6.5.8. Как уже было сказано, при *exec* все открытые файлы достаются в наследство новой программе (в частности, если между *fork* и *exec* были перенаправлены вызовом *dup2* стандартные ввод и вывод, то они останутся перенаправленными и у новой программы). Что делать, если мы не хотим, чтобы наследовались **все** открытые файлы? (Хотя бы потому, что большинством из них новая программа пользоваться не будет - в основном она будет использовать лишь **fd** 0, 1 и 2; а ячейки в таблице открытых файлов процесса они занимают). Во-первых, ненужные дескрипторы можно явно закрыть *close* в промежутке между *fork*-ом и *exec*-ом. Однако не всегда мы помним номера дескрипторов для этой операции. Более радикальной мерой является тотальная чистка:

```
for(f = 3; f < NOFILE; f++)
        close(f);
```

Есть более элегантный путь. Можно пометить дескриптор файла специальным флагом, означающим, что во время вызова *exec* этот дескриптор должен быть **автоматически** закрыт (режим file-close-on-exec - *fclex*):

```
#include <fcntl.h>
int fd = open(.....);
fcntl (fd, F_SETFD, 1);
```

Отменить этот режим можно так:

```
fcntl (fd, F_SETFD, 0);
```

Здесь есть одна тонкость: этот флаг устанавливается не для структуры *file* - "открытый файл", а непосредственно для **дескриптора** в таблице открытых процессом файлов (массив флагов: char **u_pofile**[NOFILE]). Он **не** сбрасывается при закрытии файла, поэтому нас может ожидать сюрприз:

```
 ... fcntl (fd, F_SETFD, 1); ... close(fd);
 ...
 int fd1 = open( ... );
```

Если **fd1** окажется равным **fd**, то дескриптор **fd1** будет при *exec*-е закрыт, чего мы явно не ожидали! Поэтому перед *close*(**fd**) полезно было бы отменить режим *fclex*.

6.5.9. Каждый процесс имеет **управляющий терминал** (short ***u_ttyp**). Он достается процессу в наследство от родителя (при *fork* и *exec*) и обычно совпадает с терминалом, с на котором работает данный пользователь.

Каждый процесс относится к некоторой **группе процессов** (int **p_pgrp**), которая также наследуется. Можно послать сигнал всем процессам указанной группы **pgrp**:

```
kill( -pgrp, sig );
```

Вызов

```
kill( 0, sig );
```

посылает сигнал **sig** всем процессам, чья группа совпадает с группой посылающего процесса. Процесс может узнать свою группу:

```
int pgrp = getpgrp();
```

а может стать "лидером" новой группы. Вызов

```
setpgrp();
```

делает следующие операции:

```
/* У процесса больше нет управл. терминала: */
if(p_pgrp != p_pid) u_ttyp = NULL;
/* Группа процесса полагается равной его ид-у: */
p_pgrp = p_pid;  /* new group */
```

В свою очередь, управляющий терминал тоже имеет некоторую группу (**t_pgrp**). Это значение устанавливается равным группе процесса, первым открывшего этот терминал:

```
/* часть процедуры открытия терминала */
if( p_pid == p_pgrp // лидер группы
 && u_ttyp == NULL  // еще нет упр.терм.
 && t_pgrp == 0 ){  // у терминала нет группы
        u_ttyp = &t_pgrp;
        t_pgrp =  p_pgrp;
}
```

Таким процессом обычно является процесс регистрации пользователя в системе (который спрашивает у вас имя и пароль). При закрытии терминала всеми процессами (что бывает при выходе пользователя из системы) терминал теряет группу: t_pgrp=0;

При нажатии на клавиатуре терминала некоторых клавиш:

```
c_cc[ VINTR ]     обычно DEL или CTRL/C
c_cc[ VQUIT ]     обычно CTRL/\
```

драйвер терминала посылает соответственно сигналы *SIGINT* и *SIGQUIT* всем процессам группы терминала, т.е. как бы делает

```
kill( -t_pgrp, sig );
```

Именно поэтому мы можем прервать процесс нажатием клавиши *DEL*. Поэтому, если процесс сделал *setpgrp*(), то сигнал с клавиатуры ему послать невозможно (т.к. он имеет свой уникальный номер группы != группе терминала).

