Лабораторная работа 2

Процессы. Системные вызовы fork() и exec().

Процесс в Linux (как и в UNIX) - это программа, которая выполняется в отдельном защищенном виртуальном адресном пространстве. Когда пользователь регистрируется в системе, автоматически создается новый процесс, в котором выполняется оболочка (shell), например, /bin/bash. Если интерпретатору (shell) встречается команда, соответствующая выполняемому файлу, интерпретатор выполняет ее, начиная с точки входа (entry point). Для С-программ entry point - это функция main. Запущенная программа тоже может создать процесс, т.е. запустить какую-то программу и ее выполнение тоже начнется с функции main.

Часть времени процесс выпоняется в режиме задачи (пользовательском режиме) и тогда он выпоняет собственный код, часть времени процесс выпоняется в режиме ядра и тогда он выполняет реентерабельный код операционной системы.

В Linux поддерживается классическая схема мультипрограммирования. Linux поддерживает параллельное (или квазипараллельное при наличии только одного процессора) выполнение процессов пользователя. Каждый процесс выполняется в собственном защищенном виртуальном адресном пространстве. Это значит, что нги один процесс не обратиться в адресное пространствог другого процесса. Процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на всю систему в целом.

Ядро предоставляет системные вызовы для создания новых процессов и для управления запущенными процессами. Любая программа может начать выполняться только если другой процесс ее запустит.

Для создания процессов используется системный вызов fork(). В Unix и Unix-подобных ОС процесс создается системным вызовом **fork(). В ядре системный вызов fork() для разных ОС Unix и Linux реализуется разными функциями ядра.**

Системный вызов fork() создает новый процесс, который является копией процесса-предка: процесс-потомок наследует адресное пространство процесса-предка , **дескрипторы** всех открытых файлов и сигнальную маску. Другими словами, программа которую выполняет процесс-потомок является программой родительского процесса. Процесс потомок имеет идентичные с родителем области данных и стека. Процесс-потомок начинает работу в режиме задачи после возвращения из системного вызова fork(). Тот факт, что образы памяти, переменные, регистры и все остальное и у родительского процесса, и у дочернего идентичны, могло бы привести к невозможности различить эти процессы, однако, системный вызов fork() возвращает дочернему процессу число 0, а родительскому — PID (Process IDentifier — идентификатор процесса) дочернего процесса. Оба процесса обычно проверяют возвращаемое значение и действуют соответственно:

```
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
pid_t childpid ; /* если fork завершился успешно, pid > 0 в родительском процессе */
if ((childpid = fork( )==-1)
{
   perror("Can't fork"); /* fork потерпел неудачу (например, память или какая-либо */
   exit(1)                    /*таблица заполнена) */
}
```

```
else if (childpid == 0)
 {
    /* здесь располагается дочерний код */
 } else
    {
      /* здесь располагается родительский код */
    }
return 0;
}
```

Если вызов **`fork()`** завершается аварийно, он возвращает -1. Обычно это происходит из-за ограничения числа дочерних процессов, которые может иметь родительский процесс (`CHILD_MAX`), в этом случае переменной `errno` будет присвоено значение `EAGAIN`. Если для элемента таблицы процессов недостаточно места или не хватает виртуальной памяти, переменная `errno` получит значение `ENOMEM`.

В **старых** Unix-системах код предка копировался в адресное пространство процесса-потомка. В современных системах применяется, так называемая, оптимизация fork() или, как говорят, «умный» fork: для процесса потомка создаются собственные карты трансляции адресов (таблицы страниц), но они ссылаются на адресное пространство процесса-предка (на страницы предка). При этом для страниц адресного пространства предка права доступа меняются на only- read и устанавливается флаг – copy-on-write. Если или предок или потомок попытаются изменить страницу, возникнет исключение по правам доступа. Выполняя это исключение супервизор обнаружит флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. Таким образом код процесса-предка не копируется полностью, а создаются только копии страниц, которые редактируются.

