6.5. Жизнь процессов.

6.5.1. Какие классы памяти имеют данные, в каких сегментах программы они расположены?

```
char x[] = "hello";
int y[25];
char *p;
main(){
        int z = 12;
        int v;
        static int w = 25;
        static int q;
        char s[20];
        char *pp;
        ...
        v = w + z;      /* #1 */
}
```

Ответ:

Переменная	Класс памяти	Сегмент	Начальное значение
x	static	data/DATA	"hello"
y	static	data/BSS	{0, ..., 0}
p	static	data/BSS	NULL
z	auto	stack	12
v	auto	stack	не определено
w	static	data/DATA	25
q	static	data/BSS	0
s	auto	stack	не определено
pp	auto	stack	не определено
main	static	text/TEXT	

Большими буквами обозначены сегменты, хранимые в выполняемом файле:

DATA - это **инициализированные** статические данные (которым присвоены начальные значения). Они помещаются компилятором в файл в виде готовых констант, а при запуске программы (при ее загрузке в память машины), просто копируются в память из файла.

BSS (Block Started by Symbol) - неинициализированные статические данные. Они по умолчанию имеют начальное значение 0 (NULL, "", '\0'). Эта память расписывается нулями при запуске программы, а в файле хранится лишь ее **размер.**

TEXT - сегмент, содержащий машинные команды (код).

Хранящаяся в файле выполняемая программа имеет также **заголовок** - в нем в частности содержатся размеры перечисленных сегментов и их местоположение в файле; и еще - в самом конце файла - **таблицу имен.** В ней содержатся имена всех функций и переменных, используемых в программе, и их адреса. Эта таблица используется отладчиками *adb* и *sdb*, а также при сборке программы из нескольких объектных файлов программой *ld*. Просмотреть ее можно командой

nm **имяФайла**

Для экономии дискового пространства эту таблицу часто удаляют, что делается командой

strip **имяФайла**

Размеры сегментов можно узнать командой

size **имяФайла**

Программа, загруженная в память компьютера (т.е. процесс), состоит из 3х сегментов, относящихся непосредственно к программе:

stack - стек для локальных переменных функций (автоматических переменных). Этот сегмент существует только у выполняющейся программы, поскольку отведение памяти в стеке производится выполнением некоторых машинных команд (поэтому описание автоматических переменных в Си - это на самом деле **выполняемые** операторы, хотя и не с точки зрения языка). Сегмент стека **автоматически** растет по мере надобности (если мы вызываем новые и новые функции, отводящие переменные в стеке). За этим следит аппаратура диспетчера памяти.

data - сегмент, в который склеены сегменты статических данных *DATA* и *BSS*, загруженные из файла. Этот сегмент также может изменять свой размер, но делать это надо явно - системными вызовами *sbrk* или *brk*. В частности, функция *malloc*() для размещения динамически отводимых данных увеличивает размер этого сегмента.

text - это выполняемые команды, копия сегмента *TEXT* из файла. Так строка с меткой #1 содержится в виде машинных команд именно в этом сегменте.

Кроме того, каждый процесс имеет еще:

proc - это резидентная часть паспорта процесса в таблице процессов в ядре операционной системы;

user - это 4-ый сегмент процесса - нерезидентная часть паспорта (*u-area*). К этому сегменту имеет доступ только ядро, но не сама программа.

Паспорт процесса был поделен на 2 части только из соображений экономии памяти в ядре: контекст процесса (таблица открытых файлов, ссылка на I-узел текущего каталога, таблица реакций на сигналы, ссылка на I-узел управляющего терминала, и.т.п.) нужен ядру только при обслуживании текущего активного процесса. Когда активен другой процесс эта информация в памяти ядра не нужна. Более того, если процесс из-за нехватки места в памяти машины был откачан на диск, эта информация также может быть откачана на диск и подкачана назад лишь вместе с процессом. Поэтому контекст был выделен в отдельный сегмент, и сегмент этот подключается к адресному пространству ядра лишь при выполнении процессом какого-либо системного вызова (это подключение называется "переключение контекста" - **context switch**). Четыре сегмента процесса могут располагаться в памяти машины не обязательно подряд - между ними могут лежать сегменты других процессов.

