любое значение, отличное от нуля, считается истинным. Выражение-условие принимает значение 1, если оно истинно и 0, если ложно. Например:

```
 ...
int c,d;
c=getchar();
```

```
d=c>='0' && c<='9';
```

Если введенный символ – цифра, то переменной **d** будет присвоено значение **1**, в противном случае **0**.

Вычисление логического выражения прекращается как только становится известна истинность или ложность результата.

Упражнения

1) Какое значение примет переменная **z**?
```
   int x=1, y=2, z;
a) z=x & y | x && x==y;
b) z=x & y | (x && y);
```
2) Изменятся ли значения переменных **x,y,z** и какое значение будет присвоено переменной **k**?
```
   int x, y, z, k;
   x=y=z=1;
   k=++x && ++y || ++z;
```
3) Как будет вычисляться выражение `z=x+++y` ?
4) Какое значение примет переменная **k**?
```
   int k;
   k=~(~0<<2);
```
5) Определить значение выражения и тип результата.
   а) `-3*4L-6/5;`
   б) `(7+6)%5/2;`
6) Какие значения примут целочисленные переменные **x,y,z** после выполнения следующих операторов:
```
   x=3; y=3; z=1;
   z+=x<y?x++;y++;
```
7) Реализовать эффективное умножение и деление переменной **x** на 16, используя операции поразрядного сдвига (указать возможный тип переменной **x** и диапазон ее начальных значений для корректного выполнения операции).
8) Используя тернарную операцию, присвоить **z** наименьшее из значений **a, b** и **c**.
9) `int x;`
   Инвертировать 3 бита переменной **x**, начиная 5-го, а остальные биты оставить без изменения.
10) `int z;`
    Обнулить в **z** все разряды, кроме 5 младших, а в 3 младших разрядах установить 1.
11) Правильно ли записан оператор присваивания `k=i+++j;` и в каком порядке будут выполняться операции.

ТЕМА 2. Операторы управления

Программа на языке Си состоит из набора функций. При запуске программы на выполнение управление передается на точку входа в функцию с названием **main**[1].

Функции состоят из операторов. Выражение становится оператором, если за ним идет точка с запятой. Точка с запятой заканчивает любой оператор, кроме составного оператора **{ }**[2] Например: **x=0** – это операция присваивания, а **x=0;** – оператор присваивания. Кроме операторов присваивания в языке Си присутствуют операторы управления.

Условный оператор if-else

Условный оператор применяется для принятия решения в некоторый момент выполнения программы.

```
if(<выражение>)<оператор1>[else<оператор2>]
```

Если выражение истинно, то выполняется оператор1, в противном случае оператор2.Часть else может отсутствовать.

```
if(a>b) z=a; else z=b;
```

Если в одной из веток необходимо выполнить несколько операторов, используется составной оператор **{ }**.

Примечание. После правой фигурной скобки в приведенном примере, как и в других случаях использования составного оператора, точка с запятой не ставится[3].

Во вложенных операторах **if** действует правило: **else** всегда относится к ближайшему незакрытому оператору **if**. Для изменения этого порядка выполнения операторов обычно используют составной оператор.

Задача 1. Ввести из стандартного входного потока один символ и, если это цифра, то присвоить целой переменной **k** нулевое значение, если введенный символ окажется буквой **a**, то **k** присвоить значение один, во всех остальных случаях переменной **k** присвоить значение два.

```
...  int c,k;
     c=getchar();
     if(c>='0' && c<='9') k=0;
     else if(c=='a') k=1;
```

---

[1] ее прототип фиксирован, посредством параметров функции main программа может получить информацию от запускающей среды, см. тему 6

[2] В Си ';' не разделяет операторы, как в других языках программирования, например, в Паскале, а является неотъемлемой частью оператора.

[3] Запись if(x==y){…};else{…} приведет к ошибке компиляции (здесь ; рассматривается компилятором как пустой оператор после "сокращенного" условного, а следовательно, для else нет соответствующего if)

```
else k=2;
```

Переключатель switch

При множественном ветвлении программы иногда вместо вложенных операторов **if** удобнее использовать переключатель.

```
switch(<управляющее выражение>) {
  case <константное выражение 1>: <оп1>...<опN>
[break;]
  .  .  .
  case <константное выражение M>: <оп1>...<опN>
[break;]
  [default: <оп1> . . . <опN>]
}
```

При совпадении значений управляющего выражения и константного выражения одной из ветвей, выполняются операторы этой ветви. Оператор **break** вызывает немедленный выход из переключателя. Если оператор **break** отсутствует в выбранной ветви, то будут выполнены операторы следующей ветви и т.д. до тех пор, пока не встретится оператор **break** или не закончатся ветви переключателя.

Примечание. Ветви **case** и **default** могут быть расположены в любом порядке, при этом операторы ветви **default** выполнятся только в том случае, если не будет найдено совпадения значения управляющего выражения ни с одним константным выражением других ветвей.

Далее приводится решение задачи 1 с помощью переключателя.

```
... int c,k;
switch(c=getchar()) {
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9': k=0; break;
case 'a': k=1; break;
default: k=2;
}
```

Операторы цикла while и do-while

В операторе цикла с предусловием:

```
while (<выражение>) <оператор>
```

и операторе цикла с постусловием:

```
do <оператор> while(<выражение>);
```

тело цикла выполняется до тех пор, пока выражение истинно. Если в цикле необходимо выполнить несколько операторов, используется составной оператор **{}**.

Задача 2. Из стандартного входного потока ввести строку и записать ее в массив str. Длина строки не превышает 80 символов.

```
... int c,i;
     char str[80];
     i=0;
     while((c=getchar())!='\n') str[i++]=c;
     str[i]='\0';
```

Оператор цикла for

Оператор цикла for состоит из заголовка и тела цикла.

```
for([<выражение1>];[<выражение2>];[<выражение3>])
     <оператор>
```

Оператор тела цикла выполняется до тех пор, пока **выражение2** истинно. При отсутствии **выражение2** считается истинным. **Выражение1** обычно содержит начальные присваивания, а **выражение3** – изменение значения параметра цикла на каждой итерации. В этих выражениях часто используют операцию "," (запятая). Алгоритм выполнения цикла for выглядит следующим образом:

```
< выражение1>;
while(<выражение2>) { <оператор> <выражение3>; }
```

Например, чтобы обнулить элементы целочисленного массива, можно использовать цикл for.

```
...
int m[10], i;
for(i=0;i<10;i++) m[i]=0;
```

В вырожденном случае все выражения и оператор тела цикла могут отсутствовать. Например: **for(;;);** – это бесконечный цикл.

Включение в заголовок цикла посторонних вычислений допустимо, но считается плохим стилем программирования.

```
...
int m[10], i;
for(i=0;i<10;m[i++]=0);
```

Примечание. Параметр цикла и составляющие условия его окончания могут изменяться в теле цикла. Выходное значение параметра цикла всегда определено.

Задача 3. Определить длину строки. Нулевой байт не считать.

```
...
char s[100];
int len, i;
for(i=0;s[i];i++);
len=i;
```

Задача 4. Удалить из строки все пробелы.

```
...
char s[100];
```

```
    int i, j;
    for(i=0,j=0;s[i];i++)
      if(s[i]!=' ') s[j++]=s[i];
    s[j]='\0';
```

Задача 5. Нарисовать горизонтальную гистограмму частоты вхождения введенных символов. Ввести масштабирование по горизонтальной оси так, чтобы гистограмма была развернута на все окно.

```
#include <stdio.h>
main()
{  int c, m[256]={0}, i, j, max;
    while((c=getchar())!='\n') m[c]++;
    for(i=1,max=m[0];i<256;i++)
      if(max<m[i]) max=m[i];
    for(i=0;i<256;i++)
      if(m[i]){
         printf("%c",i);
         for(j=0;j<80*m[i]/max;j++) putchar('*');
         putchar('\n');
      }
}
```

Операторы break и continue

Оператор **`break;`** вызывает немедленный выход из тела цикла или из переключателя **`switch`**.

Оператор **`continue;`** вызывает немедленный переход к следующей итерации цикла (**`while`**, **`do-while`** или **`for`**).

Упражнения

1. #define MAXLEN 256
   char str[MAXLEN];
   int i;

Пусть в символьный массив str записана строка, длина которой меньше MAXLEN.

   a) Реверсировать строку, не используя дополнительных массивов.
   b) char str1[MAXLEN];
      Копировать строку str в str1.
   c) Преобразовать строку в целое.
   d) Целое число, записанное в переменной i, преобразовать в строку.

2. Входной поток состоит из слов. Длина слова не превышает 80 символов. Разделители между словами: пробел, запятая, конец строки.
   a) Распечатать входной поток по одному слову в строке.
   b) Распечатать размер самой длинной строки.

c) Распечатать самую длинную строку.

3. Построить вертикальную гистограмму частоты вхождения произвольных символов во вводимую из стандартного входного потока строку. Гистограмма должна занимать все окно, т.е. масштабирование должно быть произведено как по вертикали, так и по горизонтали.
4. Ввести строку из стандартного входного потока в массив типа char, удалить из строки комментарии. (/* . . . */) Дополнительные массивы не использовать.
5. Написать машинно-независимую программу, обнуляющую каждую четную двоичную единицу в числе типа int.
6. Написать машинно-независимую программу, транспонирующую двоичный код целого.

ТЕМА 3. Понятие функции. Общая структура программы. Области видимости и существования переменных

Понятие функции

Как уже говорилось, программа на языке Си представляет собой набор функций. Для того, чтобы программа была выполняемой, она должна содержать функцию с названием **main**, которой передается управление при запуске программы на выполнение.

В Си отсутствует понятие вложенных функций, все функции находятся на одном уровне видимости (глобальном), и их имена должны быть уникальны в рамках программы.

В стандарте ANSI C представлено два способа описания и определения функций, так называемые «новая» (введенная стандартом) и «старая» (используемая до принятия стандарта) нотации.

Согласно новой нотации определение любой функции имеет следующий вид:

```
[<тип результата>]<имя функции>([список параметров])
{ [декларации]
  [операторы]
}
```

Согласно старой нотации функция определяется следующим образом:

```
[<тип результата>]<имя функции>([список имен парам.])
[описание параметров]
{ [декларации]
  [операторы]
}
```

В этом случае в круглых скобках после имени функции перечисляются имена параметров, а их типы задаются перед первой открывающей фигурной скобкой.

Отметим, что запись параметров в старой нотации можно встретить лишь в очень старых кусках кода на языке Си, написанных до введения первой версии стандарта ANSI C 1989 года. Компиляторы (и стандарт) допускают такую форму записи лишь из соображений обратной совместимости со старым кодом. Ни при каких обстоятельствах программисту не следует использовать старую нотацию при создании новых программ на языке Си.

Возвращаемое значение

Функция может возвращать одно значение, тип которого указывается в заголовке, а само значение задается выражением в операторе **return**:

```
return <выражение>;
```

Оператор **return** возвращает управление вызвавшей функции. Выполнение оператора **return** в функции **main** вызовет завершение программы.

Если возвращаемое значение имеет тип **int**, то тип результата при описании функции можно не указывать (хотя это и не является хорошим стилем программирования).

В языке Си нет разделения на процедуры и функции. Функция может не возвращать значение, тогда вместо типа результата при описании функции следует писать **void**, а оператор **return** используется без возвращаемого значения:

```
void f(int a) {
     /*…*/
     return;
}
```

Попытка использовать оператор **return** с параметром в функции, не возвращающей значения (и наоборот) приведет к ошибке компиляции.

Прототипы функций

В стандарт Си было добавлено средство контроля за соответствием фактических параметров в вызове функции ее формальным параметрам. Если вызов функции расположен в теле программы до ее определения, то до использования такой функции (обычно в начале программного файла или в подключаемом заголовочном файле) задается ее описание, или **прототип**. Синтаксически прототип повторяет заголовок функции, за которым вместо ее тела следует точка с запятой.

При отсутствии прототипа функции, если в выражении встретилось имя, нигде ранее не описанное, за которым следует открывающая скобка, компилятор будет считать его именем функции, возвращающей значение типа **int**, не делая никаких предположений о ее параметрах. Это означает, что при несоответствии типов формальных и фактических параметров автоматическое приведение типов не осуществляется, что может привести к ошибочному результату. А при наличии прототипа функции все приведения типов осуществляются автоматически.

Согласованность имен параметров в прототипе и в определении функции не требуется, более того, в прототипе могут быть указаны только типы параметров без имен.

Если функция не принимает никаких параметров, в круглых скобках после ее имени следует указать ключевое слово **void**. «Пустой» список параметров в прототипе функции не говорит компилятору о том, что их нет – в последнем случае компилятор просто не

делает никаких предположений о количестве и типах параметров функции (как и в случае отсутствия прототипа).

В случае старой нотации в прототипе функции параметры не указываются, и компилятор не может обнаружить ошибок при несоответствии количества формальных и фактических параметров или их типов. Пример описания и определения функции в новой и в старой нотациях:

```
/*прототип */
int f(char);
     ...
/* определение */
int f(char c)
{
 /*…тело функции…*/
}
```

новая нотация

```
/*прототип */
int f();
     ...
/* определение */
int f(c)
char c;
{
 /*…тело функции…*/
}
```

старая нотация

В языке Си аргументы всегда передаются в функцию «по значению». То же касается и возврата значения из функции.

Задача 1. Написать функцию, суммирующую два вещественных значения в двух вариантах: сумма – возвращаемое значение функции и сумма – параметр функции.

```
 . . .
double sum1(double,double);/* прототип
                                  функции sum1*/
void sum2(double,double,double*); /* прототип
                                    функции sum2*/
main()
{ double a1=2.5, u;
  int a2=3;
  u=sum1(a1,a2);
  printf("u=%f\n",u); /* u=5.5 */
  sum2(a1,a2,&u);
  printf("u=%f\n",u); /* u=5.5 */
}
double sum1(double x,double y)
{ return x+y; }
void sum2(double x,double y,double *z)
{ *z=x+y; }
```

Список параметров переменной длины

Список формальных параметров функции может быть переменной длины – в этом случае он заканчивается “, …” и должен со-

держать хотя бы один обычный именованный параметр. Примером такой функции является функция печати **printf**, описанная в заголовочном файле **<stdio.h>**:

```
int printf(const char *format, …);
```

Для работы с параметрами, количество и тип которых заранее не известны, используется механизм макросов из файла стандартной библиотеки **<stdarg.h>**. В теле функции должна быть описана переменная типа **va_list**, которая будет содержать информацию о списке неименованных аргументов. Чтобы получить доступ к неименованным параметрам через эту переменную, необходимо прежде всего обратиться к макросу **va_start.** Макрос **va_start** получает два параметра: упомянутую переменную типа **va_list** и последний именованный параметр функции. Он инициализирует переменную типа **va_list** так, чтобы она указывала на первый неименованный параметр:

```
int f(p1, p2, ...)
{
  va_list parg;
  va_start(parg, p2);/* p2 – последний
          именованный параметр функции f */
  int i = va_arg(parg, int);
     ...
  va_end(parg);
}
```

Далее, каждое обращение к макросу **va_arg** выдает очередной параметр и передвигает указатель (переменную типа **va_list**) на следующий параметр. Макросу **va_arg** передают два аргумента: переменную типа **va_list,** предварительно проинициализированную с помощью **va_start**, и имя типа ожидаемого параметра. Таким образом, программа сама должна определить, каков тип очередного ожидаемого ею аргумента из неименованного списка. Макрос **va_end** нужен для корректного завершения работы с переменной типа **va_list**.

Поскольку функции не известны ни типы передаваемых неименованных параметров, ни их количество, то для корректной работы с такими параметрами может использоваться различная дополнительная информация. Например, как в случае функции **printf**, параметр **format** полностью определяет типы и число последующих параметров. Другой способ – передавать через фиксированный параметр количество неименованных параметров, если их типы предполагаются известными.

Задача 2. Написать функцию, возвращающую максимальный из полученных ею фактических параметров (типа **unsigned int**).

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

```
#include <stdarg.h>

unsigned int unsgn_max(unsigned int count, ...);
     /* Число неименованных параметров передается через
     параметр count, который не учитывается при поиске
     максимума. */
main()
{
  printf("max = %d\n",
           unsgn_max (3, 10, 20, 30));
}
unsigned int unsgn_max(unsigned int count, ...)
{
  va_list ap;
  unsigned int res = 0, cur;
  int i;
  va_start(ap, count);
  for(i=1; i<=count; i++)
     if(res < (cur = va_arg(ap,unsigned int)))
       res = cur;
  va_end(ap);
  return res;
}
```

Общая структура программы

Программа может размещаться как в одном, так и в нескольких файлах, содержать одну или несколько функций, одна из которых считается головной (**main**) – с нее начинается выполнение программы. Определение каждой функции должно полностью размещаться в одном файле, при этом файл может содержать несколько определений различных функций.

В соответствии со структурой программы переменные обладают следующими качествами:

- **видимость переменной (область видимости);**
- **существование переменной (область существования).**

Область видимости переменных

Область видимости переменной определяет часть исходного текста программы, из любой точки которой доступна данная переменная.

С точки зрения видимости можно выделить следующие группы переменных:

- видимые в пределах блоков (локальные)[4],

[4] блоками являются: составной оператор, определение (тело) функции, условный оператор и переключатель switch, оператор цикла (вложенным блоком для него является также тело цикла)

- видимые в пределах файла,
- видимые в пределах программы.

Для переменных, определенных в начале любого блока, областью видимости является весь этот блок. В случае вложенных блоков переменные, определенные внутри вложенного блока, “перекрывают” переменные с такими же именами, определенные в объемлющем блоке (и так для любой степени вложенности). Например:

```
main()
{ int x = 1;
  if(x>0)
  {  int x = 2;
     printf(“x = %d\n”, ++x); /*выводит: x = 3 */
  }
  printf(“x = %d\n”, x); /*выводит: x = 1 */
}
```

Переменные, определенные внутри функции, “перекрывают” формальные параметры с теми же именами[5]:

```
int f(int f)
{
  int f = 1;
     …
}
```

Переменные, определенные вне блоков (т.е., фактически, вне тела какой-либо функции), доступны с точки определения до конца файла. Если на такую переменную нужно сослаться до того, как она определена, необходимо ее описание со спецификатором **extern**, например:

```
int x;
main()
{
  extern int y;
  x = y = 10;
  printf(“x=%d, y=%d\n”,x,y); /* x=10, y=10 */
}
 …
int y;
```

В файле вне функций не может встречаться несколько определений переменных (возможно, разных типов) с одним и тем же именем:

```
int x;
main()
```

[5] Следует заметить, что в приведенном ниже примере коллизии имени функции f с именем формального параметра или с переменной f, определенной внутри функции, не будет из-за различия областей видимости.

```
{
  ...
}
float x;/* ошибка: повторное определение x */
```

Указание **static**, примененное к нелокальной переменной или функции, ограничивает область их видимости концом файла[6]. Прочие же функции и нелокальные переменные могут быть доступны из другого программного файла.