Рассмотрим более подробно, что же делается при выполнении вызова fork() (рис.1):

1. Резервируется пространство свопинга для данных и стека процесса-потомка;
2. Назначается идентификатор процесса PID и структура proc потомка;
3. Инициализируется структура proc потомка. Некоторые поля этой структуры копируются от процесса-родителя: идентификаторы пользователя и группы, маски сигналов и группа процессов. Часть полей инициализируется 0. Часть полей инициализируется специфическими для потомка значениями: PID потомка и его родителя, указатель на структуру proc родителя;
4. Создаются карты трансляции адресов для процесса-потомка;
5. Выделяется область u потомка и в нее копируется область u процесса-предка;
6. Изменяются ссылки области u на новые карты адресации и пространство свопинга;
7. Потомок добавляется в набор процессов, которые разделяют область кода программы, выполняемой процессом-родителем;
8. Постранично дублируются области данных и стека родителя и модифицируются карты адресации потомка;
9. Потомок получает ссылки на разделяемые ресурсы, которые он наследует : открытые файлы (потомок наследует дескрипторы) и текущий рабочий каталог;
10. Инициализируется аппаратный контекст потомка путем копирования регистров родителя;
11. Поместить процесс-потомок в очередь готовых процессов;
12. Возвращается PID в точку возврата из системного вызова в родительском процессе и 0 - в процессе-потомке.

Процессы, например, в Unix BSD описываются структурой proc:

Основные поля структуры proc охватывают:

- идентификацию: каждый процесс обладает уникальным *идентификатором процесса* (process ID, или *PID*) и относится к определенной *группе процессов*. В современных версиях системы каждому процессу также присваивается *идентификатор сеанса* (session ID);
- расположение карты адресов ядра для области u данного процесса;
- текущее состояние процесса;
- предыдущий и следующий указатели, связывающие процесс с очередью планировщика (или очередью приостановленных процессов, если данный процесс был заблокирован);
- канал «сна» для заблокированных процессов (см. раздел 7.2.3);
- приоритеты планирования задач и связанную информацию (см. главу 5);
- информацию об обработке сигналов: маски игнорируемых, блокируемых, передаваемых и обрабатываемых сигналов (см. главу 4);
- информацию по управлению памятью;
- указатели, связывающие эту структуру со списками активных, свободных или завершенных процессов (зомби);
- различные флаги;
- указатели на расположение структуры в *очереди хэша*, основанной на PID;
- информация об иерархии, описывающая взаимосвязь данного процесса с другими.