Если процесс еще не имеет управляющего терминала (или уже его не имеет после *setpgrp*), то он может сделать **любой** терминал (который он имеет право открыть) управляющим для себя. Первый же файл-устройство, являющийся интерфейсом драйвера терминалов, который будет открыт этим процессом, станет для него управляющим терминалом. Так процесс может иметь каналы 0, 1, 2 связанные с одним терминалом, а прерывания получать с клавиатуры другого (который он сделал управляющим для себя).

Процесс регистрации пользователя в системе - */etc/getty* (название происходит от "**get tty**" - получить терминал) - запускается процессом номер 1 - */etc/init*-ом - на каждом из терминалов, зарегистрированных в системе, когда

- система только что была запущена;
- либо когда пользователь на каком-то терминале вышел из системы (интерпретатор команд завершился).

В сильном упрощении *getty* может быть описан так:

```
void main(ac, av) char *av[];
{   int f; struct termio tmodes;

    for(f=0; f < NOFILE; f++) close(f);

    /* Отказ от управляющего терминала,
     * основание новой группы процессов.
     */
    setpgrp();

    /* Первоначальное явное открытие терминала */
    /* При этом терминал av[1] станет упр. терминалом */
        open( av[1], O_RDONLY ); /* fd = 0 */
        open( av[1], O_RDWR   ); /* fd = 1 */
     f = open( av[1], O_RDWR   ); /* fd = 2 */

    // ... Считывание параметров терминала из файла
    // /etc/gettydefs. Тип требуемых параметров линии
    // задается меткой, указываемой в av[2].
    // Заполнение структуры tmodes требуемыми
    // значениями ... и установка мод терминала.
    ioctl (f, TCSETA, &tmodes);

    // ... запрос имени и пароля ...

    chdir (домашний_каталог_пользователя);

    execl ("/bin/csh", "-csh", NULL);
    /* Запуск интерпретатора команд. Группа процессов,
     * управл. терминал, дескрипторы 0,1,2 наследуются.
     */
}
```

Здесь последовательные вызовы *open* занимают последовательные ячейки в таблице открытых процессом файлов (поиск каждой новой незанятой ячейки производится с начала таблицы) – в итоге по дескрипторам 0,1,2 открывается файл-терминал. После этого дескрипторы 0,1,2 наследуются всеми потомками интерпретатора команд. Процесс *init* запускает по одному процессу *getty* на каждый терминал, как бы делая

```
/etc/getty /dev/tty01 m &
/etc/getty /dev/tty02 m &
        ...
```

и ожидает окончания любого из них. После входа пользователя в систему на каком-то терминале, соответствующий *getty* превращается в интерпретатор команд (**pid** процесса сохраняется). Как только кто-то из них умрет – *init* перезапустит *getty* на соответствующем терминале (все они – его сыновья, поэтому он знает – на каком именно терминале).

6.6. Трубы и FIFO-файлы.

Процессы могут обмениваться между собой информацией через файлы. Существуют файлы с необычным поведением – так называемые *FIFO*-файлы (**first in, first out**), ведущие себя подобно очереди. У них указатели чтения и записи **разделены.** Работа с таким файлом напоминает проталкивание шаров через трубу – с одного конца мы вталкиваем данные, с другого конца – вынимаем их. Операция чтения из **пустой** "трубы" проиостановит вызов *read* (и издавший его процесс) до тех пор, пока кто-нибудь не запишет в FIFOфайл какие-нибудь данные. Операция позиционирования указателя – *lseek*() – **неприме- нима** к FIFO-файлам. FIFO-файл создается системным вызовом

```
#include <sys/types.h>
#include <sys/stat.h>
   mknod( имяФайла, S_IFIFO | 0666, 0 );
```

где 0666 – коды доступа к файлу. При помощи FIFO-файла могут общаться даже неродственные процессы.