Если используемая переменная определена в другом программном файле, она также должна быть описана со спецификатором **extern**. При таком описании переменной память под нее не отводится, а только декларируется тип переменной, что позволяет компилятору осуществлять проверки типизации.

Примечание. Переменные, определенные внутри блока, как и формальные параметры функций, "перекрывают" переменные с теми же именами, видимые в пределах файла и в пределах программы.

Область существования переменных

Область существования переменной – это множество всех точек программы, при приходе управления на которые переменная существует, т.е. для нее выделена память. В отличие от области видимости, про область существования можно сказать, что это понятие «времени выполнения». С этой точки зрения можно выделить две группы переменных:

1. Статические переменные

Переменные, являющиеся статическими, существуют на всем протяжении работы программы. Память под эти переменные выделяется на этапе редактирования внешних связей и загрузки программы, тогда же происходит и инициализация статических переменных (следует отметить, что инициализатором для статической переменной может служить только константное выражение, а при отсутствии инициализатора статические переменные по умолчанию инициализируются нулем). Правила определения статических переменных различаются в зависимости от конкретного места программы, в котором это определение встретилось. Для определения статической переменной, локализованной в блоке, используется ключевое слово **static**, например:

```
int max; /*статическая переменная вне блока*/
int f(int param)
{
```

[6] Здесь возникает некая терминологическая путаница, поскольку, как будет рассказано ниже, ключевое слово **static** еще используется для определения статических (с точки зрения области существования) переменных, а в данном контексте с помощью **static** задаются границы области видимости.

```
    static int min; /* статическая переменная,
                       определенная внутри блока */
        …
}
```

Основным свойством статических переменных, определенных внутри блока, является сохранение их значений при выходе из блока. Например:

```
#include <stdio.h>
void print_a();
main()
{ int i;
  for(i=0; i<5; i++)
     print_a();
}

void print_a()
{
  static int a = 1;
  printf("a = %d\n", a++);

}
```

в результате напечатается:

```
a = 1
a = 2
a = 3
a = 4
a = 5
```

Все переменные, определенные вне функций, являются статическими.

2. Автоматические переменные

Автоматическими переменными являются все переменные, определенные внутри блока (функции) и не являющиеся статическими. Автоматические переменные существуют на протяжении работы блока, в котором они определены, включая блоки, вложенные в данный. Выделение памяти под автоматические переменные и их инициализация осуществляется каждый раз при входе в блок. Инициализация для автоматических переменных, фактически, эквивалентна присваиванию, т.е. в качестве инициализирующего выражения может выступать любое выражение, которое может стоять в правой части оператора присваивания. При отсутствии инициализатора начальное значение по умолчанию для автоматических переменных не определено.

```
#include <stdio.h>
void print_a();
main()
{ int i;
  for(i=0; i<5; i++)
     print_a();
}

void print_a()
{
  int a = 1;
  printf("a = %d\n", a++);

}
```

в результате напечатается:

```
a = 1
a = 1
a = 1
a = 1
a = 1
```

При выходе из блока память, выделенная под автоматические переменные, освобождается.

3. Регистровые переменные

В качестве одного из доступных программисту средств оптимизации язык Си предлагает возможность использования так называемых регистровых переменных. Это достигается за счет использования квалификатора **`register`** в определении переменных, что указывает компилятору, что данную переменную в целях ускорения программы имеет смысл разместить на регистрах, однако компилятор может проигнорировать это указание. Квалификатор **`register`** может применяться только к автоматическим переменным и формальным параметрам функций. Независимо от того, была ли переменная, описанная с квалификатором **`register`**, действительно размещена на регистрах или нет (что программисту неизвестно), для нее не определено понятие адреса (т.е. не определена операция &).

(*) – стековый кадр – непрерывная область памяти стека, используемая функцией для своей работы

fp (*frame pointer*) – указатель стекового кадра

Рис.1

Стековый кадр

Обычно размещение в памяти автоматических переменных, как и передача параметров и возврат значений из функций, реализуются с использованием стека. Параметрам функции отводится память в стеке, в которую копируются фактические значения. По окончании работы функции эта память освобождается.

Рассмотрим состояние стека на примере рекурсивной функции (см. Рис.1). В примере каждый раз при вызове функции `f` в стеке сохраняется адрес возврата и значения регистров, после чего там же отводится место для параметра `a` и автоматической переменной `i`, а также для автоматической переменной `r` в случае, если ее не удалось разместить на регистрах. Вся вместе эта область стека называется **стековым кадром.** Заметим, что при рекурсивном вызове функции `f` в стеке еще раз выделяется память под новый стековый кадр, и в ней размещаются новые копии автоматических переменных и пара-

метры нового вызова. При выходе из функции стековый кадр освобождается.

Упражнения

1. Написать функцию getletter(), читающую одну литеру из стандартного ввода с помощью getchar() и возвращающую введенную литеру, если это большая или малая латинская буква, и –1 в противном случае.
2. Написать функцию, которая считывает со стандартного ввода n чисел (n – параметр функции) и возвращает их среднее арифметическое.
3. Написать рекурсивную функцию вычисления факториала неотрицательного числа.
4. Написать функцию, которая для заданных натурального n и вещественного x вычисляет значение выражения:

$$s = \sin x + \underbrace{\sin \sin x + \ldots + \sin \sin \ldots \sin x}_{n \text{ раз}}$$

5. Написать функцию, вычисляющую n-ое число Фибоначчи (n>=0) по правилу:

$$F(n) = \begin{cases} 1, \text{ при } n=0 \text{ или } n=1 \\ F(n-1) + F(n-2), n>=2 \end{cases}$$

ТЕМА 4. Указатели и массивы

Указатели и операции с ними

Одной из отличительных особенностей языка Си является возможность использования в программах указателей. Указателем является переменная, значение которой есть адрес[7] некоторой области памяти. Указатель может ссылаться на переменные определенного типа и функции.

Например:

```
char c, *p;
```

где **c** – это переменная символьного типа, а **p** – это указатель на объекты символьного типа. Указатель **p** может содержать адрес любого объекта символьного типа.

Унарная операция **&** – это операция взятия адреса объекта, например: **p=&c;**. Унарная операция ***** – это операция косвенной адресации (или раскрытия ссылки). Например, оператор ***p='a';** интерпретируется следующим образом: объекту, на который ссылается указатель **p**, присвоить значение, равное коду символа **'a'**.

При увеличении значения указателя на единицу на самом деле адрес увеличивается на размер объекта. Например:

```
double x=1.5,y=7.1,*dp;
dp=&x;
dp++;
y=*dp-3;
```

Для примера будем считать, что переменные **x** и **y** размещены в памяти рядом (подавляющее большинство компиляторов разместит их рядом, но в стандарте языка Си это не зафиксировано), тогда после операции **dp++** значение указателя **dp** увеличится на **sizeof(double)** и он будет указывать уже на переменную **y**. В следующем операторе присваивания значение переменной **y** уменьшится на три.

Примечание. Унарные операторы ***** и **&** имеют более высокий приоритет, чем арифметические операторы, и одинаковый приоритет с унарными операторами **++** и **--** . Выражение ***p++** интерпретируется следующим образом: взять значение по адресу, хранящемуся в указателе **p**, а затем увеличить адрес, хранящийся в переменной **p**, на размер объекта, на который ссылается этот указатель. Операции ***** и **++** имеют одинаковый приоритет, но выполняются справа налево. Поэтому сначала будет выполнена постфиксная операция **p++**, но при этом увеличение значения адреса будет отложено до

[7] Речь идет об условной адресации в Си-машине. Эта адресация может не совпадать с реальной адресацией ОЗУ ЭВМ.

окончания вычисления всего выражения. Аналогично, в случае ***--p** адрес в **p** уменьшится прежде, чем по этому указателю будет получено число, т.к. здесь **--** – префиксная операция.

Нетипизированные указатели

В языке Си можно также определить указатель на неопределенный тип:

```
void *vp;
```

Такому указателю может быть присвоен адрес объекта любого типа или значение любого указателя. Обратное неверно.

```
int *ip, k;
void vp;
ip=&k;
vp=ip;          // Правильно
ip=vp;          // Неправильно
ip=(int*)vp;    // Правильно
```

Над указателем типа **void*** нельзя выполнять операцию косвенной адресации и действия адресной арифметики без явного приведения типа.

```
k+=*(int*)vp;   // Правильно
(int*)vp++;          // Неправильно
((int*)vp)++;   // Правильно
```

Понятие массива

В языке Си массив представляет собой совокупность элементов одного и того же типа. При определении массива указывается тип элементов и их количество в квадратных скобках. Например, описание:

```
int m[10];
```

вводит переменную **m** – массив из 10 целочисленных элементов.

Массив можно конструировать из объектов любого типа (например, базовых числовых типов, указателей, структур и объединений, а также других массивов – при этом получаются многомерные массивы).

Например,

```
int m[3][4];
```

это определение массива из трех элементов, каждый элемент которого в свою очередь является массивом, состоящим из четырех целочисленных объектов.

Доступ к элементам массива осуществляется с использованием операции индексирования **[]**, при этом индексация элементов начинается с нуля. Например, **a[3]** – это четвертый элемент массива **a**, а запись **m[i][j]** определяет элемент двумерного массива, где

i – номер строки, а **j** – номер столбца. Массивы хранятся построчно, т.е. в многомерном массиве быстрее всего изменяется самый правый индекс.

Инициализация массивов

При инициализации массивов инициализирующие значения заключаются в фигурные скобки. Если инициализирующих выражений не хватает, то оставшимся элементам массива присваиваются нулевые значения. Например:

```
int m1[5]={0,1,2,3}, m2[10]={0};
```

Последний элемент массива **m1** и все элементы массива **m2** будут обнулены. Если же размер массива не указан, то он определяется по количеству инициализирующих значений.

Массив из элементов типа **char** (строка) может быть проинициализирован двумя способами:

```
char str1[ ]={ 'a','b','c','\0'};
char str2[ ]="abc";
```

Под строки **str1** и **str2** будет отведено по 4 байта. Строка **str2** будет автоматически дополнена нулевым байтом.

Инициализация многомерных массивов осуществляется аналогично одномерным массивам. Например:

```
int m[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

Если количество инициализирующих значений меньше, чем указанная размерность, то соответствующим элементам присваиваются нулевые значения. Например:

```
int w[3][3]={{1,2,3},{4,5}};
```

Элементы: **w[1][2], w[2][0], w[2][1], w[2][2]** примут нулевые значения.

Для многомерных массивов пустой может быть только первая размерность, реальное значение которой определяется по количеству инициализирующих значений. Например, если в вышеприведенном примере опустить число строк, то тогда будет определена матрица 2x3.

Указатели и массивы

В Си понятия указателя и массива тесно связаны, и указатели очень часто используются при работе с массивами.

Имя массива есть константный указатель, содержащий адрес его нулевого элемента.

```
double m[10], *dp;
```

Чтобы присвоить указателю **dp** адрес начала массива **m**, можно написать:

```
dp=m;
```

или

dp=&m[0];,

что эквивалентно. Увеличение значения указателя на единицу приводит к следующему элементу массива.

```
dp=m;      - адрес m[0]
dp+1       - адрес m[1]
dp+i       - адрес m[i]
*(dp+i)    - эквивалентно m[i]
m++;       /* ошибка, константный указатель
           изменять нельзя */
```

Операция индексирования **E1[E2]** определена как ***(E1+E2)**, следовательно, запись **m[i]** эквивалентна ***(m+i)**, а **m[i][j]** тождественно ***(*(m+i)+j)**, и т.д.

Таким образом, указатели и адресную арифметику удобно использовать при поиске и обработке элементов массива.

При инициализации указателя на строку константой-строковым литералом, в него записывается адрес этой константной строки, элементы которой, вообще говоря, изменять нельзя.

```
char *p="abc";
*p='A';          /* Неправильно */
*str2='A';       /* Правильно */
```

Задача 1. В вещественном массиве найти максимальное значение.

```
double m[100], *p, max;
.  .  .
for( p=m+1,max=*m;p<m+100;p++)
  if(max<*p) max=*p;
```

Задача 2. Написать функцию суммирования двух целочисленных векторов. Размер векторов и результирующий массив передаются в качестве параметров.

```
  . . .
void sum_vec(int*,int*,int*,int);
main()
{ int m1[20],m2[20],m3[20],i;
  . . .
  sum_vec(m1,m2,m3,20);
  for(i=0;i<20;i++)
     printf("m3[%d]=%d\n",i,m3[i]);
}
void sum_vec(int x[],int *y,int z[],int k)
{ int i;
  for(i=0;i<k;i++)
     *z++=x[i]+y[i]; /*z[i]=x[i]+y[i];*/
}
```

Указатели можно присваивать, сравнивать, вычитать и складывать с целыми числами.

Задача 3. Написать функцию, определяющую длину Си-строки.

```
int len_str(char * str) {
     int len;
     char * p = str;
     while(*p) p++; /* Си-строка оканчивается
                       нулем */
     len=p-str;
     return len;
}
```

Задача 4. Написать функцию копирования строк.

```
void copy (char source[], char dest[]) {
     while( *dest++=*source++);
}
main() {
     char s[100], t[100];
     copy (s, t);
}
```

На примере двумерного целочисленного массива m рассмотрим понятие указателя на массив.

```
int *p1;
int (*p2)[4];
```

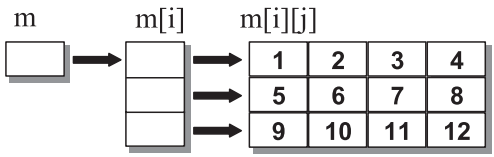
p1 – это указатель на объект целого типа, ему может быть присвоен адрес любого элемента матрицы m. Например, адрес 0-го элемента 1-ой строки можно присвоить тремя эквивалентными способами:

```
p1=&m[1][0];        p1=m[1];        p1=*(m+1);
```

p2 – это указатель на массив из четырех целочисленных элементов и ему может быть присвоен адрес любой строки матрицы, например:

```
p2=m+1;
```

Соответственно, оператор p1++; вызывает переход к следующему элементу 1-ой строки, а оператор p2++; вызывает переход к следующей строке матрицы. Тогда *p1=6, а **p2=9.



Рассмотрим подробнее работу с массивом указателей:

```
char *mas[]={"for","while","do","return",NULL};
```

Элементами массива mas являются адреса строковых констант.

`mas[i]` – адрес i-ой строки (адрес её нулевого символа).

`mas[i]+j` – адрес j-го символа i-ой строки.

Задача 5. Написать функцию, которой в качестве аргумента передается массив указателей на строки (признак конца – нулевой указатель). Распечатать последний символ каждой строки.

```
void fp(char *s[])/* void fp(char **s)
                          тождественно */
{
     int i=0;
     while(s[i]!=NULL) {
          printf("%c\n", *(s + strlen(s[i])-1));
          i++;
     }
}
```

Передача и возврат массивов из функции

Если аргументом функции является имя массива, то функции передается копия адреса начала этого массива, а сам массив не копируется (как уже говорилось, имя массива есть указатель на его нулевой элемент, и в этом случае фактически в функцию передается именно указатель на начало массива). Нельзя передать функции сам массив целиком и таким образом:

```
int arr[10];
     ...
f(*arr);
```

В этом случае функция **`f`** получит в качестве фактического значения аргумента значение нулевого элемента массива.

Поскольку при передаче параметров «по значению» функция оперирует, фактически, копиями своих аргументов, не изменяя оригиналов переданных ей значений, то внутри функции содержимое её аргумента–массива может быть изменено:

```
void f (int m[10]) {
     for (int i= 0; i < 10; i++) {
          m[i] *=2;
     }
     m++; /* корректно, но изменяется лишь локальная
копия переменной m */
}
```

Таким образом, внутри функции параметр-массив и параметр-указатель (содержащий адрес нулевого элемента массива) эквиваленты по использованию и взаимозаменяемы.

При описании аргумента функции, имеющего тип массив, допустимо опустить его размерность – в этом случае фактическим параметром может выступать массив любого размера с заданным типом элементов.

При передаче двумерного массива в качестве параметра функции возможно три варианта записи заголовка функции:

```
1) f(int  mas[3][4]);
2) f(int  mas[][4]);
3) f(int  (*p)[4]);
```

Все три варианта эквивалентны и позволяют пользоваться внутри функции как индексной записью, так и указателями.

Задача 6. Написать функцию, которая вычисляет и распечатывает сумму элементов каждого столбца вещественной матрицы 5x6.

```
void sum(double (*p)[6])
{ int i,j;
  double s,(*pp)[6];
  for(i=0;i<6;i++) {
     pp=p;
     for(j=0,s=0.0; j<5; j++) {
          s+=*(*pp+i);
          pp++;
     }
     printf("%f\n",s);
  }
}
```

Аргументы командной строки

Как правило, в операционной системе имеется возможность передавать аргументы запускаемой программе при помощи командной строки. По стандарту Си, программа может получить к ним доступ посредством аргументов функции main():

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

В первом (**argc**) передается количество аргументов командной строки, во втором (**argv**) – адрес массива указателей на строки-аргументы. По соглашению **argv[0]** всегда содержит адрес строки, в которой записано имя работающей программы. Поэтому, если значение **argc** равно 1, это означает, что в командной строке после имени программы никаких аргументов нет. Кроме того, всегда выполняется соглашение **argv[argc]=NULL**.

Стандарт POSIX.1 также определяет третий аргумент функции **main()**[8]:

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

Массив **envp[]** содержит указатели на переменные окружения, передаваемые программе. Каждая переменная – это строка вида

```
имя_переменной=значение_переменной
```

Упражнения[9]

1. Написать функцию конкатенации строк (аналог функции strcat).
2. Написать функцию сравнения строк (аналог функции strcmp).
3. Присвоить переменной единицу, если одна строка содержится в конце другой, и ноль в противном случае.
4. Поменять местами первый отрицательный элемент вещественного массива m1 с последним положительным элементом вещественного массива m2.
5. Изменить знак у всех отрицательных элементов вещественного массива X.
6. Определить, у скольких элементов целочисленного массива X равные соседи (предыдущий и последующий элементы). Записать ответ в переменную n.
7. Написать функцию проверки на равенство строк s1 и s2 при условии, что пробелы не учитываются.
8. Написать аналог функции strstr, которая возвращает указатель на первое вхождение одной строки в другую и нулевой указатель в противном случае. Используя функцию strstr, найти последнее вхождение строки “end” во введенной строке. Распечатать символы, которые следуют за этим вхождением.
9. Функции передаются 2 Си-строки:
   char s[], t[];
   Все цифры строки s записать в начало строки t, а остальные символы – в конец (в любом порядке).
10. Дано множество точек на плоскости, заданных координатами (x,y). Написать функцию, возвращающую минимальное расстояние между двумя точками.
11. Написать функцию обработки вещественной матрицы (50x20). Функция должна находить строку, содержащую наибольшее ко-

---

[8] стандарт POSIX.1 также рекомендует доступ к окружению программы через глобальную переменную **environ**:

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

имеющую аналогичное содержимое

[9] Задачи №13 и №14 взяты из первого издания книги Б.Керниган, Д.Ритчи «Язык программирования Си»

личество отрицательных элементов и распечатывать все элементы этой строки. Если таких строк несколько, то печатать все такие строки.

12. Написать функцию, осуществляющую умножение двух матриц порядка MxN и NxK.
13. Что будет выведено на печать?