Эта структура содержит всю информацию о процессе. Информация находится по адресу /usr/src/sys/sys/proc.h:

```
/*
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes.  The process structure and the substructures
 * are always addressable except for those marked "(PROC ONLY)" below,
 * which might be addressable only on a processor on which the process
 * is running.
 */
struct  proc {
        struct  proc *p_forw;           /* Doubly-linked run/sleep queue. */
        struct  proc *p_back;
        struct  proc *p_next;           /* Linked list of active procs */
        struct  proc **p_prev;          /*    and zombies. */

        /* substructures: */
        struct  pcred *p_cred;          /* Process owner's identity. */
        struct  filedesc *p_fd;         /* Ptr to open files structure. */
        struct  pstats *p_stats;        /* Accounting/statistics (PROC ONLY). */        struct  plimit *p_limit;        /* Process
limits. */
        struct  vmspace *p_vmspace;     /* Address space. */
        struct  sigacts *p_sigacts;     /* Signal actions, state (PROC ONLY). */
```

```
#define p_ucred         p_cred->pc_ucred
#define p_rlimit        p_limit->pl_rlimit

        int     p_flag;                 /* P_* flags. */
        char    p_stat;                 /* S* process status. */
        char    p_pad1[3];

        pid_t   p_pid;                  /* Process identifier. */
        struct  proc *p_hash;   /* Hashed based on p_pid for kill+exit+... */
        struct  proc *p_pgrpnxt; /* Pointer to next process in process group. */
        struct  proc *p_pptr;   /* Pointer to process structure of parent. */
        struct  proc *p_osptr;  /* Pointer to older sibling processes. */

/* The following fields are all zeroed upon creation in fork. */
#define p_startzero     p_ysptr
        struct  proc *p_ysptr;  /* Pointer to younger siblings. */
        struct  proc *p_cptr;   /* Pointer to youngest living child. */
        pid_t   p_oppid;        /* Save parent pid during ptrace. XXX */
        int     p_dupfd;        /* Sideways return value from fdopen. XXX */

        /* scheduling */
        u_int   p_estcpu;       /* Time averaged value of p_cpticks. */
        int     p_cpticks;      /* Ticks of cpu time. */
        fixpt_t p_pctcpu;       /* %cpu for this process during p_swtime */
        void    *p_wchan;       /* Sleep address. */
        char    *p_wmesg;       /* Reason for sleep. */
        u_int   p_swtime;       /* Time swapped in or out. */
        u_int   p_slptime;      /* Time since last blocked. */

        struct  itimerval p_realtimer;  /* Alarm timer. */
        struct  timeval p_rtime;        /* Real time. */
        u_quad_t p_uticks;              /* Statclock hits in user mode. */
        u_quad_t p_sticks;              /* Statclock hits in system mode. */
        u_quad_t p_iticks;              /* Statclock hits processing intr. */

        int     p_traceflag;            /* Kernel trace points. */
        struct  vnode *p_tracep;        /* Trace to vnode. */

        int     p_siglist;              /* Signals arrived but not delivered. */

        struct  vnode *p_textvp;        /* Vnode of executable. */

        char    p_lock;                 /* Process lock (prevent swap) count. */
        char    p_pad2[3];              /* alignment */

/* End area that is zeroed on creation. */
#define p_endzero       p_startcopy

/* The following fields are all copied upon creation in fork. */
#define p_startcopy     p_sigmask

        sigset_t p_sigmask;     /* Current signal mask. */
        sigset_t p_sigignore;   /* Signals being ignored. */
        sigset_t p_sigcatch;    /* Signals being caught by user. */

        u_char  p_priority;     /* Process priority. */
        u_char  p_usrpri;       /* User-priority based on p_cpu and p_nice. */
        char    p_nice;         /* Process "nice" value. */
        char    p_comm[MAXCOMLEN+1];

        struct  pgrp *p_pgrp;   /* Pointer to process group. */
```

```
        struct  sysentvec *p_sysent; /* System call dispatch information. */

        struct  rtprio p_rtprio;        /* Realtime priority. */
/* End area that is copied on creation. */
#define p_endcopy       p_addr
        struct  user *p_addr;   /* Kernel virtual addr of u-area (PROC ONLY). */
        struct  mdproc p_md;    /* Any machine-dependent fields. */

        u_short p_xstat;        /* Exit status for wait; also stop signal. */
        u_short p_acflag;       /* Accounting flags. */
        struct  rusage *p_ru;   /* Exit information. XXX */
};
```

Еще раз об оптимизации fork()