Разновидностью FIFO-файла является **безымянный** FIFO-файл, предназначенный для обмена информацией между процессом-отцом и процессом-сыном. Такой файл – канал связи как раз и называется термином "труба" или *pipe*. Он создается вызовом *pipe*:

```
int conn[2];   pipe(conn);
```

Если бы файл-труба имел имя **PIPEFILE**, то вызов *pipe* можно было бы описать как

```
mknod("PIPEFILE", S_IFIFO | 0600, 0);
conn[0] = open("PIPEFILE", O_RDONLY);
conn[1] = open("PIPEFILE", O_WRONLY);
unlink("PIPEFILE");
```

При вызове *fork* каждому из двух процессов достанется в наследство пара дескрипторов:

```
            pipe(conn);
              fork();

conn[0]----<----    ----<-----conn[1]
                FIFO
conn[1]---->----    ---->-----conn[0]
 процесс A               процесс B
```

Пусть процесс *A* будет посылать информацию в процесс *B*. Тогда процесс *A* сделает:

```
close(conn[0]);
// т.к. не собирается ничего читать
write(conn[1], ... );
```

а процесс *B*

```
close(conn[1]);
// т.к. не собирается ничего писать
read (conn[0], ... );
```

Получаем в итоге:

```
conn[1]---->----FIFO---->-----conn[0]
 процесс A               процесс B
```

Обычно поступают еще более элегантно, перенаправляя стандартный вывод *A* в канал **conn**[1]

```
    dup2 (conn[1], 1); close(conn[1]);
    write(1, ... );   /* или printf */
```

а стандартный ввод *B* – из канала **conn**[0]

```
    dup2(conn[0], 0); close(conn[0]);
    read(0, ... );    /* или gets */
```

Это соответствует конструкции

```
        $   A | B
```

записанной на языке СиШелл.

Файл, выделяемый под *pipe*, имеет ограниченный размер (и поэтому обычно целиком оседает в буферах в памяти машины). Как только он заполнен целиком – процесс, пишущий в трубу вызовом *write*, приостанавливается до появления свободного места в трубе. Это может привести к возникновению тупиковой ситуации, если писать программу неаккуратно. Пусть процесс *A* является сыном процесса *B*, и пусть процесс *B* издает вызов *wait*, не закрыв канал **conn**[0]. Процесс же *A* очень много пишет в трубу **conn**[1]. Мы получаем ситуацию, когда оба процесса спят:

A потому что труба переполнена, а процесс *B* ничего из нее не читает, так как ждет окончания *A*;

B потому что процесс-сын *A* не окончился, а он не может окончиться пока не допишет свое сообщение.

Решением служит запрет процессу *B* делать вызов *wait* до тех пор, пока он не прочитает ВСЮ информацию из трубы (не получит EOF). Только сделав после этого *close*(**conn**[0]); процесс *B* имеет право сделать *wait*.

Если процесс *B* закроет свою сторону трубы *close*(**conn**[0]) **прежде**, чем процесс *A* закончит запись в нее, то при вызове *write* в процессе *A*, система пришлет процессу *A* сигнал *SIGPIPE* – "запись в канал, из которого никто не читает".

6.6.1. Открытие *FIFO* файла приведет к блокированию процесса ("засыпанию"), если в буфере *FIFO* файла пусто. Процесс заснет внутри вызова *open* до тех пор, пока в буфере что-нибудь не появится.

Чтобы избежать такой ситуации, а, например, сделать что-нибудь иное полезное в это время, нам надо было бы **опросить** файл на предмет того – можно ли его открыть? Это делается при помощи флага *O_NDELAY* у вызова *open*.

```
    int fd = open(filename, O_RDONLY|O_NDELAY);
```

Если *open* ведет к блокировке процесса внутри вызова, вместо этого будет возвращено значение (-1). Если же файл может быть немедленно открыт – возвращается нормальный дескриптор со значением >=0, и файл открыт.