```
int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int *pa[3]={a[0],a[1],a[2]};
int *p=a[0];
  main()
  { int i;
     for(i=0;i<3;i++)
     printf("%d %d %d %d %d\n",a[i][2-
i],*a[i],*(*(a+i)+i),*pa[i],p[i]);
  }
```

14. Что будет выведено на печать?

```
char *c[]={"ENTER","NEW","POINT","FIRST"};
char **cp[]={c+3,c+2,c+1,c};
char ***cpp=cp;
main()
{  printf("%s",**++cpp);
    printf("%s",*--*++cpp+3);
    printf(" %s",*cpp[-2]+3);
    printf("%s\n",cpp[-1][-1]+1);
}
```

15. Упорядочить вещественный массив x по неубыванию, используя метод сортировки выбором (найти максимальный элемент массива и переставить его с последним элементом; затем применить этот же метод ко всем элементам, кроме последнего).
16. Упорядочить вещественный массив x по неубыванию, используя метод сортировки обменом ("метод пузырька") (последовательно сравнивать пары соседних элементов: x1 с x2, x2 с x3 и т.д., и если первый элемент пары больше второго, то переставлять их – наибольший элемент окажется в конце массива; затем применить этот же метод ко всем элементам, кроме последнего).
17. Упорядочить вещественный массив x по неубыванию, используя метод сортировки вставками (пусть первые k элементов уже упорядочены по неубыванию; взять (k+1)-ый элемент и разместить его между первыми k, не нарушая порядок).
18. Дан массив, содержащий заданное количество вещественных чисел. Написать функцию, возвращающую максимальное значение этого массива.
19. Даны целочисленные массивы X и Y, содержащие по 20 элементов. Написать функцию, которая возвращает значение u.

$$U = \begin{cases} \sum_{i=1}^{20} x_i^{\,2}, при \sum_{i=1}^{15} x_i y_i > 0 \\ \sum_{i=10}^{20} y_i^{\,2}, иначе \end{cases}$$

20. Описать функцию, определяющую, сколько элементов вещественного массива X из n элементов равны числу y.
21. Написать функцию, которой передаются 2 100-элементных целых массива, определяющую, составлены ли эти 2 массива из одних и тех же чисел без учета порядка их следования, но с учетом повторяющихся чисел (функция возвращает 1 в случае положительного ответа).
22. Написать функцию, которая за 1 просмотр циклически сдвигает полученный вещественный массив на 20 позиций влево (функция для работы должна использовать вспомогательный массив).
23. Написать функции, которым в качестве параметра передается массив указателей на строки. Признак конца – нулевой указатель.
    a) Написать функцию, которая распечатывает первые три символа из каждой строки.
    b) Написать функцию, которая печатает самую длинную строку.
    c) Написать функцию, которая печатает строки, включающие в себя строку “begin”.
24. Распечатать все аргументы командной строки (без имени вызываемой программы).
25. Если среди параметров командной строки, есть строки, содержащие “end” более одного раза, то распечатать символы этих строк, находящиеся после второго вхождения “end”.
26. Написать программу, печатающую имена и значения всех переменных окружения.

ТЕМА 5. Препроцессор[10]

Стандартная схема трансляции Си-программы состоит из двух этапов: препроцессирование и собственно компиляция.

Препроцессор представляет собой предварительную стадию обработки исходного текста программы, по окончании работы которой модифицированный исходный текст поступает на вход компилятору.

Одной из примитивных задач препроцессора является удаление комментариев, а также «склеивание» строчек, последним символом которых является «\», со следующей за ними строкой (эта особенность позволяет программисту осуществлять «перенос» слишком длинных строк, например, длинных строковых литералов) и конкатенация рядом стоящих строковых литералов. Кроме того, препроцессор выполняет *макроподстановку*, *условную компиляцию*, *подключение файлов.*

Программный файл может содержать специальные директивы препроцессору. Директивам препроцессору предшествует знак #. Например:

```
#include <stdio.h>  /* подключение файла */
#define  a "max=%d\n" /* макроподстановка */
int x=15;
max(int);
main()
{ int y, u;
  scanf("%d",&y);
  u=max(y);
  printf(a,u);
}
max(int f)
{ int k;
  k=(x>f)?x:f;
  return k;
}
```

Включение файлов

На место директивы **#include** препроцессор подставляет содержимое указанного в ней файла. Порядок поиска подключаемого файла зависит от реализации Си. Если имя файла заключено в **< >**, то он ищется в стандартном каталоге подключаемых файлов (например, в некоторой UNIX-системе это может быть каталог /usr/include), как, например, в случае директивы:

[10] Мы не рассматриваем особенности функционирования конкретных трансляторов, а описываем наиболее часто используемые возможности препроцессора.

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

Если же имя файла заключено в кавычки и не является полным (абсолютным) путевым именем файла, то препроцессор рассматривает его как относительный путь от местонахождения того файла, в котором встретился **#include**. Например:

```
#include "myfile.h"
#include "/usr/stuff/myfile.h"
```

Включаемые файлы, в свою очередь, могут содержать **#include**-директивы.

Макроподстановка

Описание макроподстановки имеет вид:

```
#define имя подставляемый_текст
```

Начиная от места появления **#define** и до конца файла, везде, где встречается имя, указанное в **#define**, вместо него препроцессор подставляет заданный подставляемый текст (кроме случаев, когда это имя встречается внутри текста в кавычках, т.е. внутри строкового литерала).

Примеры:

```
#define NUMBER 10
#define DOUBLED_NUMBER NUMBER*2 /*в #define-
определении можно использовать более ранние
определения[11] */
#define PrintHello printf("Hello,\
world"); /* "\" используется для продолжения
определения на следующей строке */
```

Можно определить макроподстановку с параметрами, что позволяет изменять подставляемый текст в зависимости от фактических параметров. Например:

```
#define SQR(x) ((x)*(x))
#define Print(x) printf(#x " = %d\n", x)
/* имя формального параметра, встретившееся в "", не
заменяется на фактический параметр, но если перед
именем формального параметра стоит #, то в
макрорасширении #имя_формального_параметра будет
заменено на строку "фактический_параметр", после
чего в нашем примере строки конкатенируются */
main()
{
  int n = 4;
  Print(n);          /* выводит на печать: n = 4 */
  Print(SQR(n)); /* SQR(n) = 16 */
```

[11] при этом в текст программы вместо DOUBLED_NUMBER подставится 10*2, а не 20

```
/* следующие два примера демонстрируют необходимость
скобок в определении SQR(x) для обеспечения нужного
порядка вычислений */
  Print(SQR(n+2));   /* SQR(n+2) = 36 */
  Print(256/SQR(n)); /* 256/SQR(n) = 16 */
}
```

Оператор ## позволяет «склеивать» аргументы в макроподстановке, например: `#define bond(left, right) left##right`

Таким образом для `bond(name, 123)` будет сгенерировано `name123`

Примечание. Повторный **`#define`** для того же имени является ошибкой, если подставляемые тексты не совпадают с точностью до разделяющих пробельных символов. Препроцессору можно дать указание «забыть» определенное имя с помощью директивы

```
#undef имя
```

(если имя не было ранее определено, ошибки не произойдет).

Важно понимать, что любая макроподстановка, в отличие от вызова функции, обрабатывается не во время выполнения программы, а на этапе ее компиляции (а точнее, препроцессирования). При этом никакие вычисления не производятся, а происходит лишь модификация исходного текста программы.

Использование в качестве аргументов макроподстановки выражений с побочными эффектами (например, выражений вида **`x++`**) может привести к нежелательным последствиям, т.к. в отличие от функций, здесь не происходит однократного вычисления аргумента при вызове, и вычисление выражения-параметра произойдет столько раз, сколько оно встречается в замещающем тексте макроподстановки.

Условная компиляция

Другой пример использования препроцессора – управление выборочным включением того или иного текста в программу в зависимости от вычисляемого на этапе препроцессирования условия.

```
if-директива выражение_или_идентификатор
...
[{#elif выражение}
...
#else
... ]
#endif
```

где **`if-директива`** – это:

```
#if константное_выражение
```

(вычислить выражение и если получилось ненулевое значение, то включить весь текст до строки **`else`**-директивы или **`#endif`**)

```
#ifdef имя /* 1, если имя уже определено */
#ifndef имя /*1, если имя не было определено */
```

Например, во избежание повторного включения файла **"myfile.h"**, текст этого файла оформляют следующим образом:

```
#ifndef MYFILE
#define MYFILE

  …содержимое
  подключаемого
  файла…

#endif
```

или

```
#if !defined(MYFILE)
#define MYFILE

  …содержимое
  подключаемого
  файла…

#endif
```

“myfile.h” “myfile.h”

Общий порядок действий препроцессора таков:

- если в конце строчки исходного текста самым последним символом является «\», он удаляется, а строка «склеивается» со следующей строкой исходного текста;
- удаляются комментарии (каждый из них заменяется на одиночный пробел);
- выполняются директивы препроцессору;
- осуществляются макроподстановки;
- если в ходе выполнения директивы **#include** происходит включение файла, для его текста рекурсивно выполняются все ранее описанные шаги;
- конкатенируются рядом стоящие строковые литералы.

ТЕМА 6. Указатели на функции. Интерпретация сложных деклараций. Оператор typedef

Указатели на функции

В языке Си можно определять указатели на функции, которые ничем не отличаются от обычных указателей. Указатель на функцию можно присваивать, размещать в массиве, передавать в функцию в качестве параметра. Например:

```
double (*fp)(double);
```

fp – это указатель на функцию с одним параметром типа **double**, возвращающую значение типа **double**. Теперь указателю **fp** может быть присвоен адрес любой функции, имеющей такой прототип, после чего его можно будет использовать наравне с именем функции.

```
double x=1.57,y;
fp=sin;
y=fp(x); /* y=sin(x) - эквивалентно */
```

Задача 1. Написать программу, вычисляющую y=f(x), где имя функции (sin, cos, exp или sqrt) и ее параметр задаются в качестве аргументов командной строки.

```
#include<stdio.h>
#include<string.h>
#include<math.h>
typedef double (*tf)(double);
main(int argc,char * argv[])
{ char *str[4]={"sin","cos","exp","sqrt"};
  tf m[4]={sin,cos,exp,sqrt};
  double x;
  int i;
  for(i=0;i<4;i++)
     if(!strcmp(argv[1],str[i])) break;
  if(i==4) {
     printf("Имя функции задано неверно\n");
     return(-1);
  }
  x=atof(argv[2]);
  printf("%s(%f)=%f\n",str[i], x, m[i](x));
}
```

При передаче функции в качестве параметра ее имя трактуется как адрес этой функции, поэтому оператор **&** перед ним не нужен (также как **&** был не нужен перед именем массива).

Например:

```
void sort(void *vector[],
          int (*comp)(void*, void*));
```

```
/*функция сортировки получает функцию
  сравнения в качестве параметра */
int complen(char *, char *);
int lexcomp(char *, char *);
main()
{ int lexorder;
  char * strings[];
   ... /* заполнение strings */
  sort((void **)strings,
(int(*)(void*,void*))(lexorder ? lexcomp : complen));
/* явные преобразования нужны для проверки компилятором
согласованности типов */
}
```

Интерпретация сложных деклараций

В декларациях обычно используется имя (идентификатор) и один из модификаторов **`*`**, **`[ ]`** и **`( )`**, причем разрешается использовать более одного модификатора в одной декларации[12]. Для раскрытия этих деклараций применяются следующие правила:

1. Чем ближе модификатор стоит к идентификатору, тем выше его приоритет.
2. Приоритет **`( )`** и **`[ ]`** выше, чем приоритет **`*`**.
3. Приоритет повышается заключением в скобки **`()`**.

Примеры:

1) Массивы и указатели

`int matrix[10][10];`	matrix – массив массивов типа int
`char **argv;`	argv – указатель на указатель на char
`int (*ip)[10];`	ip – указатель на массив из 10 элементов типа int
`int *ip[10];`	ip – 10-элементный массив указателей на int
`int *ipp[3][4];`	ipp – 3-элементный массив указателей на 4-элементный массив типа int
`int (*ipp)[3][4];`	ipp – указатель на 3-элементный массив, каждый элемент которого – 4-элементный массив типа int

2) Функции и указатели

`int *f();`	f – функция, возвращающая указатель на int
`int (*pf)();`	pf – указатель на функцию, возвра-

[12] В этой теме рассматривается не строгий синтаксис деклараций, а способ разобраться в том, как понимать декларируемое имя.

	щающую int
`char (*(*x())[])();`	x – функция, возвращающая указатель на массив указателей на функцию, возвращающую char
`char (*(*x[3])())[5];`	x – массив из 3 указателей на функцию, возвращающую указатель на массив из 5 элементов типа char

Оператор typedef

Для упрощения прочтения сложных деклараций, а также для именования, типам данных можно задавать новые имена с помощью оператора **`typedef`**. Например:

```
typedef double (*PFD)();
```

определяет тип **`PFD`** как “указатель на функцию, возвращающую **`double`**”.

Оператор **`typedef`** не создает новый тип, а декларирует новое имя (синоним) уже существующего типа.

После ключевого слова **`typedef`** следует конструкция, синтаксически аналогичная блоку описания переменных, с той лишь разницей, что вводимое ею новое имя или имена являются не именами переменных, а новыми именами типов.

Например:

```
typedef int number, *num_pointer;
```

Здесь вводится два новых имени типов: **`number`** (который представляет собой синоним типа **`int`**) и **`num_pointer`** (который представляет собой синоним указателя на **`int`**).

Упражнения

1. Напечатать таблицу значений функции на отрезке [a,b] с шагом h. Имя функции(sin, cos, tan, log, fabs, sqrt или exp), a,b и h ввести как параметры командной строки. Вызов функций реализовать через массив указателей на функции.
2. Написать функцию вычисления интеграла f(x) на отрезке [a,b] методом прямоугольников. f(x) – любая функция, интегрируемая на заданном отрезке. Функцию f(x) передавать через параметр-указатель на функцию.
3. Получить польскую инверсную запись (ПОЛИЗ) выражения, содержащего цифры 0,1, . . . 9, скобки и арифметические операции: *, /, +, - .
4. Вычислить значение выражения по его ПОЛИЗ’у.

5. Ввести в качестве аргумента командной строки произвольное скобочное выражение, содержащее числа (целые и вещественные), знаки арифметических операций (+, -, *, /) и стандартные функции из математической библиотеки. Выражение может содержать параметр X, тогда и только тогда должен быть предусмотрен ввод конкретного значения. Вычислить это выражение и результат вывести на печать. В качестве усложнения задачи предлагается ввести контроль правильности введенного выражения или не ограничивать число параметров.

ТЕМА 7. Структуры и объединения. Программирование динамических структур данных

Структуры и операции над ними

Структура – это тип данных, позволяющий сгруппировать несколько переменных (возможно различного типа) под одним именем. В общем случае декларация структуры имеет следующий вид:

```
struct[<тег структуры>]{<список деклараций полей>};
```

Например, для задания комплексных переменных удобно использовать структуру:

```
struct point {
     int x;
     int y;
} a,b;
```

Это описание вводит новый тип данных – **`struct point`**. В данном примере переменные **`a`** и **`b`** определены сразу за декларацией структуры. Если структура не имеет тэга (т.е. является неименованной), то это – единственный способ объявить переменные данного структурного типа.

Примечание. Язык Си поддерживает именную, а не структурную, типизацию, что означает, что два неименованных структурных типа, пусть и содержащие совершенно идентичные списки деклараций полей, будут считаться различными и не будут совместимы по присваиванию.

В нашем примере структура имеет тег **`point`**, что позволяет использовать его в качестве имени типа, и переменные можно определить и отдельно от описания структурного типа:

```
struct point c,d,*p;
```

Использование ключевого слова **`struct`** в наименовании структурного типа по стандарту Си является обязательным. Однако, используя оператор **`typedef`** при декларации структуры, можно задать новое, более короткое, имя типа:

```
typedef struct point { int x; int y; } sp;
```

Теперь тип sp – это синоним типа struct point, и следующие определения переменной z эквивалентны:

```
struct point z;
sp z;
```

Поля структуры могут быть проинициализированы списком константных значений:

```
struct point k={3,5};
```

Доступ к полям структуры осуществляется с помощью операции «точка»: **`c.x`**, **`c.y`**. Если же определен указатель на структуру, то для доступа к полям структуры, на которую он ссылается, используется операция **`->`**. Например:

```
p=&c;
p->x=2;  /* (*p).x  - эквивалентно */
```

Структуры могут быть вложенными:

```
struct stud {
     char fio[15];/* фамилия студента*/
     struct data { int year;
                     int mon;
                     int day;
     } d;           /* дата рождения */
     int m[3]; /* оценки в сессию */
     };
```

При такой вложенной декларации структур, типы **struct stud** и **struct data** имеют одинаковую область действия.

В Си разрешается присваивать и копировать структуры, что позволяет передавать их в функцию в качестве аргумента и передавать из функции в качестве возвращаемого значения (в отличие от массивов, структуры при этом копируются целиком), но структуры нельзя сравнивать. Например:

```
struct stud s={"Ivanov",{1980,6,30},{5,3,4}};
struct data ss;
ss=s.d; /*ss.year=1980; ss.mon=6; ss.day=30;*/
if(ss==s.d) {…} /* ошибка */
```

Задача 1. Написать функцию, параметрами которой являются массив анкет студентов (**struct stud**) и их количество. Функция печатает фамилии отличников и даты рождения.