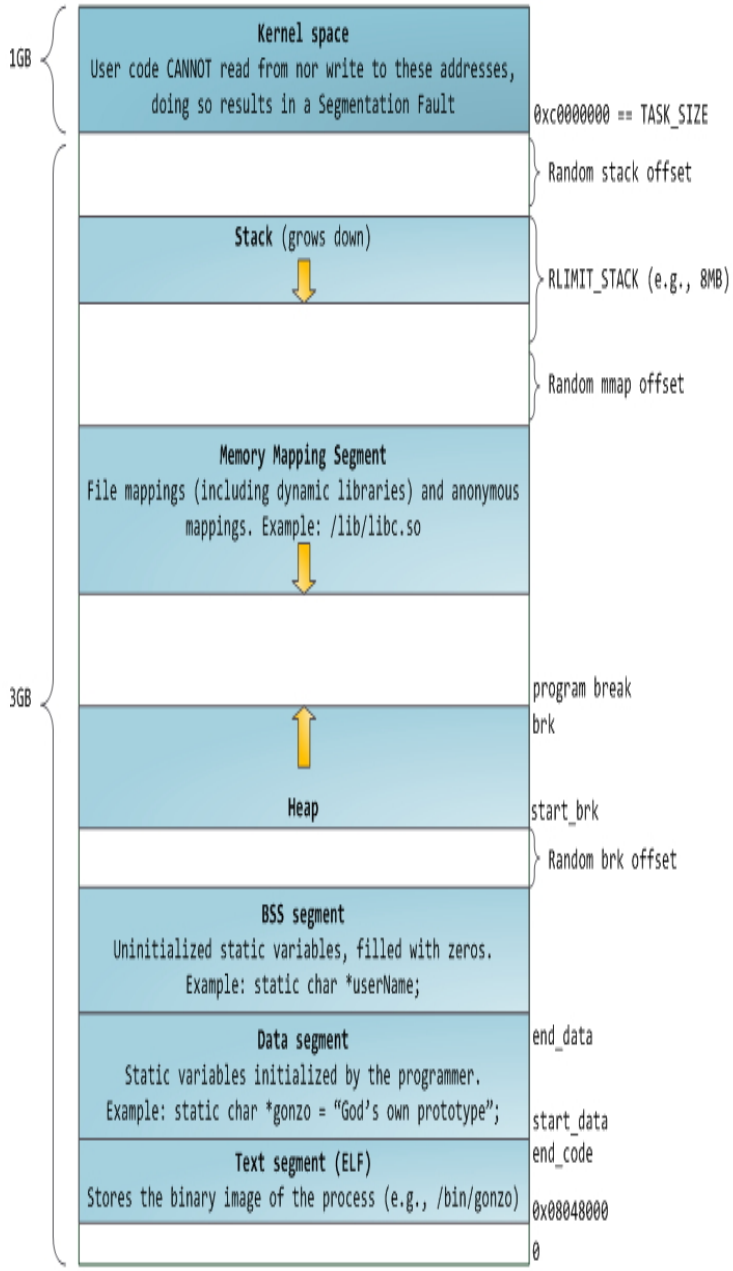
Системный вызов fork() должен предоставить процессу-потомку логически идентичную копию адресного пространства его родителя. В большинстве случаев потомок заменяет предоставленное адресное пространство, так как сразу же после выполнения fork вызывает exec или exit. Таким образом, создание копии адресного пространства (так, как это реализовано в первых системах UNIX) является не оптимальной процедурой.

Вышеописанная проблема была решена двумя различными способами. Сначала был разработан метод копирования при записи, впервые нашедший реализацию в ОС System V и в настоящий момент используемый в большинстве систем UNIX. При таком подходе

- страницы данных и стека родителя временно получают атрибут «только для чтения» и маркируются как «копируемые при записи»;
- потомок получает собственные карты трансляции адресов (таблицы страниц), которые ссылаются на страницы процесса-предка, т.е. процесс-потомок использует одни и те же страницы памяти вместе со своим родительским процессом;
- если кто-то из них (родитель или потомок) попытается изменить страницу памяти, произойдет ошибочная исключительная ситуация по правам доступа ,так как страницы доступны только для чтения. Затем ядро системы запустит обработчик исключительной ситуации, который обнаружит, что страница помечена как «копируемая при записи», и создаст новую ее новую копию, которую уже можно изменять.

Таким образом, происходит копирование только тех страниц памяти, которые требуется изменять, а не всего адресного пространства целиком. Если потомок вызовет exec() или exit(), то защита страниц памяти вновь станет обычной, и флаг «копирования при записи» будет сброшен.