Схема составных частей процесса:

```
          П  Р  О  Ц  Е  С  С
     таблица процессов:
     паспорт  в ядре       сегменты в памяти

      struct proc[]
          ####---------------> stack      1
          ####                 data       2
                               text       3
                  контекст:   struct user  4
```

Каждый процесс имеет уникальный номер, хранящийся в поле **p_pid** в структуре *proc*[*]. В ней также хранятся: адреса сегментов процесса в памяти машины (или на диске, если процесс откачан); **p_uid** - номер владельца процесса; **p_ppid** - номер процесса-родителя; **p_pri**, **p_nice** - приоритеты процесса; **p_pgrp** - группа процесса; **p_wchan** - ожидаемое процессом событие; **p_flag** и **p_stat** - состояние процесса; и многое другое. Структура *proc* определена в include-файле <**sys/proc.h**>, а структура *user* - в <**sys/user.h**>.

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

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

Вызов *fork* может завершиться неудачей если таблица процессов переполнена. Простейший способ сделать это:

```
main(){
      while(1)
        if( ! fork()) pause();
}
```

Одно гнездо таблицы процессов зарезервировано - его может использовать только суперпользователь (в целях жизнеспособности системы: хотя бы для того, чтобы запустить программу, убивающую все эти процессы-варвары).

Вызов *fork* создает копию всех 4х сегментов процесса и выделяет порожденному процессу новый паспорт и номер. Иногда сегмент *text* не копируется, а используется процессами совместно ("**разделяемый сегмент**") в целях экономии памяти. При копировании сегмента *user* контекст порождающего процесса **наследуется** порожденным процессом (см. ниже).

Проведите опыт, доказывающий что порожденный системным вызовом *fork*() процесс и породивший его – равноправны. Повторите несколько раз программу:

```
#include <stdio.h>
int pid, i, fd; char c;
main(){
   fd = creat( "TEST", 0644);
   if( !(pid = fork())){ /* сын: порожденный процесс */
           c = 'a';
           for(i=0; i < 5; i++){
             write(fd, &c, 1); c++; sleep(1);
           }
           printf("Сын %d окончен\n", getpid());
           exit(0);
   }
   /* else процесс-отец */
   c = 'A';
   for(i=0; i < 5; i++){
           write(fd, &c, 1); c++; sleep(1);
   }
   printf("Родитель %d процесса %d окончен\n",
           getpid(), pid );
}
```

В файле **TEST** мы будем от случая к случаю получать строки вида

```
aABbCcDdEe  или  AaBbcdCDEe
```

что говорит о том, что первым "проснуться" после *fork*() может **любой** из двух процессов. Если же опыт дает устойчиво строки, начинающиеся с одной и той же буквы – значит в данной реализации системы один из процессов все же запускается раньше. Но не стоит использовать этот эффект – при переносе на другую систему его может не быть!

Данный опыт основан на следующем свойстве системы *UNIX*: при системном вызове *fork*() порожденный процесс получает все открытые порождающим процессом файлы "в наследство" – это соответствует тому, что таблица открытых процессом файлов копируется в процесс-потомок. Именно так, в частности, передаются от отца к сыну стандартные каналы 0, 1, 2: порожденному процессу не нужно открывать стандартные ввод, вывод и вывод ошибок явно. Изначально же они открываются специальной программой при вашем входе в систему.

до вызова *fork*();

```
 таблица открытых
 файлов  процесса
    0   ## ---<--- клавиатура
    1   ## --->--- дисплей
    2   ## --->--- дисплей
    ... ##
    fd  ## --->--- файл TEST
    ... ##
```

после *fork*();

```
 ПРОЦЕСС-ПАПА                    ПРОЦЕСС-СЫН
 0   ## ---<--- клавиатура --->--- ## 0
 1   ## --->--- дисплей    ---<--- ## 1
 2   ## --->--- дисплей    ---<--- ## 2
 ... ##                             ## ...
 fd  ## --->--- файл TEST  ---<--- ## fd
 ... ##            |                ## ...
                   *--RWptr-->ФАЙЛ
```

Ссылки из таблиц открытых файлов в процессах указывают на структуры "открытый файл" в ядре (см. главу про файлы). Таким образом, два процесса получают доступ к **одной и той же** структуре и, следовательно, имеют **общий указатель чтения/записи** для этого файла. Поэтому, когда процессы "отец" и "сын" пишут по дескриптору **fd**, они пользуются одним и тем же указателем R/W, т.е. информация от обоих процессов записывается последовательно. На принципе наследования и совместного использования открытых файлов основан также системный вызов *pipe*.

Порожденный процесс наследует также: реакции на сигналы (!!!), текущий каталог, управляющий терминал, номер владельца процесса и группу владельца, и.т.п.