```
void f(struct stud g[],int n)
{ int i;
  for(i=0;i<n;i++) {
  if(g[i].m[0]==5 && g[i].m[1]==5 && g[i].m[2]==5)
     printf("%s %d.%d.%d\n",
          g[i].fio,g[i].d.day,g[i].d.mon,g[i].d.year)
  }
}
```

Структуры со ссылками на себя

Структуры могут содержать поля-указатели на такие же структуры. Это свойство структур используется при программировании динамических структур данных (стек, очередь, список, дерево), которые будут рассмотрены ниже. Структуры также могут содержать поля-указатели на еще не определенные структуры, что позволяет описывать структуры, ссылающиеся друг на друга, например:

```
struct s1 {
     ...
     struct s2 *ps2;
};
```

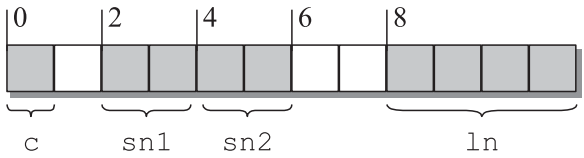
```
struct s2 {
     ...
     struct s1 *ps1;
};
```

Приведенные выше определения корректны, т.к. выделяя память под первую структуру, компилятор может определить размеры всех ее полей, ибо размер области памяти, резервируемой под любой указатель, всегда известен для конкретной машинной реализации Си.

Рассмотрим подробнее размещение структуры в памяти генератором кода. Здесь возникают две проблемы: собственно размещение объекта типа «структура» в памяти и размещение элементов структуры в области памяти, выделенной под этот объект. Адрес, с которого размещается сама структура, называется *базовым адресом*, а адрес элемента структуры относительно ее базового адреса называется *смещением.* Заметим, что каждый тип данных транслируется в определенный машинный тип (например, “байт”, “слово”, “двойное слово”), которому соответствует некоторое правило выравнивания. Рассмотрим архитектуру с байтовой адресацией и правилами размещения, согласно которым слово располагается, начиная с байта с адресом, кратным 2, а двойное слово – начиная с байта с адресом, кратным 4. В различных реализациях языка применяются различные подходы к определению того, каким может быть базовый адрес структуры. Это может быть выравнивание по типу первого элемента структуры или по максимально длинному машинному типу и т.п. Смещения элементов структуры зависят от их типов, представления их типов на конкретной архитектуре и правил выравнивания, поэтому при выделении памяти под структуры компилятором могут оставаться «пропуски».

Таким образом, важно понимать, что размер структуры не всегда равен сумме размеров ее полей. Для определения размера памяти, занимаемого переменной структурного (как и любого другого) типа, необходимо использовать оператор **`sizeof()`.**

```
struct {
  char   c;
  short  sn1;
  short  sn2;
  long   ln;
}
```

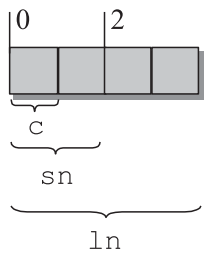


Считаем, что под переменную типа char выделяется 1 байт, под переменную типа short – слово (2 байта), а под переменную типа long – двойное слово(4 байта).

Объединения

Объединение (**union**) – это тип данных, позволяющий хранить разнородные данные (поля) в одной и той же области памяти. Синтаксис описания объединений аналогичен синтаксису описания структур, за исключением собственно ключевого слова – **union**. Фактически, объединение – это структура, все поля которой имеют нулевое смещение относительно ее базового адреса, а размер выделенной области памяти позволяет разместиться в ней самому большому полю. При выравнивании учитывается требование для максимально длинного машинного типа, используемого для представления полей объединения, т.е. если одно поле можно разместить, начиная с адреса двойного слова, а второе – с адреса байта, то компилятор следует первому требованию.

```
union {
  char   c;
  short  sn;
  long   ln;
}
```



Объединение можно инициализировать напрямую значением, соответствующим типу его первого поля:

```
union cu{
     char c;
     int I;
     double d;
} a = 'z';
```

Работа с динамической памятью

Стандартная библиотека Си содержит функции для работы с динамической памятью.

Для выделения динамической памяти служит функция:

```
#include <stdlib.h>
void * malloc(size_t size);
```

Она выделяет блок памяти указанного (в байтах) размера и возвращает указатель на него. Тип **`size_t`** представляет собой один из базовых целочисленных типов, размера которого достаточно для представления любого допустимого в данном контексте значения.

Поскольку функция возвращает нетипизированный указатель, перед использованием его, как правило, необходимо привести к требуемому типу.

В случае неудачи функция возвращает нулевой указатель.

Также для выделения динамической памяти используется функция:

```
void * calloc(size_t number, size_t size);
```

Она выделяет блок памяти для массива размером **`number`** элементов, каждый из которых занимает **`size`** байт, и инициализирует весь блок нулями. В случае неудачи функция так же возвращает нулевой указатель.

Пример использования данной функции:

```
int *ip = (int *)calloc(n, sizeof(int));
```

Для освобождения ранее выделенной динамической памяти служит функция:

```
#include <stdlib.h>
void free (void *p);
```

Ей должен быть передан указатель на начало блока памяти, ранее выделенного с помощью **`malloc()`** или **`calloc()`**. После обращения к **`free()`** этот указатель становится недействительным.

Изменить размер ранее выделенного блока памяти (как в большую, так и в меньшую сторону) можно при помощи функции:

```
#include <stdlib.h>
void * realloc(void *p, size_t newsize);
```

Эта функция при необходимости может выделить новый непрерывный блок динамической памяти требуемого размера, при этом она корректно скопирует туда содержимое старого блока и освободит старый блок памяти. Она возвращает новый указатель на блок измененного размера или, в случае неудачи, **`NULL`** (в последнем случае старый блок остается нетронутым). После обращения к

realloc() старый указатель становится недействительным. Важно понимать, что обращение к **realloc()** может быть сопряжено с большими накладными расходами из-за перемещения данных.

Пример работы с функциями выделения / освобождения памяти: выделение памяти под *массив массивов* и ее освобождение:

```
#include <stdio.h>
int main()
{
    int **arr;
    int n = 5;
    arr = (int **)malloc(sizeof(*arr) * n);
    for (int i = 0; i < n; i++){
        arr[i]=(int*)malloc(sizeof(**arr) * n);
    }
    for (int i = 0; i < n; i++){
        free(arr[i]);
    }
    free(arr);
}
```

Си не содержит средств автоматической «сборки мусора», поэтому вся забота об отсутствии «утечек памяти», когда выделенная динамическая память должным образом не освобождается программой, лежит на совести программиста.

Задача 2. Написать фрагмент программы, размещающий в динамической памяти вводимые из стандартного входного потока вещественные числа. Количество вводимых чисел вводится первым.

```
... int k,i;
    double *p;
    . . .
    scanf("%d",&k);
    p=(double*)malloc(k*sizeof(double));
    for(i=0;i<k;i++) scanf("%lf",p+i);
```

Задача 3. Ввести строку из стандартного входного потока длиной не более ста символов и разместить ее в динамической памяти.

```
... char str[100],*p;
    ...
    if(gets(str)!=NULL) {
        p=(char*)malloc(strlen(str)+1);
        strcpy(p,str);
    }
```

Как уже говорилось, возможность описывать структуры со ссылками на себя дает программисту удобное средство для реализа-

ции динамических структур данных, таких как списки, очереди, деревья и т.п.

Задача 4. Ввести строку символов из стандартного входного потока и распечатать ее в обратном порядке, построив при этом в динамической памяти стек.

```
... int q;
     struct st { int c;
                 struct st *s;
     } *p,*n;

     p=n=NULL;
     while((q=getchar())!='\n') {
          /* построение стека */
          n=(struct st*)malloc(sizeof(struct st));
          n->c=q; n->s=p;
          p=n;
     }
     while(n!=NULL) { /* печать строки */
          printf("%c",n->c);
          n=n->s;
     }
```

Упражнения

1. Дана строка символов. Написать функцию создания бинарного дерева (рекурсивный и нерекурсивный варианты). Узлы дерева имеют следующую структуру:

```
struct tree { char c;
              int n;
              struct tree* left;
              struct tree* right;
};
```

где c – символ входной строки, а n – число вхождений данного символа во входную строку. При построении дерева в качестве ключа использовать код символа.

2. Написать рекурсивную функцию подсчета количества узлов дерева.
3. Написать рекурсивную функцию определения высоты дерева.
4. Написать рекурсивную функцию печати узлов дерева в убывающем порядке.
5. Написать функцию печати узлов дерева по слоям (рекурсивный и нерекурсивный варианты).
6. Ввести набор слов. Разделители между словами: пробел, запятая, точка с запятой, конец строки. Признак конца текста – точка. Длина каждого слова не должна превышать 20 символов. Сформировать двоичное дерево, каждый узел которого содержал бы указатель на слово и число вхождений этого слова во входной по-

ток. Ключом при построении дерева должно являться само слово. По окончании формирования дерева распечатать слова в убывающем порядке, в возрастающем порядке и по уровням.

7. Ввести набор слов. Слов не более 20-ти. Разделители между словами: пробел, запятая, конец строки. Длина слова не превышает 20 символов. Разместить слова в динамической памяти (лишнего места не занимать) и сформировать массив указателей на слова. Использовать в качестве признака конца массива нулевой указатель NULL. Упорядочить слова в неубывающем порядке. Распечатать слова, каждое на отдельной строке.
8. Ввести набор строк. Количество строк не более 15-ти. Длина каждой строки не должна превышать 80 символов. Под каждую введенную строку зарезервировать в динамической памяти место под реальную ее длину, скопировать туда строку и записать адрес в массив указателей. Признак конца в массиве указателей – NULL (нулевой указатель). Написать функцию, которой передается в качестве параметра массив указателей, а выходное значение – адрес самой большой (лексикографически) строки. Используя этот адрес, распечатать самую большую строку.

ТЕМА 8. Файлы. Библиотечные функции для работы с файлами

Открытие и закрытие файла

Стандартная библиотека Си предоставляет программисту широкие возможности по работе с произвольными файлами. Любой файл предварительно должен быть открыт. Для этого используется функция **fopen**:

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

Функция получает в качестве аргументов имя файла (**name**) и режим доступа (**mode**), а возвращает файловый указатель, используемый в дальнейшем для работы с файлом. Функция возвращает нулевой указатель, если файл не может быть открыт по каким-либо причинам.

Файловый указатель ссылается на структуру типа **FILE**, содержащую следующую информацию о файле:

```
typedef struct {
  int cnt;     /* количество оставшихся литер */
  char *ptr;   /* позиция следующей литеры */
  char *base;  /* адрес буфера */
  int flag;    /* режим доступа */
  int fd;      /* дескриптор файла */
} FILE;
```

При запуске Си-программы операционная система всегда открывает три стандартных потока: входной (ему соответствует константа **stdin**, имеющая тип **FILE***), выходной (**stdout**) и файл ошибок (**stderr**). Обычно **stdin** соотнесен с клавиатурой, **stdout** и **stderr** с экраном.

При открытии файла режим доступа может принимать следующие значения:

“r”	- файл открывается только для чтения;
“w”	- файл создается только для записи, при этом, если он уже существовал, его содержимое теряется;
“a”	- файл создается или открывается для записи в конец файла;
“r+”	- файл открывается для чтения и для записи;
“w+”	- файл создается для чтения и для записи, при этом, если он уже существовал, его содержимое теряется;
“a+”	- файл создается или открывается для чтения и для записи в конец файла

После окончания работы файл должен быть закрыт с помощью функции:

```
#include <stdio.h>
int fclose(FILE* f);
```

При этом освобождаются все буфера. Если программа завершается нормально, все открытые файлы автоматически закрываются системой. Важно понимать, что количество одновременно открытых файлов в системе ограничено, поэтому программа, интенсивно открывающая файлы, должна позаботиться об их своевременном закрытии.

Функция **fclose** возвращает 0 в случае успеха и **EOF** в случае ошибки. **EOF** – это константа, обычно равная -1 и определенная в библиотеке **<limits.h>**.

Для файлов, открытых в режиме с возможностью записи, определена также специальная функция сброса буферов вывода:

```
#include <stdio.h>
int fflush(FILE *f);
```

При обращении к ней происходит дозапись на диск всех оставшихся в буфере вывода, но еще не записанных на диск данных. Для файлов, используемых в режиме и чтения, и записи, обращение к этой функции необходимо после каждой операцией записи, за которой следует операция чтения.

Чтение и запись символьных данных

Набор функций, используемых для ввода/вывода символьных данных, в большой степени аналогичен функциям для работы со стандартными потоками ввода/вывода. Например, форматированный ввод осуществляется функцией:

```
#include <stdio.h>
int fscanf(FILE *f,const char *format, . . .);
```

которая читает текстовые данные из файла **f**, преобразовывает их в соответствии со спецификаторами, содержащимися в форматной строке, и присваивает по порядку аргументам, каждый из которых должен быть представлен указателем. Аналогично функции **scanf**, **fscanf** возвращает количество считанных и преобразованных элементов. При обнаружении ошибки до присвоения значения первого поля, данные функции возвращают **EOF**.

Обратные действия производит функция форматированного вывода:

```
#include <stdio.h>
int fprintf(FILE *f,const char *format, . . .),
```

которая преобразует аргументы в текстовый вид в соответствии с форматной строкой (**format**) и пишет в выходной файл **f**.

Для чтения/записи отдельных символов также имеются функции, работающие с файлами:

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

```
int fgetc(FILE *f);
int fputc(int c, FILE *f);
```

которые аналогичны по семантике функциям **getchar()** и **putchar()**, работающим со стандартными потоками ввода/вывода:

```
#include <stdio.h>
int getchar();
int putchar(int c);
```

Обе функции **fgetc()** и **getchar()** возвращают считанный символ, а если при чтении был достигнут конец файла, то возвращается **EOF**. Его значение таково, что оно заведомо не совпадает ни с одним кодом символа в какой бы то ни было таблице кодировки. Поскольку представление -1 в типе данных **unsigned char** совпадает с максимально возможным для этого типа положительным значением, возвращаемое значение этих функций имеет тип **int**, а не **char**.

Особо внимательно следует рассмотреть библиотечные функции для чтения и записи текстовых строк. В семантике таких функций, работающих с файлами, есть существенные отличия от их аналогов, работающих со стандартными потоками:

работа с stdin/stdout	*работа с файлами*
`char * gets(char * s);` • считывает строку со стандартного ввода до символа ‘\n’ или до конца файла и записывает ее в **s**; • символ ‘\n’ при этом заменяется на ‘\0’; • возвращает строку **s** или NULL (в случае ошибки или исчерпания файла)	`char * fgets(char *s, int n, FILE *f);` • считывает не более **n** символов из файла, прекращая чтение, если достигнут символ ‘\n’ или конец файла, и записывает считанное в **s**; • символ ‘\n’, если он встретился, тоже помещается в строку **s**, которая в любом случае дополняется нулевым байтом; • возвращает строку **s** или NULL (в случае ошибки или исчерпания файла)
`int puts(char * s);` • помещает строку **s** в стандартный поток вывода, добавляя в конце символ ‘\n’; • возвращает неотрицательное целое (код последнего успешно запи-	`int fputs(char * s, FILE *f);` • помещает строку **s** в файл **f**, при этом символ ‘\n’ не добавляется (сама строка тоже может не содержать его); • возвращает неотрицательное целое (код последнего успешно записанного символа) в случае успеха ли-

санного символа) в случае успеха либо EOF в случае ошибки.	бо EOF в случае ошибки.

Таким образом, использование функции **gets()** может повлечь неприятные последствия, если буфер, выделенный под чтение строки, окажется недостаточного размера. Функция **fgets()** лишена этого недостатка, т.к. учитывает максимальный размер переданного ей буфера. Но при этом она может прочитать строку из файла не целиком, если последняя не поместилась в буфер. Распознать эту ситуацию программист может по отсутствию в конце считанной строки символа '\n'.

Примечание. Внутреннее представление строк, принятое в Си, предполагает, что массив символов, образующих строку, оканчивается нулевым байтом – символом '\0'. В то же время, строка в текстовом файле заканчивается символом перевода строки –'\n' и не содержит нулевого байта.

Например:

```
FILE *fp;
if((fp=fopen("a.a","w"))!=NULL)
   fputs("str",fp);
```

В данном фрагменте создается файл "a.a" и в него записывается строка "str". Следует заметить, что функция **fputs** пишет в файл символы строки до нулевого байта. Строка может и не содержать символа '\n', и тогда следующий **fputs** "приклеит" записываемую строку к предыдущей. Например, после fputs("string\n",fp) в файле будет записана строка "string" с '\n' на конце.

Задача 1. Переписать начальные строки (не более 10-ти) из текстового файла "aaa" в конец файла "bbb". Длина строки не превышает 80 символов (включая '\n').

```
main()
{ FILE *f1,*f2;
  char s[81];
  int i=0;
  if((f1=fopen("aaa","r"))==NULL) exit(1);
  if((f2=fopen("bbb","a"))==NULL) exit(1);
  while(fgets(s,81,f1)!=NULL && i<10) {
    i++;
    fputs(s,f2);
  }
  fclose(f1); fclose(f2);
}
```

Следует помнить, что функция **fgets()** читает не более n-1 литер в массив s, прекращая чтение, если встретился '\n', который включается в массив. В любом случае массив дополняется нулевым байтом. Если в решении задачи 1 заменить второй параметр при вызове функции **fgets()** на 80, то строки будут переписываться без искажения, но их количество может оказаться меньше десяти, если в исходном файле окажутся строки максимальной длины.

Примечание. Использование функции

```
int feof(FILE *f);
```

имеет некоторые особенности. Если при чтении из файлового потока f достигнут конец файла, то возвращается ненулевое значение, в противном случае возвращается ноль. При этом, если не предпринималась попытка прочитать информацию за концом файла, то функция **feof** не будет сигнализировать о том, что достигнут конец файла.

Например, чтобы скопировать содержимое одного файла в другой, можно было бы произвести следующие действия:

```
FILE f1,f2;
char str[80];
     .  .  .
while(!feof(f1)) {
    fgets(str,80,f1);
    fputs(str,f2);
}
```

Но тогда последняя строка исходного файла может быть переписана дважды.

Чтение и запись бинарных данных

В стандартной библиотеке предусмотрены также функции так называемого прямого ввода/вывода, предназначенные для чтения/записи несимвольных данных.

Функция:

```
size_t fread(void *p,size_t size,size_t nobj,
     FILE *f);
```

читает из входного потока в массив **p nobj** объектов размером **size** безо всякого преобразования. Она возвращает количество прочитанных объектов или ноль, если исчерпан файл.

Обратная ей функция:

```
size_t fwrite(void *p,size_t size,size_t nobj,
     FILE *f)
```

пишет в выходной поток из массива **p nobj** объектов размером **size** в непреобразованном виде. Возвращает количество записанных объектов, которое в случае ошибки меньше **nobj**.

Задача 2. В непустом файле с именем “sss” в непреобразованном(бинарном) виде записаны вещественные числа (**double**). Поменять местами первое максимальное число с последним минимальным.