В системе BSD UNIX представлен несколько иной подход к решению проблемы, реализованный в новом системном вызове vfork(). Функция vfork() не производит копирования. Вместо этого процесс-родитель предоставляет свое адресное пространство потомку (потомок получает карты трансляции адресов предка) и блокируется до тех пор, пока тот не вернет его. Затем происходит выполнение потомка в адресном пространстве родительского процесса до того времени, пока не будет произведен вызов exec или exit(), после чего ядро вернет родителю его адресное пространство и выведет его из состояния сна. Системный вызов vfork() выполняется очень быстро, так как не копирует даже карты адресации. Адресное пространство передается потомку простым копированием регистров карты адресации. Однако следует отметить, что вызов vfork является достаточно опасным, так как позволяет одному процессу использовать и даже изменять адресное пространство другого процесса. Это свойство vfork используют различные программы, такие как csh. Программист может воспользоваться vfork() вместо fork(), если планирует вслед за ним сразу вызвать exec().



Процесс-сирота

Системный вызов fork() создает новый процесс – процесс-потомок. Связь родитель – потомок создает иерархию процессов (см. указатели на потомков в struct proc). Если родительский процесс завершается раньше своих потомков, то в системе выполняется так называемое усыновление: процесс-потомок усыновляется процессом с идентификатором 1 (процессом «открывшим» терминал и создавшим терминальную группу). В Linux Ubuntu процесс-потомок усыносляется процессом-посредником systemd –user.

Задание:

Написать программу, запускающую новый процесс системным вызовом fork(). В предке вывести собственный идентификатор (функция getpid()), идентификатор группы (функция getpgrp()) и идентификатор потомка. В процессе-потомке вывести собственный идентификатор, идентификатор предка (функция getppid()) и идентификатор группы. Убедиться, что при завершении процесса-предка потомок,

который продолжает выполняться, получает идентификатор предка (PPID), равный 1 или идентификатор процесса-посредника.

Ожидание процессом - предком завершения потомков

Системный вызов **wait()** блокирует родительский процесс до момента завершения дочернего. При этом процесс-предок получает статус завершения процесса-потомка.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

. Вызов возвращает PID дочернего процесса. Обычно это дочерний процесс, который завершился. Сведения о состоянии позволяют родительскому процессу определить статус завершения дочернего процесса, т.е. значение, возвращенное из функции **main потомка** или переданное функции **exit()**. Если `stat_loc` не равен пустому указателю, информация о состоянии будет записана в то место, на которое указывает этот параметр.

Интерпретировать информацию о состоянии процесса можно с помощью макросов, описанных в файле sys/wait.h и приведенных в табл. 1

Таблица 1

Макрос	Описание
`WIFEXITED(stat_val)`	Ненулевой, если дочерний процесс завершен нормально
`WEXITSTATUS(stat_val)`	Если `WIFEXITED` ненулевой, возвращает код завершения дочернего процесса
`WIFSIGNALED(stat_val)`	Ненулевой, если дочерний процесс завершается неперехватываемым сигналом
`WTERMSIG(stat_val)`	Если `WIFSIGNALED` ненулевой, возвращает номер сигнала
`WIFSTOPPED(stat_val)`	Ненулевой, если дочерний процесс остановился
`WSTOPSIG(stat_val)`	Если `WIFSTOPPED` ненулевой, возвращает номер сигнала

Пример 2. Системный вызов `wait()`

В этом упражнении вы слегка измените программу, чтобы можно было подождать и проверить код состояния дочернего процесса. Назовите новую программу wait.c.