При системном вызове *exec*() (который заменяет **программу**, выполняемую процессом, на программу из указанного файла) все открытые каналы также достаются в наследство новой программе (а не закрываются).

6.5.3. Процесс-копия это хорошо, но не совсем то, что нам хотелось бы. Нам хочется запустить программу, содержащуюся в выполняемом файле (например *a.out*). Для этого существует системный вызов *exec*, который имеет несколько разновидностей. Рассмотрим только две:

```
char *path;
char *argv[], *envp[], *arg0, ..., *argn;
execle(path, arg0, arg1, ..., argn, NULL, envp);
execve(path, argv, envp);
```

Системный вызов *exec* заменяет **программу**, выполняемую данным процессом, на программу, загружаемую из файла **path**. В данном случае **path** должно быть полным именем файла или именем файла от текущего каталога:

```
/usr/bin/vi   a.out  ../mybin/xkick
```

Файл должен иметь код доступа "выполнение". Первые два байта файла (в его заголовке), рассматриваемые как *short int*, содержат так называемое "магическое число" (*A_MAGIC*), свое для каждого типа машин (смотри include-файл <**a.out.h**>). Его помещает в начало выполняемого файла редактор связей *ld* при компоновке программы из объектных файлов. Это число должно быть правильным, иначе система откажется запускать программу из этого файла. Бывает несколько разных магических чисел, обозначающих разные способы организации программы в памяти. Например, есть вариант, в котором сегменты *text* и *data* склеены вместе (тогда *text* не разделяем между процессами и не защищен от модификации программой), а есть - где данные и текст находятся в раздельных адресных пространствах и запись в *text* запрещена (аппаратно).

Остальные аргументы вызова - **arg0**, **...**, **argn** - это аргументы функции *main* новой программы. Во второй форме вызова аргументы не перечисляются явно, а заносятся в массив. Это позволяет формировать произвольный массив строк-аргументов во время работы программы:

```
char *argv[20];
argv[0]="ls"; argv[1]="-l"; argv[2]="-i"; argv[3]=NULL;
execv( "/bin/ls", argv);
        либо
execl( "/bin/ls", "ls","-l","-i", NULL):
```

В результате этого вызова текущая программа завершается (но не процесс!) и вместо нее запускается программа из заданного файла: сегменты *stack*, *data*, *text* старой программы уничтожаются; создаются **новые** сегменты *data* и *text*, загружаемые из файла **path**; отводится сегмент *stack* (первоначально - не очень большого размера); сегмент *user* сохраняется от старой программы (за исключением реакций на сигналы, отличных от *SIG_DFL* и *SIG_IGN* - они будут сброшены в *SIG_DFL*). Затем будет вызвана функция *main* новой программы с аргументами **argv:**

```
void main( argc, argv )
    int argc; char *argv[]; { ... }
```

Количество аргументов - **argc** - подсчитает сама система. Строка NULL не подсчитывается.

Процесс остается тем же самым - он имеет тот же паспорт (только адреса сегментов изменились); тот же номер (**pid**); все открытые прежней программой файлы остаются открытыми (с теми же дескрипторами); текущий каталог также наследуется от старой программы; сигналы, которые игнорировались ею, также будут игнорироваться (остальные сбрасываются в *SIG_DFL*). Зато "сущность" процесса подвергается перерождению - он выполняет теперь **иную** программу. Таким образом, системный вызов *exec* осуществляет вызов функции *main*, находящейся в **другой программе**, передавая ей свои аргументы в качестве входных.

Системный вызов *exec* может не удаться, если указанный файл **path** не существует, либо вы не имеете права его выполнять (такие коды доступа), либо он не является выполняемой программой (неверное магическое число), либо слишком велик для данной машины (системы), либо файл открыт каким-нибудь процессом (например еще записывается компилятором). В этом случае продолжится выполнение прежней программы. Если же вызов успешен - возврата из *exec* не происходит вообще (поскольку управление передается в другую программу).

Аргумент **argv**[0] обычно полагают равным **path**. По нему программа, имеющая несколько имен (в файловой системе), может выбрать ЧТО она должна делать. Так программа */bin/ls* имеет альтернативные имена *lr*, *lf*, *lx*, *ll*. Запускается **одна и та же** программа, но в зависимости от **argv**[0] она далее делает разную работу.

