6. Системные вызовы и взаимодействие с UNIX.

В этой главе речь пойдет о процессах. Скомпилированная программа хранится на диске как обычный нетекстовый файл. Когда она будет загружена в память компьютера и начнет выполняться – она станет **процессом.**

UNIX – многозадачная система (мультипрограммная). Это означает, что одновременно может быть запущено много процессов. Процессор выполняет их в режиме **разделения времени** – выделяя по очереди квант времени одному процессу, затем другому, третьему... В результате создается впечатление **параллельного** выполнения всех процессов (на многопроцессорных машинах параллельность истинная). Процессам, ожидающим некоторого события, время процессора не выделяется. Более того, "спящий" процесс может быть временно откачан (т.е. скопирован из памяти машины) на диск, чтобы освободить память для других процессов. Когда "спящий" процесс дождется события, он будет "разбужен" системой, переведен в ранг "готовых к выполнению" и, если был откачан будет возвращен с диска в память (но, может быть, на другое место в памяти!). Эта процедура носит название "своппинг" (**swapping**).

Можно запустить несколько процессов, выполняющих программу из **одного и того же** файла; при этом все они будут (если только специально не было предусмотрено иначе) независимыми друг от друга. Так, у каждого пользователя, работающего в системе, имеется свой собственный процесс-**интерпретатор команд** (своя копия), выполняющий программу из файла */bin/csh* (или */bin/sh*).

Процесс представляет собой изолированный "мир", общающийся с другими "мирами" во Вселенной при помощи:

a) Аргументов функции *main*:

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

Если мы наберем команду

```
$ a.out a1 a2 a3
```

то функция *main* программы из файла **a.out** вызовется с

```
argc    = 4  /* количество аргументов */
argv[0] = "a.out"       argv[1] = "a1"
argv[2] = "a2"          argv[3] = "a3"
argv[4] = NULL
```

По соглашению **argv**[0] содержит имя выполняемого файла из которого загружена эта программа*.

b) Так называемого "окружения" (или "среды") char ***envp**[], продублированного также в предопределенной переменной

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

Окружение состоит из строк вида

```
"ИМЯПЕРЕМЕННОЙ=значение"
```

Массив этих строк завершается *NULL* (как и **argv**). Для получения значения переменной с именем **ИМЯ** существует стандартная функция

```
char *getenv( char *ИМЯ );
```

Она выдает либо **значение**, либо *NULL* если переменной с таким именем нет.

c) Открытых файлов. По умолчанию (неявно) всегда открыты 3 канала:

```
                    ВВОД         В Ы В О Д
  FILE *            stdin      stdout    stderr
```

```
 соответствует fd     0         1        2
 связан с        клавиатурой     дисплеем
```

```
#include <stdio.h>
main(ac, av) char **av; {
  execl("/bin/sleep", "Take it easy", "1000", NULL);
}
```

Эти каналы достаются процессу "в наследство" от запускающего процесса и связаны с дисплеем и клавиатурой, если только не были перенаправлены. Кроме того, программа может сама явно открывать файлы (при помощи *open*, *creat*, *pipe*, *fopen*). Всего программа может одновременно открыть до 20 файлов (считая стандартные каналы), а в некоторых системах и больше (например, 64). В *MS DOS* есть еще 2 предопределенных канала вывода: **stdaux** - в последовательный коммуникационный порт, **stdprn** - на принтер.

d) Процесс имеет уникальный номер, который он может узнать вызовом

```
int pid = getpid();
```

а также узнать номер "родителя" вызовом

```
int ppid = getppid();
```

Процессы могут по этому номеру посылать друг другу сигналы:

```
kill(pid /* кому */, sig /* номер сигнала */);
```

и реагировать на них

```
signal (sig /*по сигналу*/, f /*вызывать f(sig)*/);
```

e) Существуют и другие средства коммуникации процессов: семафоры, сообщения, общая память, сетевые коммуникации.

f) Существуют некоторые другие параметры (контекст) процесса: например, его текущий каталог, который достается в наследство от процесса-"родителя", и может быть затем изменен системным вызовом

```
chdir(char *имя_нового_каталога);
```