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

Следующий фрагмент программы ждет окончания дочернего процесса:

```
if (pid > 0) {
 int status;
 pid_t child_pid;
 child_pid = wait(&status);
 printf("Child has finished: PID = %d\n", child_pid);
 if (WIFEXITED(stat_val))
  printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
 else printf("Child terminated abnormally\n");
}
```

Есть еще один системный вызов, который можно применять для ожидания дочернего процесса. Он называется `waitpid()` и применяется для ожидания завершения определенного процесса.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Аргумент `pid` — конкретный дочерний процесс, окончания которого нужно ждать. Если он равен –1, `waitpid()` возвращает информацию о любом дочернем процессе. Как и вызов `wait()`, он записывает информацию о состоянии процесса в то место, которое указывает

аргумент `stat_loc`, если последний не равен пустому указателю. Аргумент `options` позволяет изменить поведение `waitpid`. Наиболее полезная опция `WNOHANG` мешает вызову `waitpid()` приостанавливать выполнение вызвавшего его процесса. Она применяется для определения, завершился ли какой-либо из дочерних процессов, и если нет, то можно продолжить выполнение. Остальные опции такие же, как в вызове `wait()`.

Итак для того, чтобы родительский процесс периодически проверял, завершился ли конкретный дочерний процесс, можно использовать следующий вызов:

```
waitpid(child_pid, (int *)0, WNOHANG);
```

Он вернет ноль, если дочерний процесс не завершился и не остановлен, или `child_pid`, если это произошло. Вызов waitpid вернет -1 в случае ошибки и установит переменную `errno`. Это может произойти, если нет дочерних процессов (`errno` равна `ECHILD`), если вызов прерван сигналом (`EINTR`) или аргумент `options` неверный (`EINVAL`).

Задание 2: написать программу по схеме первого задания, но в процессе-предке выполнить системный вызов wait(). Убедиться, что в этом случае идентификатор процесса потомка на 1 больше идентификатора процесса-предка.

Системный вызов exec()

Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов exec(), параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц.

Системный вызов exec выполняет следующие действия:

1. Разбирает путь к исполняемому файлу и осуществляет доступ к нему.
2. Проверяет, имеет ли вызывающий процесс полномочия на выполнение файла.
3. Читает заголовок и проверяет, что он действительно исполняемый[1].
4. Если для файла установлены биты SUID или SGID, то эффективные идентификаторы UID и GID вызывающего процесса изменяет на UID и GID, соответствующие владельцу файла.
5. Копирует аргументы, передаваемые в exec, а также *переменные среды* в пространство ядра, после чего текущее пользовательское пространство готово к уничтожению.
6. Выделяет пространство свопинга для областей данных и стека.
7. Высвобождает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи vfork, производится возврат старого адресного пространства родительскому процессу.
8. Выделяет карты трансляции адресов для нового текста, данных и стека.
9. Устанавливает новое адресное пространство. Если область текста активна (какой-то другой процесс уже выполняет ту же программу), то она будет совместно использоваться с этим процессом. В других случаях пространство должно инициализироваться из выполняемого файла. Процессы в системе UNIX обычно разбиты на страницы, что означает, что каждая страница считывается в память только по мере необходимости.
10. Копирует аргументы и переменные среды обратно в новый стек приложения.
11. Сбрасывает все обработчики сигналов в действия, определенные по умолчанию, так как функции обработчиков сигналов не существуют в новой программе. Сигналы, которые были проигнорированы или заблокированы перед вызовом exec, остаются в тех же состояниях.
12. Инициализирует аппаратный контекст. При этом большинство регистров сбрасывается в 0, а указатель команд получает значение точки входа программы.

В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

. Например: системный вызов exec() запускает команду ls.

```
if (fork()==0) wait(0);
else execl("ls", "ls", 0); /* порожденный процесс */
```

Дополнительно:

В стандарте POSIX системный вызов exec объявлен в заголовочной файде unistd.h на языке C. Подобная функция объявлена в process.h для DOS, OS/2,Windows.

```
int execl(char const *path, char const *arg0, ...);
int execle(char const *path, char const *arg0, ..., char const *
const *envp);
```

```
int execlp(char const *file, char const *arg0, ...);
int execv(char const *path, char const * const * argv);
int execve(char const *path, char const * const *argv, char const
* const *envp);
int execvp(char const *file, char const * const *argv);
```