```
main()
{  FILE *f;
   double max,min,a;
   long max_p,min_p;
   if((f=fopen("sss","r+"))==NULL) exit(1);
   fread(&a,sizeof(double),1,f);
   max=min=a;
   max_p=min_p=0;
   while (fread(&a,sizeof(double),1,f)) {
     if(a>max){
       max=a;
       max_p=ftell(f)-sizeof(double);
     }
     if(a<=min){
       min=a;
       min_p= ftell(f)-sizeof(double);
     }
   }
   fseek(f,min_p,SEEK_SET);
   fwrite(&max,sizeof(double),1,f);
   fseek(f,max_p,SEEK_SET);
   fwrite(&min,sizeof(double),1,f);
   fclose(f);
}
```

Позиционирование внутри файла

Для того, чтобы узнать текущую позицию в файле, используется функция:

```
#include <stdio.h>
long ftell (FILE *f);
```

Она возвращает текущее смещение в файле относительно его начала (в байтах).

Изменить его можно при помощи функции:

```
#include <stdio.h>
int fseek (FILE *f, long offset, int from);
```

Первый ее параметр задает файл, второй – требуемое значение смещения внутри файла (оно может быть как положительным, так и отрицательным), а третий параметр представляет собой одну из трех констант, указывающих, откуда отсчитывается это новое смещение:

SEEK_CUR – значение **offset** прибавляется (с учетом знака) к текущему смещению файла;

SEEK_SET – смещение **offset** отсчитывается относительно начала файла;

SEEK_END – смещение **offset** отсчитывается относительно конца файла.

Для текстового файла допустимым значением параметра **from** является только **SEEK_SET**, а параметр **offset** должен быть либо нулем, либо значением, полученным ранее с помощью **ftell()**.

В случае, если новое смещение выходит за начало файла, функция вернет -1 (при этом текущая позиция в файле не изменится), а если новая позиция в файле должна оказаться за концом файла, то поведение функции зависит от режима открытия файла: для файлов, открытых только по чтению, такая ситуация ошибочна, а для файлов, открытых с возможностью записи – нет.

В случае успеха **fseek** возвращает нулевое значение.

При работе с однородными бинарными файлами часто возникает задача определения количества объектов, записанных в файл. Например, пусть в бинарном файле записаны числа типа **long double**.

Для определения количества этих чисел достаточно поставить указатель на конец файла:

```
FILE *f;
long pos,n;
 . . .
fseek(f,0L,SEEK_END);
```

а затем с помощью функции **ftell** определить номер текущей позиции (в байтах, нумерация начинается с нуля):

```
pos=ftell(f);
```

и разделить его на размер объекта:

```
n=pos/sizeof(long double);
```

Упражнения

1. Написать функцию сортировки файла, содержащего числа типа long, записанные в файл с помощью функции fwrite, не используя дополнительных файлов. Используя эту функцию отсортировать два файла и слить их в один упорядоченный файл. Результат записать в первый файл.
2. Инвертировать файл, содержащий числа типа int, не используя дополнительных файлов.

   а) Написать функцию для записи в файл анкет студентов. Каждая анкета есть структура:

   struct anketa { char fio[20];
   struct { int day, month, year; } data;
   short m[3];
   };

в которой поле fio – фамилия студента, поле data – дата рождения, а поле m – оценки, полученные студентом в последнюю сессию.

b) Написать функцию, которая из файла анкет студентов удаляет всех “двоечников”.

3. Оставить в файле только строки, содержащие некоторую подстроку. Имя файла и подстроку задавать в виде параметров командной строки.
   а) Длина строк не превышает 255 символов.
   б) Длина строк не ограничена.
4. Удалить из файла все строки длиной более пятидесяти символов. Длина строк не ограничена. Имя файла задается в командной строке.
5. Преобразовать файл, усекая строки до 50-ти символов.
6. Упорядочить текстовый файл (лексикографически сравнить строки, длина строки не более 80-ти символов), используя один из методов внешней сортировки.
7. Написать функцию-упаковщик файлов. Коды символов лежат в диапазоне от 0 до 127. Последовательность из n одинаковых символов заменяется на байт со значением –n и байт, содержащий код символа. Рассмотреть два варианта: a) n<=127; b) n – любое.
8. Написать функцию-распаковщик файлов, см. задачу 7.
9. Заменить в файле все “begin” на ‘{‘ ,а “end” на ‘}’.

ТЕМА 9. Программирование в среде ОС UNIX. Организация работы с процессами

Понятие процесса. PID. Контекст процесса

Следует отметить, что конкретной ОС с названием UNIX, как таковой, не существует. Говоря об «операционной системе UNIX», обычно подразумевают целое семейство различных операционных систем, в основу которых положены сходные общие концепции, структуры и алгоритмы. В UNIX процесс является единицей управления и потребления ресурсов, а с точки зрения реализации ОС процесс представляет собой объект, зарегистрированный в таблице процессов ядра UNIX.

Каждому процессу в UNIX сопоставлено некое уникальное целое число, называемое идентификатором процесса – PID. Некоторые значения идентификаторов являются зарезервированными и назначаются специальным процессам ОС, например, во многих ОС семейства UNIX процесс с PID=0 ассоциируется с работой ядра ОС, а процесс с PID=1 – это процесс init.

Процесс характеризуется контекстом, который включает в себя тело процесса, содержимое регистров и аппаратных таблиц ЦП (аппаратный контекст) и атрибуты процесса (PID его родителя, идентификаторы владельца и группы, приоритет, таблицу открытых файлов, диспозицию сигналов и др.).

Порождение процессов

Для создания всех новых процессов в UNIX в операционной системе UNIX, за исключением процессов с PID=0 и PID=1, используется системный вызов **`pid_t fork (void).`**

Порожденный процесс получает свой уникальный PID. Для нового процесса создается контекст, содержимое которого по большей части копируется из контекста родительского процесса. В частности, тело порожденного процесса содержит копии сегмента кода и сегмента данных его родителя. Кроме того, в порожденном процессе наследуются таблица открытых файлов и разделяемые ресурсы процесса-отца, диспозиция сигналов, переменные окружения и др. атрибуты.

Не наследуются порожденным процессом сигналы, ждущие доставки в родительский процесс (в том числе установки взведенного «будильника», заданные системным вызовом **`alarm()`**), блокировки файлов, установленные родительским процессом.

По завершении системного вызова **`fork()`** каждый из процессов – родительский и порожденный, получив управление, продолжит выполнение с точки возврата из системного вызова **`fork().`** Вы-

зов **fork()** в случае удачного завершения возвращает сыновнему процессу значение **0**, а родительскому **PID** порожденного процесса. В случае неудачного завершения, т.е. если сыновний процесс не был порожден, системный вызов **fork()** возвращает –1, код ошибки устанавливается в переменной **errno**.

Задача 1. Породить сыновний процесс, затем в обоих процессах вывести на печать собственный PID и PID родительского процесса.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
printf("PID=%d; PPID=%d \n",getpid(), getppid());
/*печать PID  текущего процесса и PID процесса-
предка */
fork();
/*создание нового процесса, с этого момента два
процесса функционируют параллельно и независимо*/
printf("PID=%d; PPID=%d \n",getpid(), getppid());
/*оба процесса печатают PID  текущего процесса и PID
процесса-предка*/
return 0;
}
```

Оба процесса узнают свой собственный идентификатор процесса с помощью вызова **getpid()**, а идентификатор родительского процесса – с помощью вызова **getppid()**.

Механизм замены тела процесса

Семейство системных вызовов **exec()** производит замену тела вызывающего процесса, после чего данный процесс начинает выполнять другую программу, начиная с ее точки входа. После окончания выполнения новой программы возврата к первоначальной программе затем не происходит.

Выполнение «нового» тела происходит в рамках уже существующего процесса, т.е. сохраняется идентификатор процесса, а также идентификатор родительского процесса, ранее открытые файлы, приоритет и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных, а также изменяются следующие атрибуты процесса:

- диспозиция сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, так как в новой программе могут от-

сутствовать ранее установленные функции-обработчики сигналов;

- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен s-бит;
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного вызова **fcntl()** был установлен флаг **close-on-exec**. Соответствующие файловые дескрипторы будут помечены как свободные.

Часто системный вызов **fork()** используется совместно с системным вызовом **exec().**

Задача 2. Породить сыновний процесс, в котором запустить на выполнение программу ls.

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     pid_t pid;
     if((pid=fork())==0)
     {
     /* процесс-потомок */
     execl("/bin/ls", "ls", "-l",
     (char*)0);
     /*  или execlp("ls","ls", "-l",(char*)0);*/

     printf("это напечатается в случае неудачного
обращения к предыдущей функции, к примеру, если не
был найден файл ls");
     }
}
```

Завершение процесса. Ожидание завершения потомков

Для завершения выполнения процесса предназначен системный вызов **_exit(int code)**. С помощью параметра этого вызова процесс может передать породившему его процессу информацию о статусе своего завершения. Принято, хотя и не является обязательным правилом, чтобы процесс возвращал нулевое значение при нормальном завершении, и ненулевое – в случае какой-либо ошибки или нештатной ситуации.

В стандартной библиотеке Си имеется функция **exit()**, которая, помимо обращения к системному вызову **_exit()**, осуществля-

ет ряд дополнительных действий, таких как, например, очистка стандартных буферов ввода-вывода.

Кроме обращения к вызову **_exit()**, другими причинами завершения процесса могут быть выполнение оператора **return**, входящего в состав функции **main()**, а также получение некоторых сигналов.

Процесс-предок имеет возможность получить информацию о завершении своего потомка. Для этого служит системный вызов **pid_t wait (int *status).**

При обращении к этому вызову выполнение родительского процесса приостанавливается до тех пор, пока один из его потомков не завершится либо не будет остановлен. Если у процесса имеется несколько потомков, процесс будет ожидать завершения любого из них. PID завершившегося потомка передается в качестве возвращаемого значения **wait(),** а через параметр вызова можно получить информацию о причине завершения потомка и коде возврата.

Часто используется сочетание системных вызовов **fork()-wait(),** если процесс-сын предназначен для выполнения некоторой программы, вызываемой посредством функции **exec().** Фактически этим предоставляется процессу-родителю возможность контролировать окончание выполнения процессов-потомков.

Задача 3. Написать программу, которая поочередно запускает на выполнение несколько программ (без параметров), имена которых переданы ей в командной строке.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
int main(int argc, char **argv)
{
int i;
for (i=1; i<argc; i++) /*важный момент: нумерация в
цикле с 1, поскольку argv[0] – имя самого
исполняемого файла */
{
int status;
pid_t pid;
if(fork()>0)
{
/*процесс-предок ожидает сообщения
от процесса-потомка о завершении */
pid = wait(&status);
printf(”process %d finished\n”, pid);
```

```
continue;
}
execlp(argv[i], argv[i], (char*)0);
return -1; /*попадем сюда при неуспехе exec()*/
}
return 0;
}
```

Задача 4. Процесс-отец создает процесса-сына, который в свою очередь создает своего сына – внука главного процесса. Написать программу: отец выводит на стандартный поток вывода число 1, его сын – число 2, внук главного процесса – число 3. При этом, на стандартный поток вывода всегда должна выводиться последовательность 3 2 1.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    pid_t pid1, pid2;
    int status = -1;
    if ((pid1 = fork()) < 0){
        return -1;
    } else if (!pid1){
        if ((pid2 = fork()) < 0){
            return -1;
        } else if (!pid2){
            printf("3 ");
            exit(0);
        }
        wait(&status);
        printf("2 ");
        exit(0);
    }
    wait(NULL);
    printf("1\n");
    exit(0);
    return 0;
}
```

Упражнения

1. Написать два варианта программы, которая запускает на выполнение следующую строку shell:
   `gcc 1.c -o 1.exe -lm`

посредством вызова **execl/execlp** и **execv/execvp**.

2. В командной строке передаются имена исполняемых файлов. Написать программу, которая запускает указанные исполняемые файлы на выполнение параллельно.

ТЕМА 10. Низкоуровневый ввод/вывод

ОС UNIX предоставляет средства низкоуровневых (т.е лежащих на уровне ОС, а не вышележащих библиотек) операций с файлами. Для этого в UNIX имеются ***системные вызовы***, осуществляющие непосредственное обращение к средствам **ОС** и, в частности, к ядру.

Для каждого процесса ОС поддерживает ***таблицу открытых файлов процесса (ТОФ).*** Номер строки этой таблицы фактически является ***файловым дескриптором***, с которым оперируют все эти вызовы. Следует отметить, что при порождении процесса ОС и интерпретатор команд автоматически открывают и сопоставляют первые три дескриптора:

0 - стандартный поток ввода информации;

1 - стандартный поток вывода информации;

2 - стандартный поток вывода диагностик.

Открытие, создание, закрытие файла. Права доступа к файлам

Для выполнения операции открытия файла используется системный вызов **`open(char *name, int flag[,int perms])`**, которому в качестве параметров передаются полное имя открываемого файла, комбинация флагов, определяющих режим открытия, а также значения прав доступа, которые используются в случае, когда указан флаг «создавать файл, если он ранее не существовал». Если файл уже был создан, данные права игнорируются.

Если файл был успешно открыт, **`open`** возвращает файловый дескриптор, по которому мы будем обращаться к файлу. В случае ошибки **`open`** возвращает -1.

В UNIX права доступа к файлу задаются для трех категорий пользователей: владельца файла, группы, к которой он принадлежит, и всех остальных пользователей. Для каждой из этих групп доступ к файлу определяется наличием права на чтение данных**,** права на запись данных и права на выполнение файла. Права доступа к файлу задаются в виде двоичного числа-комбинации из 9 бит, в которой 1 в соответствующем разряде означает наличие соответствующего права, а 0 – его отсутствие. Как правило, для удобства восприятия это число записывается в виде восьмеричной константы из 3х цифр, где старшая цифра задает права владельца, средняя – права группы, младшая – права остальных.

В качестве второго параметра может выступать комбинация следующих флагов:

O_RDONLY – открыть файл только для чтения;

O_WRONLY – открыть файл только для записи;

O_RDWR – открыть файл для чтения и для записи (заметим, что **O_RDWR != O_RDONLY | O_WRONLY**);
O_APPEND – указатель по чтению/записи установить на конец файла;
O_CREAT – если файл с таким именем не существует, создать новый файл;
O_EXCL – если файл с таким именем существует, выдать ошибку;
O_TRUNC – установить нулевую длину файла (стереть содержимое).

Задача 1. Создать для записи файл с именем "a.a" и следующими правами доступа: владельцу разрешен доступ по чтению и записи, группе и остальным – только по чтению. Если файл с таким именем уже существует, выдать ошибку.

```
 . . .
int fd;
if((fd=open("a.a",O_WRONLY | O_CREAT |
     O_EXCL,0644))==-1)
/* эквивалентно creat("a.a",0644) */
{
     printf("error when creating file\n");
     exit(1);
}
else /* создание файла прошло успешно */
```

Примечание. Существует также отдельный системный вызов **`int creat(char *name, int perms)`** специально для создания файла. Его использование эквивалентно вызову **`open`** с набором флагов O_WRONLY | O_CREAT | O_TRUNC.

Для завершения работы с файлом и освобождения файлового дескриптора служит системный вызов **`int close(int fd)`**. Данный системный вызов возвращает 0 в случае успеха и -1 в случае ошибки.

Чтение и запись данных

Чтение данных из файла осуществляется при помощи системного вызова **`ssize_t read(int fd, void* buf, size_t count)`**.

В качестве параметров ему указываются дескриптор ранее открытого файла (**`fd`**), указатель на область памяти, куда следует поместить считанные из файла данные (**`buf`**) и количество байт, которое необходимо прочитать (**`count`**). Функция возвращает количество

фактически прочитанных байт. Если оно равно нулю, то файл исчерпан. Если файл не был пустым, то гарантируется, что **read** прочитает как минимум 1 и не более **count** байт.

Запись в файл осуществляется при использовании системного вызова **ssize_t write(int fd, void* buf, size_t count).**

Функция возвращает количество фактически записанных байт. При этом как минимум 1 и не более **count** байт из области памяти, на которую указывает **buf,** будет записано в файл с дескриптором **fd**.

Задача 2. Скопировать содержимое файла f1 в файл f2, где имена f1 и f2 задаются в командной строке. Файлу f2 установить такие же права доступа, как и у файла f1.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
main(int argc, char**argv)
{ char buf[1024];
   int fd1,fd2;
   ssize_t n;
   struct stat s;
   if((fd1=open(argv[1],O_RDONLY))==-1)
     {
          printf("ошибка при открытии файла\n");
          return -1;
     }
   stat(argv[1],&s); /* получение прав доступа */
   if((fd2=creat(argv[2],s.st_mode)==-1)
     {
          printf("ошибка при создании файла\n");
          return -1;
     }
   while((n=read(fd1,buf,1024))>0)
          write(fd2,buf,n);
}
```

При создании нового файла f2 будут копироваться права доступа файла f1. Если же файл f2 уже существовал, его права доступа останутся прежними. Здесь была использована функция **int stat(char *name, struct stat *s),** которая заполняет переданную ей структуру **s** информацией из описателя файла с именем **name**. Поле **st_mode** структуры **struct stat** содержит информацию о типе файла и о правах доступа к нему.