У каждого процесса есть свой **собственный** текущий рабочий каталог (в отличие от *MS DOS*, где текущий каталог одинаков для всех задач). К "прочим" характеристикам отнесем также: управляющий терминал; группу процессов (**pgrp**); идентификатор (номер) владельца процесса (**uid**), идентификатор группы владельца (**gid**), реакции и маски, заданные на различные сигналы; и.т.п.

g) Издания других запросов (системных вызовов) к операционной системе ("богу") для выполнения различных "внешних" операций.

h) Все остальные действия происходят внутри процесса и никак не влияют на другие процессы и устройства ("миры"). В частности, один процесс НИКАК не может получить доступ к памяти другого процесса, если тот не позволил ему это явно (механизм **shared memory**); адресные пространства процессов независимы и изолированы (равно и пространство ядра изолировано от памяти процессов).

Операционная система выступает в качестве **коммуникационной среды**, связывающей "миры"-процессы, "миры"-внешние устройства (включая терминал пользователя); а также в качестве распорядителя ресурсов "Вселенной", в частности - времени (по очереди выделяемого активным процессам) и пространства (в памяти компьютера и на дисках).

Мы уже неоднократно упоминали "системные вызовы". Что же это такое? С точки зрения Си-программиста - это обычные функции. В них передают аргументы, они возвращают значения. Внешне они ничем не отличаются от написанных нами или библиотечных функций и вызываются из программ одинаковым с ними способом.

С точки же зрения реализации - есть глубокое различие. Тело функции-сисвызова расположено не в нашей программе, а в резидентной (т.е. постоянно находящейся в памяти компьютера) управляющей программе, называемой **ядром операционной системы**[*].

Поведение всех программ в системе вытекает из поведения системных вызовов, которыми они пользуются. Даже то, что *UNIX* является многозадачной системой, непосредственно вытекает из наличия системных вызовов *fork*, *exec*, *wait* и спецификации их функционирования! То же можно сказать про язык Си - мобильность программы зависит в основном от набора используемых в ней **библиотечных функций** (и, в меньшей степени, от диалекта самого языка, который должен удовлетворять **стандарту** на язык Си). Если две разные системы предоставляют

все эти функции (которые могут быть по-разному реализованы, но должны делать одно и то же), то программа будет компилироваться и работать в обоих системах, более того, работать в них **одинаково.**

Сам термин "системный вызов" как раз означает "вызов системы для выполнения действия", т.е. вызов функции в ядре системы. Ядро работает в **привелегированном режиме**, в котором имеет доступ к некоторым системным таблицам*, регистрам и портам внешних устройств и диспетчера памяти, к которым обычным программам доступ аппаратно запрещен (в отличие от *MS DOS*, где все таблицы ядра доступны пользовательским программам, что создает раздолье для вирусов). Системный вызов происходит в 2 этапа: сначала в пользовательской программе вызывается библиотечная функция-"корешок", тело которой написано на ассемблере и содержит команду генерации **программного прерывания.** Это - главное отличие от нормальных Си-функций - вызов по прерыванию. Вторым этапом является реакция ядра на прерывание:

1. переход в привелегированный режим;
2. разбирательство, КТО обратился к ядру, и подключение *u-area* этого процесса к адресному пространству ядра (**context switching**);
3. извлечение аргументов из памяти запросившего процесса;
4. выяснение, ЧТО же хотят от ядра (один из аргументов, невидимый нам - это **номер** системного вызова);
5. проверка корректности остальных аргументов;
6. проверка прав процесса на допустимость выполнения такого запроса;
7. вызов тела требуемого системного вызова - это обычная Си-функция в ядре;
8. возврат ответа в память процесса;
9. выключение привелегированного режима;
10. возврат из прерывания.

Во время системного вызова (шаг 7) процесс может "заснуть", дожидаясь некоторого события (например, нажатия кнопки на клавиатуре). В это время ядро передаст управление другому процессу. Когда наш процесс будет "разбужен" (событие произошло) - он продолжит выполнение шагов системного вызова.