The base of each is **exec** (execute), followed by one or more letters:

e – An array of pointers to environment variables is explicitly passed to the new process image.

l – Command-line arguments are passed individually to the function.

p – Uses the PATH environment variable to find the file named in the *path* argument to be executed.

v – Command-line arguments are passed to the function as an array of pointers.

path

The argument specifies the path name of the file to execute as the new process image. Arguments beginning at *arg0* are pointers to arguments to be passed to the new process image. The *argv* value is an array of pointers to arguments.

arg0

The first argument *arg0* should be the name of the executable file. Usually it is the same value as the *path* argument. Some programs may incorrectly rely on this argument providing the location of the executable, but there is no guarantee of this nor is it standardized across platforms.

envp

Argument *envp* is an array of pointers to environment settings. The *exec* calls named ending with an *e* alter the environment for the new process image by passing a list of environment settings through the *envp* argument. This argument is an array of character pointers; each element (except for the final element) points to a null-terminated string defining anenvironment variable.

Each null-terminated string has the form:

```
name=value
```

where *name* is the environment variable name, and *value* is the value of that that variable. The final element of the *envp* array must be null.

In the `execl`, `execlp`, `execv`, and `execvp` calls, the new process image inherits the current environment variables.

Так как новый процесс не создается, идентификатор процесса не меняется, но код, данные, куча и стек процесса заменяются кодом, данными, кучей и стеком новой программы.

Передать значение окружению программы может глобальная переменная `environ`. Другой вариант — дополнительный аргумент в функциях`execle` и `execve`, способный передавать строки, используемые как окружение новой программы.

Если вы хотите применить функцию `exec` для запуска программы `ps`, можно выбирать любую функцию из семейства `exec`, как показано в вызовах приведенного далее фрагмента программного кода:

```
#include <unistd.h>
/* Пример списка аргументов */

/* Учтите, что для argv[0] необходимо имя программы */
char *const ps_argv[] = {"ps", "ax", 0};
/* Не слишком полезный пример окружения */
char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};
/* Возможные вызовы функций exec */
execl("/bin/ps", "ps", "ax", 0);
/* предполагается, что ps в /bin */
execlp("ps", "ps", "ax", 0);
/* предполагается, что /bin в PATH */
execle("/bin/ps", "ps", "ax", 0, ps_envp);
/* передается свое окружение */
execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Процессы «зомби»

Применение вызова `fork()` для создания процессов может оказаться очень полезным, но необходимо отслеживать дочерние процессы. Когда дочерний процесс завершается, связь его с родителем сохраняется до тех пор, пока родительский процесс в свою очередь не завершится нормально, или не вызовет `wait()`. Следовательно, запись о дочернем процессе не исчезает из таблицы процессов немедленно. Становясь неактивным, дочерний процесс все еще остается в системе, поскольку его код завершения должен быть сохранен, на случай если родительский процесс в дальнейшем вызовет `wait`.

Процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов.

Можно увидеть переход процесса в состояние зомби, если изменить количество сообщений в программе из примера 1 с вызовом `fork()`. Если дочерний процесс выводит меньше сообщений, чем родительский, он закончится первым и будет существовать как зомби, пока не завершится родительский процесс.

Задание 3:

Написать программу, в которой процесс-потомок вызывает системный вызов exec(), а процесс-предок ждет завершения процесса-потомка.

Программные каналы

Системный вызов pipe() создает неименованный программный канал. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор. Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов.

Программные каналы имеют встроенные средства взаимоисключения — массив файловых дескрипторов: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают. Для этого определяется массив файловых дескрипторов, как показано в примере:

```
 int fd[2];
pipe(fd);
if (( pid=fork())<0)
 {
   err_sys("Error fork()");
  }
else if (pid==0)
{
  /*child*/
   close(fd[0]);
   write(fd[1], … );
}
else {
        /*parent*/
           close(fd[1]);
           read(fd[0], … );