Задача 3. Написать функцию **`void file_copy(int in_fd, int out_fd)`**, осуществляющую копирование всех данных из одного файлового дескриптора (**`in_fd`**) в другой (**`out_fd`**) с использованием локального буфера размером 4096 байт.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
void
file_copy(int in_fd, int out_fd)
{
    char buf[4096];
    ssize_t n;
    while((n = read(in_fd, buf, sizeof(buf))) != 0)
    {
        if (n == -1) {
            exit(errno);
        }
        char *p = buf;
        while (n) {
            ssize_t nw = write(out_fd, p, n);
            if (nw == -1) {
                exit(errno);
            }
            p += nw;
            n -= nw;
        }
    }
}
```

Позиционирование файлового указателя

При открытии файла текущая позиция файлового указателя (т.е. точка, начиная с которой будет осуществляться очередная операция чтения/записи) устанавливается на начало либо конец файла (в зависимости от режима открытия).

Изменить текущую позицию файлового указателя можно при помощи системного вызова **`off_t lseek(int fd, off_t offset, int whence)`**. Первый аргумент этого вызова задает дескриптор файла, второй – смещение файлового указателя в байтах, а третий аргумент указывает, относительно какой точки задается смещение, указанное во втором параметре.

Значения третьего аргумента могут быть такими:

SEEK_SET – смещение задается относительно начала файла;

SEEK_CUR – смещение задается относительно текущей позиции файлового указателя;

SEEK_END – смещение задается относительно конца файла.

Отметим, что второй аргумент может быть как положительным, так и отрицательным значением.

Функция допускает перемещение файлового указателя вперед за текущий конец файла, однако реального изменения размеров файла при этом не происходит. Если в дальнейшем будет производиться запись с такой новой позиции, промежуток между началом этой записи и прежним концом файла будет заполнен нулями. **lseek** за начало файла делать нельзя – будет ошибка.

В случае успеха функция возвращает новую позицию файлового указателя, т.е. его смещение в байтах от начала файла, в случае неудачи возвращается –1.

lseek можно использовать для получения текущего положения в файле:

```
off_t pos = lseek(fd, 0, SEEK_CUR);
```

А для получения размера файла необходимо выполнить:

```
off_t len = lseek(fd, 0, SEEK_END);
```

Дублирование дескрипторов

UNIX предоставляет два системных вызова, позволяющих дублировать файловые дескрипторы: **int dup(int fd)** и **int dup2(int oldfd, int newfd).**

Отличия их в следующем. Вызов **dup** ищет в таблице открытых файлов процесса первый (от начала) свободный дескриптор и ассоциирует его с тем же самым файлом (или другим объектом, например, каналом), что и дескриптор **fd**, переданный ему в качестве параметра.

Вызов **dup2** создает копию дескриптора **oldfd** в дескрипторе **newfd**, при этом, если ранее дескриптор **newfd** был ассоциирован с каким-то другим открытым файлом, то перед переназначением он освобождается.

Упражнения

1. С помощью системных вызовов ввода/вывода низкого уровня переписать первые 100 строк исходного файла в файл f1, остальные строки в файл f2. Длина строк не превышает 1024 символа. Имена файлов задаются в командной строке.
2. С помощью системных вызовов ввода/вывода низкого уровня обрезать строки в исходном файле до 50-ти символов. Длина строк не превышает 1024 символа. Права доступа к файлу не должны изменяться.

ТЕМА 11. Простейшие средства взаимодействия процессов

Программные каналы

Одним из простейших средств взаимодействия процессов в операционной системе UNIX является механизм каналов. Неименованный канал есть некая сущность, в которую можно помещать и извлекать данные, для чего служат два файловых дескриптора, ассоциированных с каналом: один для записи в канал, другой – для чтения. Для создания канала служит системный вызов **`int pipe(int *fd)`**.

Данный системный вызов создает канал (который фактически является некоторым буфером в оперативной памяти ограниченного размера) и возвращает через параметр **`fd`** массив из двух ассоциированных с ним файловых дескрипторов: один для записи в канал – **`fd[1]`**, другой для чтения – **`fd[0]`**.

Эти дескрипторы являются дескрипторами открытых файлов, с которыми можно работать, используя такие системные вызовы как **`read()`**, **`write()`**, **`dup()`** и так далее. Они, как и прочие дескрипторы открытых файлов, наследуются при порождении сыновнего процесса (что и позволяет использовать каналы как средство общения между процессами).

В случае успеха системный вызов **`pipe`** возвращает 0, в случае ошибки – -1.

Однако существуют различия между обычным файлом и каналом. В отличие от файла, к неименованному каналу невозможен доступ по имени, т.е. единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы. После того, как будут закрыты все дескрипторы, ассоциированные с каналом, ОС автоматически освободит занимаемый каналом буфер. Кроме того, канал реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же самой последовательности, в какой они были записаны. Это означает, что для файловых дескрипторов, ассоциированных с каналом, не определена операция позиционирования **`lseek()`** (при попытке обратиться к этому вызову произойдет ошибка).

Кроме того, существует ряд отличий при организации операций чтения и записи в канал.

При чтении из канала:

- При чтении из пустого канала процесс блокируется до момента появления в канале данных.
- Если из канала читается порция данных меньшая, чем находящаяся в канале, то эта порция считывается по страте-

гии FIFO, а оставшаяся порция непрочитанных данных остается в канале.

- Если делается попытка прочитать больше данных, чем имеется в канале, то будет прочитано (т.е. изъято из канала) доступное количество данных.
- Отметим, что блокировка происходит лишь при условии, что есть хотя бы один открытый дескриптор записи в канал. Если закрывается последний дескриптор записи в данный канал, то в канал помещается код конца файла EOF. В этом случае процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и признак конца файла, благодаря которому блокирования при чтении в этом случае не происходит. Соответственно, если заблокированы два и более процесса на чтение, то порядок разблокировки определяется конкретной реализацией.

При записи в канал:

- Если процесс пытается записать большее число байтов, чем доступное свободное пространство канала (но не превышающее предельный размер канала), то записывается возможное количество данных, после чего процесс, осуществляющий запись, блокируется до тех пор, пока в канале не появится достаточное количество места для завершения операции записи.
- Если же процесс пытается записать в канал порцию данных, превышающую предельный размер канала, то будет записано доступное количество данных, после чего процесс заблокируется до появления в канале свободного места любого размера (пусть даже и всего 1 байт), затем процесс разблокируется, вновь производит запись на доступное место в канале, и если данные для записи еще не исчерпаны, вновь блокируется до появления свободного места и т.д., пока не будут записаны все данные, после чего происходит возврат из вызова **`write()`**.
- Если процесс пытается осуществить запись в канал, с которым не ассоциирован ни один дескриптор чтения, то он получает сигнал **`SIGPIPE`** (тем самым ОС уведомляет его о недопустимости такой операции)**.**

Процесс может избежать блокирований при работе с каналом, изменив для канала режим блокировки с использованием системного вызова **`fcntl()`**. В неблокирующем режиме управление будет сразу возвращено процессу с возможной установкой соответствую-

щего кода ошибки (в зависимости от типа операции и возможных переполнений).

Чаще всего канал используется для обмена данными между несколькими процессами. Для этого используется тот факт, что при порождении сыновнего процесса в нем наследуется таблица файловых дескрипторов процесса-отца, т.е. все файловые дескрипторы, доступные процессу-отцу, будут доступны и процессу-сыну. Таким образом, если перед порождением потомка был создан канал, файловые дескрипторы для доступа к каналу будут унаследованы и сыном. В итоге обоим процессам оказываются доступны дескрипторы, связанные с каналом, и они могут использовать канал для обмена данными.

Примечание. Обмен данными через канал возможен, разумеется, не только между процессом-отцом и его потомком, но и вообще между любыми родственными процессами – единственным требованием здесь является необходимость создавать канал в порождающем процессе прежде, чем дескрипторы канала будут унаследованы порожденными процессами.

Задача 1. Реализовать конвейерное выполнение программ **print|wc,** при котором две программы работают параллельно, причем содержимое стандартного вывода программы **print** будет являться стандартным вводом для программы **wc**. Программа **print** печатает текст. Программа **wc** подсчитывает количество строк, слов и символов в своем стандартном вводе.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
int fd[2];
pipe(fd);  /*организован канал*/
if (fork()== 0)
{
/*процесс-сын */
dup2(fd[1], 1); /* отождествили стандартный вывод с
     файловым дескриптором канала, предназначенным
     для записи */
close(fd[1]);   /* закрыли файловый дескриптор
     канала, предназначенный для записи */
close(fd[0]);   /* закрыли файловый дескриптор
     канала, предназначенный для чтения */
exelp("print", "print", 0); /* запустили программу
     print */
exit(1);
}
/*процесс-родитель*/
```

```
dup2(fd[0], 0); /* отождествили стандартный ввод с
     файловым дескриптором канала,   предназначенным
     для чтения*/
close(fd[0]);   /* закрыли файловый дескриптор
     канала, предназначенный для чтения */
close(fd[1]);  /* закрыли файловый дескриптор
     канала, предназначенный для записи */
execl("/usr/bin/wc", "wc", 0); /* запустили
     программу wc */
}
```

Сигналы

Сигналы представляют собой средство уведомления процесса о наступлении некоторого события в системе. Инициатором посылки сигнала может выступать как другой процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например, завершение порожденного процесса, прерывание работы процесса нажатием комбинации Ctrl-C, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал. Кроме того, зарезервирован один или несколько номеров сигналов, семантика которых определяется пользовательскими процессами по своему усмотрению (например, процессы могут посылать друг другу сигналы с целью синхронизации).

Количество различных сигналов в современных версиях UNIX около 30, каждый из них имеет уникальное имя и номер. Описания представлены в файле **<signal.h>**. В таблице ниже приведено несколько примеров сигналов[13]:

Числовое значение	Константа	Значение сигнала
2	SIGINT	Прерывание выполнения по нажатию Ctrl-C
3	SIGQUIT	Аварийное завершение работы
9	SIGKILL	Уничтожение процесса
14	SIGALRM	Прерывание от программного таймера
18	SIGCHLD	Завершился процесс-потомок

Сигналы являются механизмом асинхронного взаимодействия, т.е. момент прихода сигнала процессу заранее неизвестен. Однако процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход. В этом

[13] Следует заметить, что в разных версиях UNIX имена сигналов могут различаться.

плане сигналы можно рассматривать как программный аналог аппаратных прерываний.

При получении сигнала процессом возможны три варианта реакции на полученный сигнал:

- процесс реагирует на сигнал стандартным образом, установленным по умолчанию (для большинства сигналов действие по умолчанию – это завершение процесса);
- процесс может заранее установить специальный способ обработки конкретного сигнала, в этом случае по приходу этого сигнала вызывается функция-обработчик, определенная процессом (при этом говорят, что сигнал перехватывается);
- процесс может проигнорировать сигнал.

Для каждого сигнала процесс может устанавливать свой вариант реакции, например, некоторые сигналы он может игнорировать, некоторые перехватывать, а на остальные установить реакцию по умолчанию. При этом по ходу своей работы процесс может изменять вариант реакции на тот или иной сигнал. Однако, некоторые сигналы невозможно ни перехватить, ни игнорировать. Они используются ядром ОС для управления работой процессов (например, **`SIGKILL`**, **`SIGSTOP`**).

Если в процесс одновременно доставляется несколько различных сигналов, то порядок их обработки не определен. Если же обработки ждут несколько экземпляров одного и того же сигнала, то ответ на вопрос, сколько экземпляров будет доставлено в процесс – все или один – зависит от конкретной реализации ОС.

В любом случае, в момент прихода сигнала, для которого задана пользовательская функция-обработчик, нормальное выполнение процесса прерывается, и управление переходит на точку входа обработчика сигнала. По выходу из функции-обработчика выполнение процесса возобновляется с той точки, на которой оно было прервано.

Ситуация, когда сигнал приходит в момент выполнения системного вызова, в разных версиях UNIX обрабатывается поразному: например, обработка сигнала может быть отложена до завершения системного вызова; либо системный вызов автоматически перезапускается после его прерывания сигналом; либо системный вызов вернет –1, а в переменной **`errno`** будет установлено значение **`EINTR`**.

Для отправки сигнала служит системный вызов **`int kill (pid_t pid, int sig)`**. Первым параметром вызова служит идентификатор процесса, которому посылается сигнал (процесс может

послать сигнал и самому себе). Если значение первого параметра есть 0, сигнал будет передан всем процессам, которые принадлежат той же группе, что и процесс, посылающий сигнал, за исключением процессов с идентификаторами 0 и 1.

Во втором параметре передается номер посылаемого сигнала. Если этот параметр равен 0, то будет выполнена проверка корректности обращения к **kill()** (в частности, существование процесса с идентификатором **pid**), но никакой сигнал в действительности посылаться не будет.

Если процесс-отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых реальный или эффективный идентификатор владельца процесса совпадает с реальным или эффективным идентификатором владельца процесса-отправителя.

Для определения реакции на получение того или иного сигнала в процессе служит системный вызов **void (*signal (int sig, void (*disp) (int))) (int)**. Аргумент **sig** – номер сигнала, для которого устанавливается реакция, а **disp** – либо определенная пользователем функция-обработчик сигнала, либо одна из констант: **SIG_DFL** (установить для данного сигнала обработку по умолчанию, т.е. стандартную реакцию системы) и **SIG_IGN** (игнорировать данный сигнал). При успешном завершении функция возвращает указатель на предыдущий обработчик данного сигнала (он может использоваться процессом, например, для последующего восстановления прежней реакции на сигнал).

Определенная пользователем функция-обработчик сигнала должна принимать один целочисленный аргумент (в нем будет передан фактический номер обрабатываемого сигнала, что позволяет использовать один обработчик для разных сигналов), и не возвращать никаких значений.

В ранних версиях UNIX: каждый раз в момент получения сигнала его диспозиция (т.е. способ реакции на сигнал) сбрасывалась в действие по умолчанию, т.о. если процесс желал многократно обрабатывать сигнал своим собственным обработчиком, он должен был каждый раз при обработке сигнала заново устанавливать реакцию на него.

Задача 2. Написать программу, которая завершается на пятое нажатие комбинации клавиш Ctrl-C.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
int count = 1;
```

```
void SigHndlr (int s)   /* обработчик сигнала */
{
printf("\n I got SIGINT %d time(s) \n",
++ count);
if (count == 5) signal (SIGINT, SIG_DFL);
/* ставим обработчик сигнала по умолчанию */
else signal (SIGINT, SigHndlr);
/* восстанавливаем обработчик сигнала */
}

int main(int argc, char **argv)
{
signal (SIGINT, SigHndlr); /* установка реакции на
сигнал */
while (1); /*”тело программы” */
return 0;
}
```

Задача 3. Реализовать игру «пинг-понг» между двумя процессами (процессы посылают друг другу целое число, всякий раз увеличивая его на 1, и когда число достигнет некоего максимума, оба процесса завершаются), используя лишь один канал.

Данная программа дает пример использования каналов и сигналов для осуществления связи между процессами – весьма типичной ситуации в системе. При этом канал играет роль среды двусторонней передачи информации, но при таком его использовании процессам требуется дополнительное средство синхронизации для определения очередности чтения/записи в канал. Таким средством синхронизации будут сигналы.

```
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_CNT 100
int target_pid, cnt;
int fd[2];
int status;

void SigHndlr(int s)
{
/* в обработчике сигнала происходит и чтение, и
запись */
     signal(SIGUSR1, SigHndlr);

     if (cnt < MAX_CNT)
     {
```

```
          read(fd[0], &cnt, sizeof(int));
          printf("%d \n", cnt);
          cnt++;
          write(fd[1], &cnt, sizeof(int));
/* посылаем сигнал второму: пора читать из канала */
          kill(target_pid, SIGUSR1);
     }
     else
          if (target_pid == getppid())
          {
/* условие окончания игры проверяется потомком */
printf("Child is going to be terminated\n");
               close(fd[1]); close(fd[0]);
               /* завершается потомок */
               exit(0);
          } else
          kill(target_pid, SIGUSR1);
}

int main(int argc, char **argv)
{
     pipe(fd); /* организован канал */
     signal (SIGUSR1, SigHndlr);
/* установлен обработчик сигнала для обоих процессов
*/
     cnt = 0;