O_NDELAY является зависимым от семантики того файла, который мы открываем. К примеру, можно использовать его с файлами устройств, например именами, ведущими к последовательным портам. Эти файлы устройств (порты) обладают тем свойством, что одновременно их может открыть только один процесс (так устроена реализация функции *open* внутри драйвера этих устройств). Поэтому, если один процесс уже работает с портом, а в это время второй пытается его же открыть, второй "заснет" внутри *open*, и будет дожидаться освобождения порта *close* первым процессом. Чтобы не ждать – следует открывать порт с флагом *O_NDELAY*.

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

    /* Убрать больше не нужный O_NDELAY */
    void nondelay(int fd){
            fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) & ~O_NDELAY);
    }
    int main(int ac, char *av[]){
            int fd;
            char *port = ac > 1 ? "/dev/term/a" : "/dev/cua/a";

    retry:  if((fd = open(port, O_RDWR|O_NDELAY)) < 0){
                    perror(port);
                    sleep(10);
                    goto retry;
            }
            printf("Порт %s открыт.\n", port);
            nondelay(fd);

            printf("Работа с портом, вызови эту программу еще раз!\n");
            sleep(60);
            printf("Все.\n");
```

```
        return 0;
}
```

Вот протокол:

```
su# a.out & a.out xxx
[1] 22202
Порт /dev/term/a открыт.
Работа с портом, вызови эту программу еще раз!
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
/dev/cua/a: Device busy
Все.
Порт /dev/cua/a открыт.
Работа с портом, вызови эту программу еще раз!
su#
```

6.7. Нелокальный переход.

Теперь поговорим про **нелокальный переход.** Стандартная функция *setjmp* позволяет установить в программе "контрольную точку"*, а функция *longjmp* осуществляет прыжок в эту точку, выполняя за один раз выход **сразу из нескольких** вызванных функций (если надо)*. Эти функции не являются системными вызовами, но поскольку они реализуются машинно-зависимым образом, а используются чаще всего как реакция на некоторый сигнал, речь о них идет в этом разделе. Вот как, например, выглядит рестарт программы по прерыванию с клавиатуры:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf jmp;  /* контрольная точка */

/* прыгнуть в контрольную точку */
void onintr(nsig){ longjmp(jmp, nsig); }

main(){
   int n;
   n = setjmp(jmp);  /* установить контрольную точку */
   if( n ) printf( "Рестарт после сигнала %d\n", n);
   signal (SIGINT, onintr);     /* реакция на сигнал */
   printf("Начали\n");
   ...
}
```

setjmp возвращает 0 при **запоминании** контрольной точки. При прыжке в контрольную точку при помощи *longjmp*, мы оказываемся снова в функции *setjmp*, и эта функция возвращает нам значение второго аргумента *longjmp*, в этом примере - **nsig.**

Прыжок в контрольную точку очень удобно использовать в алгоритмах перебора с возвратом (**backtracking**): либо - если ответ найден - прыжок на печать ответа, либо если ветвь перебора зашла в тупик - прыжок в точку ветвления и выбор другой альтернативы. При этом можно делать прыжки и в рекурсивных вызовах одной и той же функции: с более высокого уровня рекурсии в вызов более низкого уровня (в этом случае *jmp_buf* лучше делать автоматической переменной - своей для каждого уровня вызова функции).

6.7.1. Перепишите следующий алгоритм при помощи *longjmp*.

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

```
  if( i * j == 21 ){
      printf( "  Годятся (%d,%d)\n", i,j);
      value = j; return FOUND;
  }
  return NOTFOUND;
}
```

Вот ответ, использующий нелокальный переход вместо цепочки *return*-ов:

```
#include <setjmp.h>
jmp_buf jmp;
main(){   int i;
  if( i = setjmp(jmp))  /* после прыжка */
        printf("Ответ %d\n", --i);
  else  /* установка точки */
    for(i=2; i < 10; i++)
      printf( "пробуем i=%d\n", i), test1(i);
}
test1(i){ int j;
  for(j=1; j < 10 ; j++ )
      printf( "пробуем j=%d\n", j), test2(i,j);
}
test2(i, j){
  printf( "пробуем(%d,%d)\n", i, j);
  if( i * j == 21 ){
     printf( "  Годятся (%d,%d)\n", i,j);
     longjmp(jmp, j + 1);
  }
}
```