Большинство системных вызовов возвращают в программу в качестве своего значения признак успеха: 0 - все сделано, (-1) - сисвызов завершился неудачей; либо некоторое содержательное значение при успехе (вроде дескриптора файла в *open*(), и (-1) при неудаче. В случае неудачного завершения в предопределенную переменную *errno* заносится номер ошибки, описывающий причину неудачи (коды ошибок предопределены, описаны в include-файле <**errno.h**> и имеют вид *Ечтото*). Заметим, что при УДАЧЕ эта переменная просто **не изменяется** и может содержать любой мусор, поэтому проверять ее имеет смысл лишь в случае, если ошибка действительно произошла:

```
#include <errno.h>      /* коды ошибок */
extern int errno;
extern char *sys_errlist[];
int value;
if((value = sys_call(...)) < 0 ){
   printf("Error:%s(%d)\n", sys_errlist[errno],
                            errno );
   exit(errno); /* принудительное завершение программы */
}
```

Предопределенный массив *sys_errlist*, хранящийся в стандартной библиотеке, содержит строки-расшифровку смысла ошибок (по-английски). Посмотрите описание функции *per- ror*().

6.1. Файлы и каталоги.

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

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

typeOf( name ) char *name;
{  int type; struct stat st;
   if( stat( name, &st ) < 0 ){
           printf( "%s не существует\n", name );
           return 0;
   }
   printf("Файл имеет %d имен\n", st.st_nlink);

   switch(type = (st.st_mode & S_IFMT)){
   case S_IFREG:
        printf( "Обычный файл размером %ld байт\n",
                    st.st_size ); break;
   case S_IFDIR:
           printf( "Каталог\n" );      break;
```

```
        case S_IFCHR:   /* байтоориентированное  */
        case S_IFBLK:   /* блочноориентированное */
                printf( "Устройство\n" );   break;
        case S_IFIFO:
                printf( "FIFO-файл\n" );    break;
        default:
                printf( "Другой тип\n" );   break;
        }       return type;
    }
```

6.1.2. Напишите программу, печатающую: свои аргументы, переменные окружения, информацию о всех открытых ею файлах и используемых трубах. Для этой цели используйте системный вызов

```
struct stat st; int used, fd;
for(fd=0; fd < NOFILE; fd++ ){
  used = fstat(fd, &st) < 0 ? 0 : 1;
  ...
}
```

Программа может использовать дескрипторы файлов с номерами 0..*NOFILE*-1 (обычно 0..19). Если *fstat* для какого-то **fd** вернул код ошибки (<0), это означает, что данный дескриптор не связан с открытым файлом (т.е. не используется). *NOFILE* определено в include-файле <**sys/param.h**>, содержащем разнообразные параметры данной системы.

6.1.3. Напишите упрощенный аналог команды *ls*, распечатывающий содержимое текущего каталога (файла с именем ".") без сортировки имен по алфавиту. Предусмотрите чтение каталога, чье имя задается как аргумент программы. Имена "." и ".." не выдавать.

Формат каталога описан в header-файле <**sys/dir.h**> и в "канонической" версии выглядит так: каталог - это файл, состоящий из структур *direct*, каждая описывает одно имя файла, входящего в каталог:

```
struct  direct {
   unsigned short d_ino;   /* 2 байта: номер I-узла */
   char    d_name[DIRSIZ]; /* имя файла             */
};
```

В семействе *BSD* формат каталога несколько иной - там записи имеют разную длину, зависящую от длины имени файла, которое может иметь длину от 1 до 256 символов.

Имя файла может состоять из **любых** символов, кроме '\0', служащего признаком конца имени и '/', служащего разделителем. В имени допустимы пробелы, управляющие символы (но не рекомендуются!), **любое** число точек (в отличие от *MS DOS*, где допустима **единственная** точка, отделяющая собственно имя от суффикса (расширения)), разрешены даже непечатные (т.е. управляющие) символы! Если имя файла имеет длину 14 (*DIRSIZ*) символов, то оно **не оканчивается** байтом '\0'. В этом случае для печати имени файла возможны три подхода:

1. Выводить символы при помощи *putchar*()-а в цикле. Цикл прерывать по индексу равному *DIRSIZ*, либо по достижению байта '*\0*'.
2. Скопировать поле **d_name** в другое место:

   ```
   char buf[ DIRSIZ + 1 ];
   strncpy(buf, d.d_name, DIRSIZ);
   buf[ DIRSIZ ] = '\0';
   ```

   Этот способ лучший, если имя файла надо не просто напечатать, но и запомнить на будущее, чтобы использовать в своей программе.

3. Использовать такую особенность функции *printf*():

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

   struct direct d;
      ...
   printf( "%*.*s\n", DIRSIZ, DIRSIZ, d.d_name );
   ```