Аргумент **envp** - это "окружение" программы (см. начало этой главы). Если он не задан - передается окружение текущей программы (наследуется содержимое массива, на который указывает переменная *environ*); если же задан явно (например, окружение скопировано в какой-то массив и часть переменных подправлена или добавлены новые переменные) - новая программа получит новое окружение. Напомним, что окружение можно прочесть из предопределенной переменной char ***environ*, либо из третьего аргумента функции *main* (см. начало главы), либо функцией *getenv*().

Системные вызовы *fork* и *exec* не склеены в один вызов потому, что между *fork* и *exec* в процессе-сыне могут происходить некоторые действия, нарушающие симметрию процесса-отца и порожденного процесса: установка реакций на сигналы, перенаправление ввода/вывода, и.т.п. Смотри пример "интерпретатор команд" в приложении. В *MS DOS*, не имеющей параллельных процессов, вызовы *fork*, *exec* и *wait* склеены в **один** вызов *spawn*. Зато при этом приходится делать перенаправления ввода-вывода в **порождающем** процессе перед *spawn*, а после него - восстанавливать все как было.

6.5.4. Завершить **процесс** можно системным вызовом

```
void exit( unsigned char retcode );
```

Из этого вызова не бывает возврата. Процесс завершается: сегменты *stack*, *data*, *text*, *user* уничтожаются (при этом все открытые процессом файлы закрываются); память, которую они занимали, считается свободной и в нее может быть помещен другой процесс. Причина смерти отмечается в паспорте процесса - в структуре *proc* в таблице процессов внутри ядра. Но паспорт еще не уничтожается! Это состояние процесса называется "зомби" - живой мертвец.

В паспорт процесса заносится код ответа **retcode**. Этот код может быть прочитан процессом-родителем (тем, кто создал этот процесс вызовом *fork*). Принято, что код 0 означает успешное завершение процесса, а любое положительное значение 1..255 означает неудачное завершение с таким кодом ошибки. Коды ошибок заранее не предопределены: это личное дело процессов отца и сына - установить между собой какие-то соглашения по этому поводу. В старых программах иногда писалось *exit*(-1); Это некорректно - код ответа должен быть неотрицателен; код -1 превращается в код 255. Часто используется конструкция *exit*(**errno**);

Программа может завершиться не только **явно** вызывая *exit*, но и еще двумя способами:

- если происходит возврат управления из функции *main*(), т.е. она кончилась - то вызов *exit*() делается неявно, но с непредсказуемым значением **retcode**;
- процесс может быть **убит** сигналом. В этом случае он не выдает никакого кода ответа в процесс-родитель, а выдает признак "процесс убит".

6.5.5. В действительности *exit*() - это еще не сам системный вызов завершения, а стандартная **функция**. Сам системный вызов называется *_exit*(). Мы можем переопределить функцию *exit*() так, чтобы по окончании программы происходили некоторые действия:

```
void exit(unsigned code){
  /* Добавленный мной дополнительный оператор: */
  printf("Закончить работу, "
         "код ответа=%u\n", code);

  /* Стандартные операторы: */
  _cleanup();  /* закрыть все открытые файлы.
                * Это стандартная функция *-*/
  _exit(code); /* собственно сисвызов */
}

int f(){ return 17; }
void main(){
  printf("aaaa\n"); printf("bbbb\n"); f();
  /* потом откомментируйте это:  exit(77); */
}
```

Здесь функция *exit* вызывается **неявно** по окончании *main*, ее подставляет в программу компилятор. Дело в том, что при запуске программы *exec*-ом, первым начинает выполняться код так называемого "**стартера**", подклеенного при сборке программы из файла */lib/crt0.o*. Он выглядит примерно так (в действительности он написан на ассемблере):

```
... // вычислить argc, настроить некоторые параметры.
main(argc, argv, envp);
exit();
```

или так (взято из проекта *GNU*[*] -):

```
int errno = 0;
char **environ;
_start(int argc, int arga)
{
/* OS and Compiler dependent!!!! */
char **argv = (char **) &arga;
char **envp = environ = argv + argc + 1;
/* ... возможно еще какие-то инициализации,
 * наподобие setlocale( LC_ALL, "" ); в SCO UNIX */
```

```
    exit (main(argc, argv, envp));
    }
```

Где должно быть

```
    int main(int argc, char *argv[], char *envp[]){
              ...
            return 0;  /* вместо exit(0); */
    }
```

Адрес функции *_start*() помечается в одном из полей заголовка файла формата *a.out* как адрес, на который система должна передать управление после загрузки программы в память (точка входа).