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

6.7.2. В чем ошибка?

```
#include <setjmp.h>

jmp_buf jmp;
main(){
     g();
     longjmp(jmp,1);
}
g(){ printf("Вызвана g\n");
     f();
     printf("Выхожу из g\n");
}
f(){
     static n;
     printf( "Вызвана f\n");
     setjmp(jmp);
     printf( "Выхожу из f %d-ый раз\n", ++n);
}
```

Ответ: *longjmp* делает прыжок в функцию *f*(), из которой уже произошел возврат управления. При переходе в тело функции в обход ее заголовка не выполняются машинные команды "пролога" функции – функция остается "неактивированной". При возврате из вызванной таким "нелегальным" путем функции возникает ошибка, и программа падает. Мораль: в функцию, которая НИКЕМ НЕ ВЫЗВАНА, нельзя передавать управление. Обратный прыжок из *f*() в *main*() – был бы законен, поскольку функция *main*() является активной, когда управление находится в теле функции *f*(). Т.е. можно "прыгать" из вызванной функции в вызывающую: из *f*() в *main*() или в *g*(); и из *g*() в *main*();

```
--        -|   f    |  стек      прыгать
 |   g    |  вызовов   сверху вниз
 |   main |  функций   можно - это соответствует
 ----------            выкидыванию нескольких
                       верхних слоев стека
```

но нельзя наоборот: из *main*() в *g*() или *f*(); а также из *g*() в *f*(). Можно также совершать прыжок в пределах одной и той же функции:

```
f(){ ...
        A:   setjmp(jmp);
             ...
             longjmp(jmp, ...); ...
             /* это как бы goto A; */
}
```

6.8. Хозяин файла, процесса, и проверка привелегий.

UNIX - многопользовательская система. Это значит, что одновременно на разных терминалах, подключенных к машине, могут работать **разные** пользователи (а может и один на нескольких терминалах). На каждом терминале работает **свой** интерпретатор команд, являющийся потомком процесса */etc/init*.

6.8.1. Теперь - про функции, позволяющие узнать некоторые данные про любого пользователя системы. Каждый пользователь в *UNIX* имеет уникальный **номер:** идентификатор пользователя (**user id**), а также уникальное **имя:** регистрационное имя, которое он набирает для входа в систему. Вся информация о пользователях хранится в файле */etc/passwd*. Существуют функции, позволяющие по номеру пользователя узнать регистрационное имя и наоборот, а заодно получить еще некоторую информацию из *passwd*:

```
#include <stdio.h>
#include <pwd.h>
struct passwd *p;
int   uid;   /* номер */
char *uname; /* рег. имя */

uid = getuid();
p   = getpwuid( uid   );
        ...
p   = getpwnam( uname );
```

Эти функции возвращают указатели на статические структуры, скрытые внутри этих функций. Структуры эти имеют поля:

```
p->pw_uid     идентиф. пользователя (int uid);
p->pw_gid     идентиф. группы пользователя;

        и ряд полей типа char[]
p->pw_name    регистрационное имя пользователя (uname);
p->pw_dir     полное имя домашнего каталога
  (каталога, становящегося текущим при входе в систему);
p->pw_shell   интерпретатор команд
  (если "", то имеется в виду /bin/sh);
p->pw_comment произвольная учетная информация (не используется);
p->pw_gecos   произвольная учетная информация (обычно ФИО);
p->pw_passwd  зашифрованный пароль для входа в
   систему. Истинный пароль нигде не хранится вовсе!
```

Функции возвращают значение **p**==*NULL*, если указанный пользователь не существует (например, если задан неверный **uid**). **uid** хозяина данного процесса можно узнать вызовом *getuid*, а **uid** владельца файла - из поля **st_uid** структуры, заполняемой системным вызовом *stat* (а идентификатор группы владельца - из поля **st_gid**). Задание: модифицируйте наш аналог программы *ls*, чтобы он выдавал в текстовом виде имя владельца каждого файла в каталоге.