     if (target_pid = fork())
     {
/* Предку остается только ждать завершения потомка
*/
          while(wait(&status) == -1);
printf("Parent is going to be terminated\n");
          close(fd[1]); close(fd[0]);
          return 0;
     }
     else
     {
          /* процесс-потомок узнает PID родителя */
          target_pid = getppid();
          /* потомок начинает пинг-понг */
          write(fd[1], &cnt, sizeof(int));
          kill(target_pid, SIGUSR1);
          for(;;); /* бесконечный цикл */
     }
}
```

Упражнения

1. Написать программу, реализующую следующую команду Shell:
   a. pr1 arg1 arg2 | pr2; pr3>>f1.dat
   b. pr1<f1.dat | pr2>f2.dat; pr3
   c. pr1 | pr2 | . . . | prn
   d. pr1 arg1>f1.dat; pr2 | pr3 | pr4>>f1.dat, где pr_i – имена исполняемых файлов, а f_i.dat – имена файлов данных. Имена в указанном порядке задаются в командной строке.
2. Написать программу, обрабатывающую каждое четное возникновение сигнала SIGTRAP формированием двух сыновних процессов, связанных каналом. Завершить работу по 3-му нажатию CTRL+C.
3. Написать программу подсчета поступивших сигналов SIGTRAP между 2-ым и 4-ым нажатием CTRL+C. Завершить работу по 7-му нажатию CTRL+C.
4. Написать программу параллельного поиска подстроки в файлах, имена которых заданы в командной строке. Поиск подстроки в каждом из заданных файлов организовать в процессе – потомке, создав для каждого файла свой процесс. Программа завершается, как только подстрока будет найдена в одном из файлов. Все незавершенные процессы должны быть прерваны.
5. Написать программу слияния двух файлов в третий. Файл-результат формируется чередованием отрезков по N символов из первого и второго файлов (если один из файлов длиннее другого, то его оставшаяся часть приписывается в конец файла-результата). Имена файлов и величина N задаются в командной строке. Исходные файлы читаются разными процессами; эти же процессы по очереди записывают по N байт из обрабатываемых ими файлов в файл-результат. Синхронизацию их работы организовать с помощью сигналов.
6. Написать программу копирования содержимого одного файла в другой. Копирование осуществляет вспомогательный процесс. Если во время копирования встретилась строка, длина которой больше N, этот процесс сообщает процессу-родителю о такой ситуации. Головной процесс запрашивает у пользователя, что делать с этой строкой:
   D – не записывать строку в формируемый файл;
   C – записать только первые N символов;
   A – прекратить копирование;
   и сообщает вспомогательному процессу о принятом решении. Головной процесс ждет окончания работы, сообщает пользователю о том, что копирование завершено, и

завершается сам. Имена файлов и величина N задаются в командной строке.

ТЕМА 12. Средства межпроцессного взаимодействия System V

Рассматриваемую в этой главе группу средств межпроцессного взаимодействия называют System V IPC (interprocess communication), так как изначально эти средства были реализованы именно в UNIX System V, однако к настоящему моменту они включены практически во все версии UNIX.

System V IPC включает в себя **очереди сообщений**, **семафоры, разделяемую память**.

Именование разделяемых объектов

Для всех средств IPC приняты общие правила именования объектов, позволяющие процессу получить доступ к такому объекту. Для именования объекта IPC используется ключ, представляющий собой целое число. Ключи являются уникальными во всей UNIX-системе идентификаторами объектов IPC, и, зная ключ для некоторого объекта, процесс может получить к нему доступ. При этом процессу возвращается дескриптор объекта, который в дальнейшем используется для всех операций с ним. Проведя аналогию с файловой системой, можно сказать, что ключ аналогичен имени файла, а получаемый по ключу дескриптор – файловому дескриптору, получаемому во время операции открытия файла.

Ключ для каждого объекта IPC задается в момент его создания тем процессом, который его порождает, а все процессы, желающие получить в дальнейшем доступ к этому объекту, должны указывать тот же самый ключ.

Как видно, встает проблема именования разделяемого ресурса: необходим некий механизм получения заведомо уникального ключа для именования ресурса, но вместе с тем нужно, чтобы этот механизм позволял всем процессам, желающим работать с одним ресурсом, получить одно и то же значение ключа.

Для решения этой задачи служит функция **`key_t ftok(char *filename, int proj).`** Эта функция генерирует значение ключа по путевому имени некоторого существующего файла и добавочному целому, передаваемым в качестве параметров. Она использует для создания значения ключа номер индексного дескриптора указанного файла и информацию о файловой системе, в которой он находится. Если файл с указанным именем не существует или недоступен вызывающему процессу, функция возвращает -1. Гарантируется, что при повторном запуске **`ftok()`** с теми же параметрами будет получено то же самое значение ключа[14]. Предполагается, что

[14] при условии, что указанный файл между двумя обращениями к ftok() не удалялся и не пересоздавался заново, т.к. при этом он получит другой номер файлового дескриптора

передаваемое путевое имя файла известно всем процессам вызывающей программы и имеет в рамках ее некоторый смысл (например, это может быть имя демона-сервера, имя файла данных, используемого сервером и т.п.).

Смысл же второго аргумента функции **ftok()** – добавочного целого – в том, что он позволяет генерировать разные значения ключа по одному и тому же значению первого параметра – имени файла. Тем самым, используя одно и то же путевое имя, можно получить несколько ключей, если нам требуется доступ к нескольким различным ресурсам в рамках одной и той же программы. Кроме того, это позволяет программисту поддерживать несколько версий своей программы, которые будут использовать одно и то же имя файла, но разные добавочные параметры для генерации ключа, и тем самым получат возможность в рамках одной системы работать с разными разделяемыми ресурсами.

Функция **ftok()** не является системным вызовом, а предоставляется библиотекой.

Общие принципы работы с разделяемыми ресурсами

Как уже говорилось, общим для всех ресурсов является механизм именования. Кроме того, для каждого IPC-ресурса поддерживается идентификатор его владельца и структура, описывающая права доступа к нему. Подобно файлам, права доступа задаются отдельно для владельца, его группы и всех остальных пользователей; однако, для разделяемых ресурсов поддерживается только две категории доступа: по чтению и записи. Априори считается, что возможность изменять свойства ресурса и удалять его имеется только у процесса, эффективный идентификатор пользователя которого совпадает с идентификатором владельца ресурса. Владельцем ресурса назначается пользователь, от имени которого выполнялся процесс, создавший ресурс, однако создатель может передать права владельца другому пользователю. В заголовочном файле **<sys/ipc.h>** определен тип **struct ipc_perm**, который описывает права доступа к любому IPC-ресурсу. Поля в этой структуре содержат информацию о создателе и владельце ресурса и их группах, правах доступа к ресурсу и его ключе.

Для создания разделяемого ресурса с заданным ключом, либо подключения к уже существующему ресурсу с таким ключом используются ряд системных вызовов, имеющих общий суффикс **get**. Общими параметрами для всех этих вызовов являются ключ и флаги.

В качестве значения ключа при создании любого IPC-объекта может быть указано значение **IPC_PRIVATE**. При этом создается ре-

сурс, который будет доступен только породившему его процессу. Такие ресурсы обычно порождаются родительским процессом, который затем сохраняет полученный дескриптор в некоторой переменной и порождает своих потомков. Так как потомкам доступен уже готовый дескриптор созданного объекта, они могут непосредственно работать с ним, не обращаясь предварительно к «**get**»-методу. Таким образом, созданный ресурс может совместно использоваться родительским и порожденными процессами. Однако, если один из этих процессов повторно вызовет «**get**»-метод с ключом **IPC_PRIVATE**, в результате будет получен другой, совершенно новый разделяемый ресурс, так как при обращении к «**get**»-методу с ключом **IPC_PRIVATE** всякий раз создается новый объект нужного типа[15].

Если при обращении к «**get**»-методу указан ключ, отличный от **IPC_PRIVATE**, происходит следующее:

- Происходит поиск объекта с заданным ключом среди уже существующих объектов нужного типа. Если объект с указанным ключом не найден, и среди флагов указан флаг **IPC_CREAT,** будет создан новый объект. При этом значение параметра флагов должно содержать побитовое сложение флага **IPC_CREAT** и константы, указывающей права доступа для вновь создаваемого объекта.

- Если объект с заданным ключом не найден, и среди переданных флагов отсутствует флаг **IPC_CREAT**, «**get**»-метод вернет –1, а в переменной **errno** будет установлено значение **ENOENT.**

- Если объект с заданным ключом уже существует, «**get**»-метод вернет дескриптор для этого существующего объекта. Если такая ситуация нежелательна (т.е. означает случайную коллизию имен), следует указать в параметре флагов наряду с флагом **IPC_CREAT** и правами доступа еще и флаг **IPC_EXCL** – в этом случае «**get**»-метод вернет -1, если объект с таким ключом уже существует (переменная **errno** будет установлена в значение **EEXIST**).

- при подключении к уже существующему объекту дополнительно проверяются права доступа к нему. В случае, если процесс, запросивший доступ к объекту, не имеет на то

[15] гарантируется, что функция **ftok()** ни при каких значениях своих входных параметров не может сгенерировать ключ, совпадающий с **IPC_PRIVATE.**

прав, «**get**»-метод вернет –1, а в переменной **errno** будет установлено значение **EACCESS.**

Для каждого типа объектов IPC существует некое ограничение на максимально возможное количество одновременно существующих в системе объектов данного типа. Если при попытке создания нового объекта указанное ограничение будет превышено, «**get**»-метод, совершавший попытку создания объекта, вернет -1, а в переменной **errno** будет указано значение **ENOSPC**.

Даже если ни один процесс не подключен к разделяемому ресурсу, система не удаляет его автоматически. Удаление объектов IPC является обязанностью одного из работающих с ним процессов и для этого системой предоставляются соответствующие функции по управлению объектами System V IPC.

Очередь сообщений

Очередь сообщений представляет собой некое хранилище типизированных сообщений, организованное по принципу FIFO. Любой процесс может помещать новые сообщения в очередь и извлекать из очереди имеющиеся там сообщения. Каждое сообщение имеет тип, представляющий собой некоторое целое число. Благодаря наличию типов сообщений, очередь можно интерпретировать двояко – рассматривать ее либо как сквозную очередь неразличимых по типу сообщений, либо как некоторое объединение подочередей, каждая из которых содержит элементы определенного типа. Извлечение сообщений из очереди происходит согласно принципу FIFO – в порядке их записи, однако процесс-получатель может указать, из какой подочереди он хочет извлечь сообщение, или, иначе говоря, сообщение какого типа он желает получить – в этом случае из очереди будет извлечено самое «старое» сообщение нужного типа (см. Рис.2).

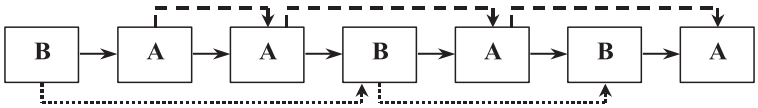


Рис.2 Типизированные очереди сообщений

Рассмотрим набор системных вызовов, поддерживающий работу с очередями сообщений.

Доступ к очереди сообщений

Для создания новой или для доступа к существующей очереди используется системный вызов **int msgget (key_t key, int**

msgflag). В случае успеха вызов возвращает положительный дескриптор очереди, который может в дальнейшем использоваться для операций с ней, в случае неудачи – -1. Первым аргументом вызова является ключ, вторым – флаги, управляющие поведением вызова. Подробнее детали процесса создания/подключения к ресурсу описаны выше.

Отправка сообщения

Для отправки сообщения используется функция **int msgsnd (int msgid, const void *msgp, size_t msgsz, int msgflg)**. Ее первый аргумент – идентификатор очереди, полученный в результате вызова **msgget()**. Второй аргумент – указатель на буфер, содержащий реальные данные (тело сообщения) и тип сообщения, подлежащего посылке в очередь, в третьем аргументе указывается длина тела сообщения в байтах.

Буфер интерпретируется как структура, соответствующая следующему шаблону:

```
#include <sys/msg.h>
struct msgbuf {
     long msgtype; // тип сообщения
     char msgtext[1]; //данные (тело сообщения)
};
```

В реальности такая структура не означает, что телом сообщения обязательно должен быть лишь один символ (и вообще текст, состоящий из символов). Приложение может определить собственную структуру для сообщения, содержащую в качестве тела любые поля необходимой длины – важно лишь, чтобы первым полем всегда выступало целое типа long, соответствующее типу сообщения. Остальные данные в этой структуре системой никак не интерпретируются и передаются в качестве тела сообщения «как есть».

В заголовочном файле **<sys/msg.h>** определена константа **MSGMAX**, описывающая максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве **msgtext** превышает это значение, системный вызов вернет –1.

Четвертый аргумент данного вызова может принимать значения 0 или **IPC_NOWAIT**. В случае отсутствия флага **IPC_NOWAIT** вызывающий процесс будет блокирован (т.е. приостановит работу), если для посылки сообщения недостаточно системных ресурсов, т.е. если полная длина сообщений в очереди будет больше максимально допустимого. Если же флаг **IPC_NOWAIT** будет установлен, то в такой ситуации выход из вызова произойдет немедленно, и возвращаемое значение будет равно –1.

В случае удачной записи возвращаемое значение вызова равно 0.

Получение сообщения.

Для получения сообщения имеется функция **int msgrcv (int msgid, void *msgp, size_t msgsz, long msgtyp, int msgflg)**. Первые три аргумента – это дескриптор очереди, указатель на буфер, куда следует поместить данные, и максимальный размер (в байтах) тела сообщения, которое можно туда поместить. Буфер, используемый для приема сообщения, должен иметь структуру, описанную выше.

Четвертый аргумент указывает тип сообщения, которое процесс желает получить. Если значение этого аргумента есть 0, то будет получено сообщение любого типа. Если значение аргумента **msgtyp** больше 0, из очереди будет извлечено сообщение указанного типа. Если же значение аргумента **msgtyp** отрицательно, то тип принимаемого сообщения определяется как наименьшее значение среди типов, которые меньше модуля **msgtyp**. В любом случае, как уже говорилось, из подочереди с заданным типом (или из общей очереди, если тип не задан) будет выбрано самое старое сообщение.

Последним аргументом является комбинация (побитовое сложение) флагов. Если среди флагов не указан **IPC_NOWAIT**, и в очереди не найдено ни одного сообщения, удовлетворяющего критериям выбора, процесс будет заблокирован до появления такого сообщения. Однако, если такое сообщение существует, но его длина превышает указанную в аргументе **msgsz**, то процесс заблокирован не будет, и вызов сразу вернет –1. Сообщение при этом останется в очереди. Если же флаг **IPC_NOWAIT** указан, то вызов сразу вернет –1.

Процесс может также указать флаг **MSG_NOERROR** – в этом случае он может прочитать сообщение, даже если его длина превышает указанную емкость буфера. В этом случае в буфер будет записано первые **msgsz** байт из тела сообщения, а остальные данные отбрасываются.

В случае удачного чтения возвращаемое значение вызова равно фактической длине тела полученного сообщения в байтах.

Управление очередью сообщений

Для управления очередью сообщений используется функция **int msgctl(int msgid, int cmd, struct msgid_ds *buf)**.

Данный вызов служит для получения или изменения процессом управляющих параметров, связанных с очередью и уничтожения очереди. Его аргументы – идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди. Тип **msgid_ds** описан в заголовочном файле **<sys/message.h>**, и представляет собой структуру, в полях которой

хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре **buf**;

IPC_SET – заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре **buf**;

IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

Задача 1. Написать программу: основной процесс читает некоторую текстовую строку из стандартного ввода, и в случае, если строка начинается с буквы **'a'**, эта строка в качестве сообщения будет передана первому потомку, если **'b'** – второму потомку**,** а по получении строки **'q'** будет осуществлен выход. Процессы-потомки распечатывают полученные строки на стандартный вывод.

Основной процесс.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

struct {
     long mtype;     /* тип сообщения */
     char Data[256];      /* сообщение */
} Message;

int main(int argc, char **argv)
{
     key_t key;  int msgid;  char str[256];

     key = ftok("/usr/mash",'s');
/*получаем уникальный ключ, однозначно определяющий
доступ к ресурсу */
     msgid=msgget(key, 0666 | IPC_CREAT);
/*создаем очередь сообщений , 0666 определяет права
доступа */

     for(;;) {
          /* запускаем вечный цикл */
          gets(str); /* читаем строку */
          strcpy(Message.Data, str);
```

```
          /* и копируем ее в буфер сообщения */
          switch(str[0]){
               case 'a':
               case 'A':
                    Message.mtype = 1;
                    /* устанавливаем тип */
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    break;
               case 'b':
               case 'B':
                    Message.mtype = 2;
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    break;
               case 'q':
               case 'Q':
                    Message.mtype = 1;
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    Message.mtype = 2;
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    sleep(10);
/* ждем получения сообщений процессами-потомками*/
                    msgctl(msgid, IPC_RMID, NULL);
/* уничтожаем очередь*/
                    return 0;
               default:
                    break;
          }
     }
}
Процесс-потомок
/* второй потомок аналогичен с точностью до
четвертого параметра в msgrcv */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

struct {
          long mtype;
          char Data[256];
} Message;

int main(int argc, char **argv)
{
     key_t key;  int msgid;
```

```
        key = ftok("/usr/mash",'s');
        /* получаем ключ по тем же параметрам */
        msgid = msgget(key, 0666 | IPC_CREAT);
        /*подключаемся к очереди сообщений */
        for(;;) {
             msgrcv(msgid, (struct msgbuf*) (&Message),
             256, 1, 0);
             if (Message.Data[0]=='q' ||
                  Message.Data[0]=='Q') break;
             printf("\nПроцесс-приемник A: %s",
                  Message.Data);
        }
        return 0;
}
```

Разделяемая память

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти. Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области памяти.

Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к **`malloc()`**), однако, как уже говорилось, сама по себе разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать

Создание общей памяти

Для создания объекта разделяемой памяти служит вызов **`int shmget (key_t key, int size, int shmemflg)`**.

Аргументы этого вызова: **`key`** – ключ для доступа к разделяемой памяти; **`size`** задает размер области памяти, к которой процесс желает получить доступ. Если в результате вызова **`shmget()`** будет создана новая область разделяемой памяти, то ее размер будет соответствовать значению **`size`**. Если же процесс подключается к существующей области разделяемой памяти, то значение **`size`** должно быть не более ее размера, иначе вызов вернет -1. Если процесс при подключении к существующей области разделяемой памяти указал в аргументе **`size`** значение, меньшее ее фактического размера, то впо-

следствии он сможет получить доступ только к первым **size** байтам этой области.

В заголовочном файле **<sys/shm.h>** определены константы **SHMMIN** и **SHMMAX**, задающие минимально возможный и максимально возможный размер области разделяемой памяти. Если процесс пытается создать область разделяемой памяти, размер которой не удовлетворяет этим границам, системный вызов **shmget()** окончится неудачей.

Третий параметр определяет флаги, управляющие поведением вызова. Подробнее алгоритм создания/подключения разделяемого ресурса был описан выше.

В случае успешного завершения вызов возвращает положительное число – дескриптор области памяти, в случае неудачи – -1.

Доступ к разделяемой памяти

При помощи вызова **char *shmat(int shmid, char *shmaddr, int shmflg)** процесс подсоединяет область разделяемой памяти, дескриптор которой указан в **shmid**, к своему виртуальному адресному пространству. После выполнения этой операции процесс сможет читать и модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.

В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Чаще всего, однако, в качестве значения этого аргумента передается 0, что означает, что система сама может выбрать адрес начала разделяемой памяти.

Примечание. Передача конкретного адреса в этом параметре имеет смысл в том случае, если, к примеру, в разделяемую память записываются указатели на нее же (например, в ней хранится связанный список) – в этой ситуации для того, чтобы использование этих указателей имело смысл и было корректным для всех процессов, подключенных к памяти, важно, чтобы во всех процессах адрес начала области разделяемой памяти совпадал.

Третий аргумент представляет собой комбинацию флагов. В качестве значения этого аргумента может быть указан флаг **SHM_RDONLY**, который указывает на то, что подсоединяемая область будет использоваться только для чтения.

Эта функция возвращает адрес, начиная с которого будет отображаться присоединяемая разделяемая память. В случае неудачи вызов возвращает -1.

Открепление разделяемой памяти

Вызов **int shmdt(char *shmaddr)** позволяет отсоединить разделяемую память, ранее присоединенную посредством вызова **shmat()**.

Параметр **shmaddr** – адрес прикрепленной к процессу памяти, который был получен при вызове **shmat()**.

В случае успешного выполнения функция возвращает 0, в случае неудачи – -1.

Управление разделяемой памятью

Вызов **int shmctl(int shmid, int cmd, struct shmid_ds *buf)** используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения. Аргументы вызова – дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти. Тип **shmid_ds** описан в заголовочном файле **<sys/shm.h>**, и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединенных к ней в данный момент, и статистика обращений к области памяти.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре **buf**;

IPC_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре **buf**. Выполнить эту операцию может процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней;

IPC_RMID – удалить разделяемую память. Как уже говорилось, удалить ресурс IPC может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем ресурса, либо процесс с правами привилегированного пользователя;

SHM_LOCK, SHM_UNLOCK – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

Семафоры

Семафоры, как правило, используются для синхронизации доступа нескольких процессов к другим разделяемым ресурсам, так

как сами по себе прочие средства IPC не предоставляют механизма синхронизации.