Какой код ответа попадет в *exit*() в этих примерах (если отсутствует **явный** вызов *exit* или *return*) - непредсказуемо. На *IBM PC* в вышенаписанном примере этот код равен 17, то есть значению, возвращенному последней вызывавшейся функцией. Однако это не какое-то специальное соглашение, а случайный эффект (так уж устроен код, создаваемый этим компилятором).

6.5.6. Процесс-отец может дождаться окончания своего потомка. Это делается системным вызовом *wait* и нужно по следующей причине: пусть отец - это интерпретатор команд. Если он запустил процесс и продолжил свою работу, то **оба** процесса будут предпринимать попытки читать ввод с клавиатуры терминала - интерпретатор ждет команд, а запущенная программа ждет данных. Кому из них будет поступать набираемый нами текст - непредсказуемо! Вывод: интерпретатор команд должен "заснуть" на то время, пока работает порожденный им процесс:

```
    int pid;  unsigned short status;
              ...
    if((pid = fork()) == 0 ){
            /* порожденный процесс */
             ...  // перенаправления ввода-вывода.
             ...  // настройка сигналов.
            exec(....);
            perror("exec не удался"); exit(1);
    }
    /* иначе это породивший процесс */
    while((pid = wait(&status)) > 0 )
      printf("Окончился сын pid=%d с кодом %d\n",
              pid, status >> 8);
    printf( "Больше нет сыновей\n");
```

wait приостанавливает* - выполнение вызвавшего процесса до момента окончания **любого** из порожденных им процессов (ведь можно было запустить и нескольких сыновей!). Как только какой-то потомок окончится - *wait* проснется и выдаст номер (**pid**) этого потомка. Когда никого из живых "сыновей" не осталось - он выдаст (-1). Ясно, что процессы могут оканчиваться не в том порядке, в котором их порождали. В переменную **status** заносится в специальном виде код ответа окончившегося процесса, либо номер сигнала, которым он был убит.

```
    #include <sys/types.h>
    #include <sys/wait.h>
            ...
    int status, pid;
            ...
    while((pid = wait(&status)) > 0){
        if( WIFEXITED(status)){
          printf( "Процесс %d умер с кодом %d\n",
                           pid,            WEXITSTATUS(status));
        } else if( WIFSIGNALED(status)){
          printf( "Процесс %d убит сигналом %d\n",
                           pid,             WTERMSIG(status));
          if(WCOREDUMP(status)) printf( "Образовался core\n" );
          /* core - образ памяти процесса для отладчика adb */
        } else if( WIFSTOPPED(status)){
          printf( "Процесс %d остановлен сигналом %d\n",
                           pid,            WSTOPSIG(status));
        } else if( WIFCONTINUED(status)){
          printf( "Процесс %d продолжен\n",
                           pid);
        }
    }
            ...
```

Если код ответа нас не интересует, мы можем писать *wait*(NULL).

Если у нашего процесса не было или больше нет живых сыновей - вызов *wait* ничего не ждет, а возвращает значение (-1). В написанном примере цикл *while* позволяет дождаться окончания **всех** потомков.

В тот момент, когда процесс-отец получает информацию о причине смерти потомка, паспорт умершего процесса наконец **вычеркивается** из таблицы процессов и может быть переиспользован новым процессом. До того, он хранится в таблице процессов в состоянии "zombie" - "живой мертвец". Только для того, чтобы кто-нибудь мог узать статус его завершения.

Если процесс-отец завершился **раньше** своих сыновей, то кто же сделает *wait* и вычеркнет паспорт? Это сделает процесс номер 1: */etc/init*. Если отец умер раньше процессов-сыновей, то система заставляет процесс номер 1 "усыновить" эти процессы. *init* обычно находится в цикле, содержащем в начале вызов *wait*(), то есть ожидаетокончания любого из своих сыновей (а они у него всегда есть, о чем мы поговорим подробнее чуть погодя). Таким образом *init* занимается чисткой таблицы процессов, хотя это не единственная его функция.

Вот схема, поясняющая жизненный цикл любого процесса:

```
        |pid=719,csh
        |
    if(!fork())------->--------* pid=723,csh
        |                       |                загрузить
     wait(&status)           exec("a.out",...) <-- a.out
        :                    main(...){           с диска
        :                      |
        :pid=719,csh           | pid=723,a.out
      спит(ждет)             работает
        :                      |
        :                    exit(status) умер
        :                    }
     проснулся <---проснись!--RIP
        |
        |pid=719,csh
```

Заметьте, что номер порожденного процесса не обязан быть **следующим** за номером родителя, а только **больше** него. Это связано с тем, что **другие** процессы могли создать в системе новые процессы **до** того, как наш процесс издал свой вызов *fork*.