6.8.2. Владелец файла может изменить своему файлу идентификаторы владельца и группы вызовом

```
chown(char *имяФайла, int uid, int gid);
```

т.е. "подарить" файл другому пользователю. Забрать чужой файл себе невозможно. При этой операции биты *S_ISUID* и *S_ISGID* в кодах доступа к файлу (см. ниже) сбрасываются, поэтому создать "Троянского коня" и, сделав его хозяином суперпользователя, получить неограниченные привелегии - не удастся!

6.8.3. Каждый файл имеет своего владельца (поле **di_uid** в I-узле на диске или поле **i_uid** в копии I-узла в памяти ядра *А. Богатырев, 1992-95 Си в UNIX*[*]). Каждый процесс также имеет своего владельца (поля **u_uid** и **u_ruid** в *u-area*). Как мы видим, процесс имеет **два** параметра, обозначающие владельца. Поле **ruid** называется "*реальным идентификатором*" пользователя, а **uid** "*эффективным идентификатором*". При вызове *exec*() заменяется программа, выполняемая данным процессом:

```
 старая программа  exec    новая программа
     ruid -->-----------------><---> ruid
     uid  -->--------*-------->---> uid (new)
                     |
                выполняемый файл
                 i_uid (st_uid)
```

Как видно из этой схемы, реальный идентификатор хозяина процесса наследуется. Эффективный идентификатор обычно также наследуется, за исключением одного случая: если в кодах доступа файла (**i_mode**) выставлен бит *S_ISUID* (set-uid bit), то значение поля **u_uid** в новом процессе станет равно значению **i_uid** файла с программой:

```
/* ... во время exec ... */
p_suid = u_uid;     /* спасти */
if( i_mode & S_ISUID ) u_uid = i_uid;
if( i_mode & S_ISGID ) u_gid = i_gid;
```

т.е. эффективным владельцем процесса станет владелец файла. Здесь **gid** - это идентификаторы **группы владельца** (которые тоже есть и у файла и у процесса, причем у процесса - реальный и эффективный).

Зачем все это надо? Во-первых затем, что ПРАВА процесса на доступ к какому-либо файлу проверяются именно для **эффективного** владельца процесса. Т.е. например, если файл имеет коды доступа

```
mode = i_mode & 0777;
             /* rwx rwx rwx */
```

и владельца **i_uid**, то процесс, пытающийся открыть этот файл, будет "проэкзаменован" в таком порядке:

```
if( u_uid == 0 )  /* super user */
     то доступ разрешен;
else if( u_uid == i_uid )
     проверить коды (mode & 0700);
else if( u_gid == i_gid )
     проверить коды (mode & 0070);
else проверить коды (mode & 0007);
```

Процесс может узнать свои параметры:

```
unsigned short uid  = geteuid();  /* u_uid  */
unsigned short ruid = getuid();   /* u_ruid */
unsigned short gid  = getegid();  /* u_gid  */
unsigned short rgid = getuid();   /* u_rgid */
```

а также установить их:

```
setuid(newuid);  setgid(newgid);
```

Рассмотрим вызов *setuid*. Он работает так (**u_uid** - относится к процессу, издавшему этот вызов):

```
if(      u_uid == 0 /* superuser */ )
         u_uid = u_ruid =    p_suid =  newuid;
else if( u_ruid == newuid || p_suid == newuid )
         u_uid = newuid;
else     неудача;
```

Поле **p_suid** позволяет set-uid-ной программе восстановить эффективного владельца, который был у нее до *exec*-а.

Во-вторых, все это надо для следующего случая: пусть у меня есть некоторый файл **BASE** с хранящимися в нем секретными сведениями. Я являюсь владельцем этого файла и устанавливаю ему коды доступа 0600 (чтение и запись разрешены **только** мне). Тем не менее, я хочу дать другим пользователям возможность работать с этим файлом, однако контролируя их деятельность. Для этого я пишу **программу**, которая выполняет некоторые действия с файлом **BASE**, при этом проверяя законность этих действий, т.е. позволяя делать не все что попало, а лишь то, что я в ней предусмотрел, и под жестким контролем. Владельцем файла **PROG**, в котором хранится эта программа, также являюсь я, и я задаю этому файлу коды доступа 0711 (rwx--x--x) - всем можно выполнять эту программу. Все ли я сделал, чтобы позволить другим пользоваться базой **BASE** через программу (и только нее) **PROG**? Нет!