Если файл был стерт, то в поле **d_ino** записи каталога будет содержаться 0 (именно поэтому I-узлы нумеруются начиная с 1, а не с 0). При удалении файла содержимое его (блоки) уничтожается, I-узел освобождается, но имя в каталоге не затирается физически, а просто помечается как стертое: **d_ino**=0; Каталог при этом никак не уплотняется и не укорачивается! Поэтому имена с **d_ino**==0 выдавать не следует - это имена уже уничтоженных файлов.

При создании нового имени (*creat*, *link*, *mknod*) система просматривает каталог и переиспользует первый от начала свободный слот (ячейку каталога) где **d_ino**==0, записывая новое имя в него (только в этот момент

старое имя-призрак окончательно исчезнет физически). Если пустых мест нет - каталог удлиняется.

Любой каталог **всегда** содержит два стандартных имени: "*.*" - ссылка на этот же каталог (на его собственный I-node), "*..*" - на вышележащий каталог. У корневого каталога "*/*" оба этих имени ссылаются на него же самого (т.е. содержат **d_ino**==2).

Имя каталога не содержится в нем самом. Оно содержится в "родительском" каталоге ...

Каталог в *UNIX* - это обычный дисковый файл. Вы можете **читать** его из своих программ. Однако **никто** (включая суперпользователя*) не может **записывать** что-либо в каталог при помощи *write*. Изменения содержимого каталогов выполняет **только ядро**, отвечая на запросы в виде системных вызовов *creat*, *unlink*, *link*, *mkdir*, *rmdir*, *rename*, *mknod*. Коды доступа для каталога интерпретируются следующим образом:

w **запись**
: *S_IWRITE*. Означает право создавать и уничтожать в каталоге имена файлов при помощи этих вызовов. То есть: право создавать, удалять и переименовывать файлы в каталоге. Отметим, что для переименования или удаления файла вам не требуется иметь доступ по записи к самому файлу - достаточно иметь доступ по записи к каталогу, содержащему его имя!

r **чтение**
: *S_IREAD*. Право читать каталог как обычный файл (право выполнять *opendir*, см. ниже): благодаря этому мы можем получить список имен файлов, содержащихся в каталоге. Однако, если мы ЗАРАНЕЕ знаем имена файлов в каталоге, мы МОЖЕМ работать с ними - если имеем право доступа "**выполнение**" для этого каталога!

x **выполнение**
: *S_IEXEC*. Разрешает **поиск** в каталоге. Для открытия файла, создания/удаления файла, перехода в другой каталог (*chdir*), система выполняет следующие действия (осуществляемые функцией *namei*() в ядре): чтение каталога и поиск в нем указанного имени файла или каталога; найденному имени соответствует номер I-узла **d_ino**; по номеру узла система считывает с диска сам I-узел нужного файла и по нему добирается до содержимого файла. Код "**выполнение**" - это как раз разрешение такого просмотра каталога *системой*. Если каталог имеет доступ на чтение - мы можем получить список файлов (т.е. применить команду *ls*); но если он при этом не имеет кода доступа "выполнение" - мы не сможем получить доступа ни к одному из файлов каталога (ни открыть, ни удалить, ни создать, ни сделать *stat*, ни *chdir*). Т.е. "**чтение**" разрешает применение вызова *read*, а "**выполнение**" - функции ядра *namei*. Фактически "**выполнение**" означает "доступ к файлам в данном каталоге"; еще более точно - к I-nodam файлов этого каталога.

t **sticky bit**
: *S_ISVTX* - для каталога он означает, что удалить или переименовать некий файл в данном каталоге могут только: владелец каталога, владелец данного файла, суперпользователь. И никто другой. Это исключает удаление файлов чужими.