6.5.7. Кроме того, *wait* позволяет отслеживать остановку процесса. Процесс может быть приостановлен при помощи посылки ему сигналов *SIGSTOP*, *SIGTTIN*, *SIGTTOU*, *SIGTSTP*. Последние три сигнала посылает при определенных обстоятельствах драйвер терминала, к примеру *SIGTSTP* - при нажатии клавиши *CTRL/Z*. Продолжается процесс посылкой ему сигнала *SIGCONT*.

В данном контексте, однако, нас интересуют не сами эти сигналы, а другая схема манипуляции с отслеживанием статуса порожденных процессов. Если указано **явно**, система может посылать процессу-родителю сигнал *SIGCLD* в момент изменения статуса любого из его потомков. Это позволит процессу-родителю **немедленно** сделать *wait* и немедленно отразить изменение состояние процесса-потомка в своих внутренних списках. Данная схема программируется так:

```
void pchild(){
        int pid, status;

        sighold(SIGCLD);
        while((pid = waitpid((pid_t) -1, &status, WNOHANG|WUNTRACED)) > 0){
          dorecord:
                записать_информацию_об_изменениях;
        }
        sigrelse(SIGCLD);

        /* Reset */
        signal(SIGCLD, pchild);
}
        ...
main(){
        ...
        /* По сигналу SIGCLD вызывать функцию pchild */
        signal(SIGCLD, pchild);
        ...
        главный_цикл;
}
```

Секция с вызовом *waitpid* (разновидность вызова *wait*), прикрыта парой функций *sighold-sigrelse*, запрещающих приход сигнала *SIGCLD* внутри этой критической секции. Сделано это вот для чего: если процесс начнет модифицировать таблицы или списки в районе метки dorecord:, а в этот момент придет еще один сигнал, то функция *pchild* будет вызвана рекурсивно и тоже попытается модифицировать таблицы и списки, в которых еще остались незавершенными перестановки ссылок, элементов, счетчиков. Это приведет к разрушению данных.

Поэтому сигналы должны приходить последовательно, и функции *pchild* вызываться также последовательно, а не рекурсивно. Функция *sighold* откладывает доставку сигнала (если он случится), а *sigrelse* - разрешает

доставить накопившиеся сигналы (но если их пришло несколько одного типа – все они доставляются как **один** такой сигнал. Отсюда цикл вокруг *waitpid*).

Флаг *WNOHANG* – означает "не ждать внутри вызова *wait*", если ни один из потомков не изменил своего состояния; а просто вернуть код (-1)". Это позволяет вызывать *pchild* даже без получения сигнала: ничего не произойдет. Флаг *WUNTRACED* – означает "выдавать информацию также об остановленных процессах".

* – Процесс может узнать его вызовом **pid**=*getpid*();

* – *cleanup*() закрывает файлы, открытые *fopen*()ом, "вытряхая" при этом данные, накопленные в буферах, в файл. При аварийном завершении программы файлы все равно закрываются, но уже не явно, а операционной системой (в вызове *_exit*). При этом содержимое недосброшенных буферов будет **утеряно**.

* – программы, распространяемые в исходных текстах из *Free Software Foundationtion* (FSF). Среди них – *C++* компилятор *g++* и редактор *emacs*. Смысл слов *GNU* – "generally not *UNIX*" – проект был основан как противодействие начавшейся коммерциализации *UNIX* и закрытию его исходных текстов. "Сделать как в *UNIX*, но лучше".

* – "Живой" процесс может пребывать в одном из нескольких состояний: процесс ожидает наступления какого-то события ("спит"), при этом ему не выделяется время процессора, т.к. он не готов к выполнению; процесс готов к выполнению и стоит в очереди к процессору (поскольку процессор выполняет другой процесс); процесс готов и выполняется процессором в данный момент. Последнее состояние может происходить в двух режимах пользовательском (выполняются команды сегмента *text*) и системном (процессом был издан системный вызов, и сейчас выполняется функция в ядре). Ожидание события бывает только в системной фазе – внутри системного вызова (т.е. это "синхронное" ожидание). Неактивные процессы ("спящие" или ждущие ресурса процессора) могут быть временно откачаны на диск.

Назад | Содержание | Вперед

[Главная] [Гостевая]