Если кто-то другой запустит программу **PROG**, то эффективный идентификатор процесса будет равен идентификатору этого **другого** пользователя, и программа **не сможет** открыть мой файл **BASE**. Чтобы все работало, процесс, выполняющий программу **PROG**, должен работать как бы от моего имени. Для этого я должен вызовом *chmod* либо командой

```
chmod u+s PROG
```

добавить к кодам доступа файла **PROG** бит *S_ISUID*.

После этого, при запуске программы **PROG**, она будет получать эффективный идентификатор, равный **моему** идентификатору, и таким образом сможет открыть и работать с файлом **BASE**. Вызов *getuid* позволяет выяснить, кто вызвал мою программу (и занести это в протокол, если надо).

Программы такого типа - не редкость в *UNIX*, если владельцем программы (файла ее содержащего) является суперпользователь. В таком случае программа, имеющая бит доступа *S_ISUID* работает **от имени суперпользователя** и может выполнять некоторые действия, запрещенные обычным пользователям. При этом программа внутри себя делает всяческие проверки и периодически спрашивает пароли, то есть при работе

защищает систему от дураков и преднамеренных вредителей. Простейшим примером служит команда *ps*, которая считывает таблицу процессов из памяти ядра и распечатывает ее. Доступ к физической памяти машины производится через файл-псевдоустройство */dev/mem*, а к памяти ядра */dev/kmem*. Чтение и запись в них позволены **только** суперпользователю, поэтому программы "общего пользования", обращающиеся к этим файлам, должны иметь бит set-uid.

Откуда же изначально берутся значения **uid** и **ruid** (а также **gid** и **rgid**) у процесса? Они берутся из процесса регистрации пользователя в системе: */etc/getty*. Этот процесс запускается на каждом терминале как процесс, принадлежащий суперпользователю (**u_uid**==0). Сначала он запрашивает имя и пароль пользователя:

```
#include <stdio.h>  /* cc -lc_s */
#include <pwd.h>
#include <signal.h>
struct passwd *p;
char userName[80], *pass, *crpass;
extern char *getpass(), *crypt();
  ...
/* Не прерываться по сигналам с клавиатуры */
signal (SIGINT, SIG_IGN);
for(;;){
  /* Запросить имя пользователя: */
  printf("Login: "); gets(userName);
  /* Запросить пароль (без эха): */
  pass = getpass("Password: ");
  /* Проверить имя: */
  if(p = getpwnam(userName)){
     /* есть такой пользователь */
     crpass = (p->pw_passwd[0]) ? /* если есть пароль */
              crypt(pass, p->pw_passwd) : pass;
     if( !strcmp( crpass, p->pw_passwd))
              break; /* верный пароль */
  }
  printf("Login incorrect.\a\n");
}
signal (SIGINT, SIG_DFL);
```