Совет: для каталога полезно иметь такие коды доступа:

```
chmod o-w,+t каталог
```

В системах *BSD* используется, как уже было упомянуто, формат каталога с переменной длиной записей. Чтобы иметь удобный доступ к именам в каталоге, возникли специальные функции чтения каталога: *opendir*, *closedir*, *readdir*. Покажем, как простейшая команда *ls* реализуется через эти функции.

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int listdir(char *dirname){
    register struct dirent *dirbuf;
    DIR *fddir;
    ino_t dot_ino = 0, dotdot_ino = 0;

    if((fddir = opendir (dirname)) == NULL){
        fprintf(stderr, "Can't read %s\n", dirname);
        return 1;
    }
    /* Без сортировки по алфавиту */
    while ((dirbuf = readdir (fddir)) != NULL ) {
        if (dirbuf->d_ino == 0) continue;
        if (strcmp (dirbuf->d_name, "." ) == 0){
                dot_ino = dirbuf->d_ino;
                continue;
        } else if(strcmp (dirbuf->d_name, "..") == 0){
                dotdot_ino = dirbuf->d_ino;
                continue;
```

```
            } else printf("%s\n", dirbuf->d_name);
        }
        closedir (fddir);

        if(dot_ino    == 0) printf("Поврежденный каталог: нет имени \".\"\n");
        if(dotdot_ino == 0) printf("Поврежденный каталог: нет имени \"..\"\n");
        if(dot_ino && dot_ino == dotdot_ino)  printf("Это корневой каталог диска\n");

        return 0;
    }

    int main(int ac, char *av[]){
        int i;

        if(ac > 1) for(i=1; i < ac; i++) listdir(av[i]);
        else                             listdir(".");

        return 0;
    }
```

Обратите внимание, что тут не требуется добавление '\0' в конец поля *d_name*, поскольку его предоставляет нам сама функция *readdir*().

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

```
    rmdir(имя_каталога);
```

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

```
    unlink(имя_файла);
```

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

* - Именно это имя показывает команда *ps -ef*

* - Собственно, операционная система характеризуется набором предоставляемых ею системных вызовов, поскольку все концепции, заложенные в системе, доступны нам **только** через них. Если мы имеем две реализации системы с **разным** внутренним устройством ядер, но предоставляющие **одинаковый интерфейс** системных вызовов (их набор, смысл и поведение), то это все-таки *одна и та же* система! Ядра могут не просто отличаться, но и быть построенными на совершенно различных принципах: так обстоит дело с *UNIX*-ами на однопроцессорных и многопроцессорных машинах. Но для нас ядро - это "черный ящик", полностью определяемый его **поведением**, т.е. своим **интерфейсом** с программами, но не внутренним устройством. Вторым параметром, характеризующим ОС, являются **форматы данных**, используемые системой: форматы данных для сисвызовов и формат информации в различных файлах, в том числе формат оформления выполняемых файлов (формат данных **в физической памяти** машины в этот список *не* входит - он зависим от реализации и от процессора). Как правило, программа пишется так, чтобы использовать соглашения, принятые в данной системе, для чего она просто включает ряд стандартных include-файлов с описанием этих форматов. **Имена** этих файлов также можно отнести к интерфейсу системы.

* - Таким как таблица процессов, таблица открытых файлов (всех вместе и для каждого процесса), и.т.п.

* - Суперпользователь (**superuser**) имеет **uid**==0. Это "привелегированный" пользователь, который имеет право делать ВСЕ. Ему доступны любые сисвызовы и файлы, несмотря на коды доступа и.т.п.

Назад | Содержание | Вперед

[Главная] [Гостевая]