Классический семафор представляет собой особый вид числовой переменной, над которой определены две неделимые операции: уменьшение ее значения с возможным блокированием процесса и увеличение значения с возможным разблокированием одного из ранее заблокированных процессов.

Как правило, использование классических семафоров в качестве средства синхронизации доступа к другим разделяемым объектам предполагает следующую схему:

- с каждым разделяемым ресурсом связывается один семафор из набора;
- положительное значение семафора означает возможность доступа к ресурсу (ресурс свободен), неположительное – отказ в доступе (ресурс занят);
- перед тем как обратиться к ресурсу, процесс уменьшает значение соответствующего ему семафора, при этом, если значение семафора после уменьшения должно оказаться отрицательным, то процесс будет заблокирован до тех пор, пока семафор не примет такое значение, чтобы при уменьшении его значение оставалось неотрицательным;
- закончив работу с ресурсом, процесс увеличивает значение семафора (при этом разблокируется один из ранее заблокированных процессов, ожидающих увеличения значения семафора, если таковые имеются);
- в случае реализации взаимного исключения используется двоичный семафор, т.е. такой, что он может принимать только значения 0 и 1: такой семафор всегда разрешает доступ к ресурсу не более чем одному процессу одновременно.

Объект System V IPC представляет собой **набор семафоров,** причем каждый отдельный семафор в наборе является расширением классического понятия: если классический семафор предполагает лишь одну, рассмотренную выше, схему использования его в качестве средства синхронизации конкурирующих процессов, то для семафора System V добавляется возможность блокирования на семафоре в ожидании его обнуления.

Доступ к семафорам

Для получения доступа к массиву семафоров (или его создания) используется системный вызов **int semget (key_t key, int nsems, int semflag).**

Первый параметр функции **semget()** – ключ для доступа к разделяемому ресурсу, второй – количество семафоров в создаваемом наборе (длина массива семафоров) и третий параметр – флаги, управляющие поведением вызова. Подробнее процесс создания разделяемого ресурса описан выше. Отметим семантику прав доступа к семафорам: процесс, имеющий право доступа к массиву семафоров по чтению, может проверять значение семафоров; процесс, имеющий право доступа по записи, может как проверять, так и изменять значения семафоров.

В случае, если среди флагов указан **IPC_CREAT**, аргумент **nsems** должен представлять собой положительное число, если же этот флаг не указан, значение **nsems** игнорируется. Отметим, что в заголовочном файле **<sys/sem.h>** определена константа **SEMMSL**, задающая максимально возможное число семафоров в наборе. Если значение аргумента **nsems** больше этого значения, вызов **semget()** завершится неудачно.

В случае успеха вызов **semget()** возвращает положительный дескриптор созданного разделяемого ресурса, в случае неудачи – -1.

Операции над семафорами

Используя полученный дескриптор, можно изменять значения одного или нескольких семафоров в наборе, а также проверять их значения на равенство нулю, для чего используется системный вызов **int semop (int semid, struct sembuf *semop, size_t nops).**

Этому вызову передаются следующие аргументы:

semid – дескриптор массива семафоров;

semop – массив из объектов типа **struct sembuf**, каждый из которых задает одну операцию над семафором;

nops – длина массива **semop**. Количество семафоров, над которыми процесс может одновременно производить операцию в одном вызове **semop()**, ограничено константой **SEMOPM**, описанной в файле **<sys/sem.h>**. Если процесс попытается вызвать **semop()** с параметром **nops**, большим этого значения, этот вызов вернет неуспех.

Структура **sembuf** имеет следующий вид:

```
struct sembuf {
     short sem_num;  /* номер семафора в векторе */
     short sem_op;   /* производимая операция */
     short sem_flg;  /* флаги операции */
}
```

Общий принцип обработки этой структуры следующий:

Пусть значение семафора с номером **sem_num** равно **sem_val**.

1. если значение операции не равно нулю:
   - оценивается значение суммы **sem_val + sem_op**.
   - если эта сумма больше либо равна нулю, то значение данного семафора устанавливается равным этой сумме: **sem_val = sem_val + sem_op**
   - если же эта сумма меньше нуля, то действие процесса будет приостановлено до тех пор, пока значение суммы **sem_val + sem_op** не станет больше либо равно нулю, после чего значение семафора устанавливается равным этой сумме: **sem_val = sem_val + sem_op**
2. Если код операции **sem_op** равен нулю:
   - Если при этом значение семафора (**sem_val**) равно нулю, происходит немедленный возврат из вызова
   - Иначе происходит блокирование процесса до тех пор, пока значение семафора не обнулится, после чего происходит возврат из вызова

Таким образом, ненулевое значение поля **sem_op** обозначает необходимость прибавить к текущему значению семафора значение **sem_op**, а нулевое – дождаться обнуления семафора.

Поле sem_flg в структуре sembuf содержит комбинацию флагов, влияющих на выполнение операции с семафором. В этом поле может быть установлен флаг IPC_NOWAIT, который предписывает соответствующей операции над семафором не блокировать процесс, а сразу возвращать управление из вызова semop(). Вызов semop() в такой ситуации вернет –1. Кроме того, в этом поле может быть установлен флаг SEM_UNDO, в этом случае система запомнит изменение значения семафора, произведенные данным вызовом, и по завершении процесса автоматически ликвидирует это изменение. Это предохраняет от ситуации, когда процесс уменьшил значение семафора, начав работать с ресурсом, а потом, не увеличив значение семафора обратно, по какой-либо причине завершился. В этом случае остальные процессы, ждущие доступа к ресурсу, оказались бы заблокированы навечно.

Управление массивом семафоров.

С помощью системного вызова **int semctl (int semid, int num, int cmd, union semun arg)** можно запрашивать и изменять управляющие параметры разделяемого ресурса, а также удалять его.

Первый параметр вызова – дескриптор массива семафоров. Параметр **num** представляет собой индекс семафора в массиве, параметр **cmd** задает операцию, которая должна быть выполнена над данным семафором. Последний аргумент имеет тип **union semun** и используется для считывания или задания управляющих параметров одного семафора или всего массива, в зависимости от значения аргумента **cmd**. Тип данных **union semun** определен в файле **<sys/sem.h>** и выглядит следующим образом:

```
union semun {
     int val;  // значение одного семафора
     struct semid_ds *buf;     /* параметры массива
          семафоров в целом */
     ushort    *array;         /* массив значений
          семафоров */
}
```

где **struct semid_ds** – структура, описанная в том же файле, в полях которой хранится информация обо всем наборе семафоров в целом, а именно, количество семафоров в наборе, права доступа к нему и статистика доступа к массиву семафоров.

Приведем некоторые наиболее часто используемые значения аргумента **cmd**:

IPC_STAT – скопировать управляющие параметры набора семафоров по адресу **arg.buf**;

IPC_SET – заменить управляющие параметры набора семафоров на те, которые указаны в **arg.buf**. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя, при этом процесс может изменить только владельца массива семафоров и права доступа к нему;

IPC_RMID – удалить массив семафоров. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя;

GETALL, SETALL – считать / установить значения всех семафоров в массив, на который указывает **arg.array**;

GETVAL – возвратить значение семафора с номером **num**. Последний аргумент вызова игнорируется;

SETVAL – установить значение семафора с номером num равным **arg.val**.

В случае успешного завершения вызов возвращает значение, соответствующее конкретной выполнявшейся операции (0, если не оговорено иное), в случае неудачи – -1.

Отметим, что использование данного вызова с командами **GETVAL** и **GETALL** позволяет процессу проверить значение семафора без риска оказаться заблокированным.

Команды же **SETVAL** и **SETALL** чаще всего используются для инициализации семафоров, начальное значение которых по семантике алгоритма работы с ними должно быть отлично от нуля[16].

Задача 1. Для бинарного семафора, единичное значение которого соответствует свободному ресурсу, а нулевое – занятому, с помощью системного вызова semop() реализовать операции P и V.

```
struct sembuf   P={0,-1,0};
struct sembuf   V={0,1,0};
semop(semid,&P,1);
/* Освобождение ресурса:  */
semop(semid,&V,1);
```

Задача 2. Для бинарного семафора, нулевое значение которого соответствует свободному ресурсу, а единичное – занятому, с помощью системного вызова semop реализовать операции P и V.

```
struct sembuf  P[2]={
{0,0,0},   // ожидание освобождения ресурса
{0,1,0}}; // захват ресурса

semop(semid,P,2); /* операция неделимая и
     выполняется только над всем массивом. */

struct sembuf  V={0,-1,0};
     /* Освобождение ресурса:  */
semop(semid,&V,1);
```

Задача 3. Работа с разделяемой памятью с синхронизацией семафорами.

Программа будет оперировать с разделяемой памятью.

1 процесс – создает ресурсы “разделяемая память” и “семафоры”, далее он начинает принимать строки со стандартного ввода и записывает их в разделяемую память.

2 процесс – читает строки из разделяемой памяти.

[16] строго говоря, инициализация семафоров требуется всегда, ибо стандарт не обязывает реализацию присваивать семафору какое-либо значение в момент его создания, хотя большинство современных реализаций IPC **делают** это, присваивая семафорам начальное значение 0.

Таким образом, имеется критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Поэтому следует использовать дополнительную синхронизацию. В качестве средства синхронизации используются семафоры.

1й процесс:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX   256
int main(int argc, char **argv)
{
     key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[NMAX];

     key = ftok("/usr/ter/exmpl", 'S');
     /* создаем уникальный ключ */
     semid = semget(key, 1, 0666 | IPC_CREAT);
     /* создаем один семафор с определенными правами
     доступа */
     shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
     /* создаем разделяемую память на 256 элементов
     */
     shmaddr = shmat(shmid, NULL, 0);
     /* подключаемся к разделу памяти, в shmaddr -
     указатель на буфер с разделяемой памятью */
     semctl(semid,0,SETVAL, (int) 0);
     /* инициализируем семафор значением 0 */
     sops.sem_num = 0;
     sops.sem_flg = 0;
     do { /* запуск цикла */
          printf("Введите строку:");
          if (fgets(str, NMAX, stdin) == NULL)
          {
               /* окончание ввода */
               /* пишем признак завершения - строку
               "Q" */
               strcpy(str, "Q");
          }
          /* в текущий момент семафор открыт для
          этого процесса */
          strcpy(shmaddr, str); /* копируем строку в
          разд. память */
          /* предоставляем второму процессу
          возможность войти */
```

```
            sops.sem_op = 3; /* увеличение семафора на
            3 */
            semop(semid, &sops, 1);
            /* ждем, пока семафор будет открыт для 1го
            процесса -  для следующей итерации цикла
            */
            sops.sem_op = 0; /* ожидание обнуления
            семафора */
            semop(semid, &sops, 1);
      } while (str[0] != 'Q');
      /* в данный момент второй процесс уже дочитал
      из разделяемой памяти и отключился от нее –
      можно ее удалять*/
      shmdt(shmaddr) ; /* отключаемся от разделяемой
      памяти */
      shmctl(shmid, IPC_RMID, NULL);
      /* уничтожаем разделяемую память */
      semctl(semid, 0, IPC_RMID, (int) 0);
      /* уничтожаем семафор */
      return 0;
}
```

2й процесс:

```
/* необходимо корректно определить существование
ресурса, если он есть - подключиться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX   256

int main(int argc, char **argv)
{
      key_t key;
      int semid, shmid;
      struct sembuf sops;
      char *shmaddr;
      char str[NMAX];

      key = ftok("/usr/ter/exmpl",'S');
      /* создаем тот же самый ключ */
      semid = semget(key, 1, 0666 | IPC_CREAT);
      shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
      /* аналогично предыдущему процессу  -
      инициализации ресурсов */
      shmaddr = shmat(shmid, NULL, 0);
      sops.sem_num = 0;
      sops.sem_flg = 0;
      /* запускаем цикл */
```

```
        do {
             printf("Waiting… \n"); /* ожидание на
             семафоре */
             sops.sem_op = -2;
             /* будем ожидать, пока "значение семафора"
             + "значение sem_op" не станет
             положительным, т.е. пока значение семафора
             не станет как минимум 3 (3-2=1 > 0) */
             semop(semid, &sops, 1);
             /* теперь значение семафора равно 1 */
             strcpy(str, shmaddr); /* копируем строку
             из разд.памяти */
             /*критическая секция - работа с
             разделяемой памятью - в этот момент
             первый процесс к разделяемой памяти
             доступа не имеет*/
             if (str[0] == 'Q')
             {
                    /*завершение работы - освобождаем
                    разделяемую память */
                    shmdt(shmaddr);
             }
             /*после работы – обнулим семафор*/
             sops.sem_op=-1;
             semop(semid, &sops, 1);
             printf("Read from shared memory: %s\n",
             str);
        } while (str[0] != 'Q');
        return 0;
}
```

Отметим, что данный пример демонстрирует два разных приема использования семафоров для синхронизации: первый процесс блокируется в ожидании обнуления семафора, т.е. для того, чтобы он мог войти в критическую секцию, значение семафора должно стать нулевым; второй процесс блокируется при попытке уменьшить значение семафора до отрицательной величины, для того, чтобы этот процесс мог войти в критическую секцию, значение семафора должно быть не менее 3. В этом примере, помимо взаимного исключения процессов, достигается строгая последовательность действий двух процессов: они получают доступ к критической секции строго по очереди.

Упражнения

1. Реализовать с помощью IPC средств систему, поддерживающую обмен строковыми сообщениями между пользователями (chat – систему).

2. Решить вышеприведенную **задачу 1** с помощью **двух бинарных** семафоров.
3. Реализовать игру в «пинг-понг» между двумя процессами (процессы посылают друг другу целое число, всякий раз увеличивая его на 1, и когда число достигнет некоего максимума, оба процесса завершаются), используя в качестве среды передачи данных очередь сообщений.
4. Реализовать игру в «пинг-понг» между двумя процессами, используя разделяемую память. Для синхронизации работы процессов использовать механизм семафоров.
5. Написать реализацию классической задачи «читателей и писателей» с использованием механизма семафоров и разделяемой памяти (формулировку задачи и описание базового алгоритма можно найти, например, в [7]).

Приложение 1. Приоритеты и порядок вычисления операторов.

Обозначение	Операция	Порядок выполнения

`( )`
`[ ]`
`->  .`
	вызов функции
	индексное выражение
	операции доступа к полям записей
		Слева направо

`!`
`~`
`++  --`
`-`
`(тип)`
`*`
`&`
`sizeof`
	логическое “НЕ”
	поразрядное “НЕ”
	инкрементная и декрементная операции
	унарный минус
	приведение к типу
	косвенная адресация
	взятие адреса
	получение размера объекта
		Справа налево

`*  /` `%`	умножение, деление, остаток от деления	Слева направо
`+  -`	сложение, вычитание	Слева направо
`<<  >>`	поразрядные сдвиги	Слева направо
`<  <=` `>  >=`	операции отношения	Слева направо
`==  !=`	операции отношения	Слева направо
`&`	поразрядное “И”	Слева направо
`^`	поразрядное исключающее “ИЛИ”	Слева направо
`|`	поразрядное “ИЛИ”	Слева направо
`&&`	логическое “И”	Слева направо
`||`	логическое “ИЛИ”	Слева направо
`? :`	тернарный оператор	Справа налево

`=`
`+=  -=`
`*=  /=  %=`
`<<=  >>=`
`&=  ^=  |=`
	операции присваивания	Справа налево

`,`	операция запятая	Слева направо

Приоритет операций убывает сверху вниз.
Операции из одной ячейки таблицы имеют одинаковый приоритет.

ЛИТЕРАТУРА

1. Б.Керниган, Д.Ритчи. Язык программирования Си. Вильямс, 2013.
2. А.Робачевский Операционные системы UNIX. 2-е изд., перераб. и доп. Санкт-Петербург, BHV-Санкт-Петербург, 2007
3. Э.Немет, Г.Снайдер, С.Сибасс, Т.Р.Хейн. UNIX: руководство системного администратора. Для профессионалов. 3-е издание. Санкт-Петербург, Питер; Киев, Издательская группа BHV, 2002.
4. У.Стивенс. UNIX: взаимодействие процессов. Санкт-Петербург, Питер, 2002.
5. Руденко Т.В. Сборник задач и упражнений по языку Си. Москва, МГУ, 1999.
6. Н.Д.Васюкова, И.В.Машечкин, В.В.Тюляева, Е.М.Шляховая. Краткий конспект семинарских занятий по языку Си. Москва, МГУ, 1999.
7. Н.В.Вдовикина, И.В.Машечкин, А.Н.Терехин, А.Н.Томилин. Операционные системы: взаимодействие процессов. Москва, Макс-Пресс, 2008.
8. И.В.Машечкин, М.И.Петровский, П.Д.Скулачев, А.Н.Терехин. Системное Программное Обеспечение: файловые системы ОС Unix и Windows NT. Москва, Диалог-МГУ,1997г.
9. Г. Шилдт. Полный справочник по Си. Издание 4-е. «Вильямс», 2010.

Учебное издание

КАЗАЧУК Мария Андреевна
МАШЕЧКИН Игорь Валерьевич
ПОПОВ Иван Сергеевич
ТЕРЕХИН Андрей Николаевич
ТЮЛЯЕВА Вера Викторовна

ПРОГРАММИРОВАНИЕ В ОС UNIX НА ЯЗЫКЕ СИ

Учебно-методическое пособие

Второе издание, исправленное и дополненное

Издательский отдел
факультета вычислительной математики и кибернетики
МГУ имени М. В. Ломоносова
Лицензия ИД № 05899 от 24.09.01 г.
119992, ГСП-2, Москва, Ленинские горы,
МГУ имени М. В. Ломоносова, 2-й учебный корпус

Издательство «МАКС Пресс»
Главный редактор: *Е. М. Бугачева*

Подписано в печать 02.03.2020 г.
Формат 60x90 1/16. Усл. печ. л. 6,75.
Тираж 120 экз. Изд. № 051.

Издательство ООО «МАКС Пресс»
Лицензия ИД N00510 от 01.12.99 г.

119992, ГСП-2, Москва, Ленинские горы,
МГУ им. М. В. Ломоносова, 2-й учебный корпус, 527.
Тел. 8(495) 939–3890/91. Тел./Факс 8(495) 939–3891.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
115201, г. Москва, ул. Котляковская, д.3, стр. 13.