Затем он выполняет:

```
// ... запись информации о входе пользователя в систему
// в файлы /etc/utmp (кто работает в системе сейчас)
// и       /etc/wtmp (список всех входов в систему)
        ...
setuid( p->pw_uid ); setgid( p->pw_gid );
chdir ( p->pw_dir ); /* GO HOME! */
// эти параметры будут унаследованы
// интерпретатором команд.
        ...
// настройка некоторых переменных окружения envp:
// HOME     = p->pw_dir
// SHELL    = p->pw_shell
// PATH     = нечто по умолчанию, вроде :/bin:/usr/bin
// LOGNAME (USER) = p->pw_name
// TERM     = считывается из файла
//            /etc/ttytype по имени устройства av[1]
// Делается это как-то подобно
//   char *envp[MAXENV], buffer[512]; int envc = 0;
//   ...
//   sprintf(buffer, "HOME=%s", p->pw_dir);
//   envp[envc++] = strdup(buffer);
//   ...
//   envp[envc] = NULL;
        ...
// настройка кодов доступа к терминалу. Имя устройства
// содержится в параметре av[1] функции main.
chown (av[1], p->pw_uid, p->pw_gid);
chmod (av[1], 0600 );  /* -rw------- */
// теперь доступ к данному терминалу имеют только
// вошедший в систему пользователь и суперпользователь.
// В случае смерти интерпретатора команд,
// которым заменится getty, процесс init сойдет
// с системного вызова ожидания wait() и выполнит
//  chown ( этот_терминал, 2 /*bin*/, 15 /*terminal*/ );
//  chmod ( этот_терминал, 0600 );
// и, если терминал числится в файле описания линий
// связи /etc/inittab как активный (метка respawn), то
// init перезапустит на этом_терминале новый
// процесс getty при помощи пары вызовов fork() и exec().
        ...
// запуск интерпретатора команд:
execle( *p->pw_shell ? p->pw_shell : "/bin/sh",
                  "-", NULL, envp );
```

В результате он становится процессом пользователя, вошедшего в систему. Таковым же после *exec*-а, выполняемого *getty*, остается и интерпретатор команд **p->pw_shell** (обычно */bin/sh* или */bin/csh*) и все его потомки.

На самом деле, в описании регистрации пользователя при входе в систему, сознательно было допущено упрощение. Дело в том, что все то, что мы приписали процессу *getty*, в действительности выполняется **двумя** программами: */etc/getty* и */bin/login*.

Сначала процесс *getty* занимается настройкой параметров линии связи (т.е. терминала) в соответствии с ее описанием в файле */etc/gettydefs*. Затем он запрашивает **имя** пользователя и заменяет себя (при помощи сисвызова *exec*) процессом *login*, передавая ему в качестве одного из аргументов полученное имя пользователя.

Затем *login* запрашивает пароль, настраивает окружение, и.т.п., то есть именно он производит все операции, приведенные выше на схеме. В конце концов он заменяет себя интерпретатором команд.

Такое разделение делается, в частности, для того, чтобы считанный пароль в случае опечатки не хранился бы в памяти процесса *getty*, а уничтожался бы при очистке памяти завершившегося процесса *login*. Таким образом пароль в истинном, незашифрованном виде хранится в системе минимальное время, что затрудняет его подсматривание средствами электронного или программного шпионажа. Кроме того, это позволяет изменять систему проверки паролей не изменяя программу инициализации терминала *getty*.

Имя, под которым пользователь вошел в систему на данном терминале, можно узнать вызовом стандартной функции

```
char *getlogin();
```

Эта функция не проверяет **uid** процесса, а просто извлекает запись про данный терминал из файла */etc/utmp*.

Наконец отметим, что владелец **файла** устанавливается при создании этого файла (вызовами *creat* или *mknod*), и полагается равным эффективному идентификатору создающего процесса.

```
di_uid = u_uid;     di_gid = u_gid;
```

6.8.4. Напишите программу, узнающую у системы и распечатывающую: номер процесса, номер и имя своего владельца, номер группы, название и тип терминала на котором она работает (из переменной окружения *TERM*).

* – В некотором буфере запоминается текущее состояние процесса: положение вершины стека вызовов функций (**stack pointer**); состояние всех регистров процессора, включая регистр адреса текущей машинной команды (**instruction pointer**).

* – Это достигается восстановлением состояния процесса из буфера. Изменения, происшедшие за время между *setjmp* и *longjmp* в статических данных не отменяются (т.к. они не сохранялись).

* – При открытии файла и вообще при любой операции с файлом, в таблицах ядра заводится **копия** I-узла (для ускорения доступа, чтобы постоянно не обращаться к диску). Если I-узел в памяти будет изменен, то при закрытии файла (а также периодически через некоторые промежутки времени) эта копия будет записана обратно на диск. Структура I-узла в памяти – struct *inode* – описана в файле <**sys/inode.h**>, а на диске – struct *dinode* – в файле <**sys/ino.h**>.