      }
```

Задание 4:

Написать программу, в которой предок и потомок обмениваются сообщением через программный канал.

Сигналы

Сигнал - способ информирования процесса ядром о происшествии какого-то события. Если возникает несколько однотипных событий, процессу будет подан только один сигнал. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло.

Установить реакцию на поступление сигнала можно с помощью системного вызова signal
func = signal(snum, function);

snum - номер сигнала, a function - адрес функции, которая должна быть выполнена при поступлении указанного сигнала. Возвращаемое значение - адрес функции, которая будет реагировать на поступление сигнала. Вместо function можно указать ноль или единицу. Если был указан ноль, то при поступлении сигнала snum выполнение процесса будет прервано аналогично вызову exit. Если указать единицу, данный сигнал будет проигнорирован, но это возможно не для всех процессов.

Например:

```
# include <iostream.h>
# include <signal.h>
void catch_sig(int sig_numb)
{
   signal(sig_numb, catch_sig);
   cout<<”catch_sig”<<sig_numb<<endl;
}
int main(void)
{
  signal(SIGNERM, catch_sig);
  signal(SIGINT, SIG_IGN);
  signal(SIGSEGV, SIG_DFL);
}
```

С помощью системного вызова kill() можно сгенерировать сигналы и передать их другим процессам.
kill(pid, snum);
где pid - идентификатор процесса, а snum - номер сигнала, который будет передан процессу. Обычно kill() используется для того, чтобы принудительно завершить ("убить") процесс.
Pid состоит из идентификатора группы процессов и идентификатора процесса в группе. Если вместо pid указать нуль, то сигнал snum будет направлен всем процессам, относящимся к данной группе (понятие группы процессов аналогично группе пользователей). В одну группу включаются процессы, имеющие общего предка, идентификатор группы процесса можно изменить с помощью системного вызова setpgrp. Если вместо pid указать -1, ядро передаст сигнал всем процессам, идентификатор пользователя которых равен идентификатору текущего выполнения процесса, который посылает сигнал.

Таблица 2.

Сигналы POSIX

POSIX определяет 28 сигналов, которые можно классифицировать следующим образом:

Название	Действие по умолчанию	Описание	Тип
SIGABRT	Завершение с дампом памяти	Сигнал посылаемый функцией `abort()`	Управление
SIGALRM	Завершение	Сигнал истечения времени, заданного `alarm()`	Уведомление
SIGBUS	Завершение с дампом памяти	Неправильное обращение в физическую память	Исключение
SIGCHLD	Игнорируется	Дочерний процесс завершен или остановлен	Уведомление
SIGCONT	Продолжить выполнение	Продолжить выполнение ранее остановленного процесса	Управление
SIGFPE	Завершение с дампом	Ошибочная арифметическая операция	Исключение

	памяти		
SIGHUP	Завершение	Закрытие терминала	Уведомление
SIGILL	Завершение с дампом памяти	Недопустимая инструкция процессора	Исключение
SIGINT	Завершение	Сигнал прерывания (Ctrl-C) с терминала	Управление
SIGKILL	**завершение**	**Безусловное завершение**	**управление**
SIGPIPE	Завершение	Запись в разорванное соединение (пайп, сокет)	Уведомление
SIGQUIT	Завершение с дампом памяти	Сигнал «Quit» с терминала (Ctrl-\)	Управление
SIGSEGV	Завершение с дампом памяти	Нарушение при обращении в память	Исключение
SIGSTOP	**остановка процесса**	**Остановка выполнения процесса**	**управление**
SIGTERM	Завершение	Сигнал завершения (сигнал по умолчанию для утилиты kill)	Управление
SIGTSTP	Остановка процесса	Сигнал остановки с терминала (Ctrl-Z).	Управление
SIGTTIN	Остановка процесса	Попытка чтения с терминала фоновым процессом	Управление
SIGTTOU	Остановка процесса	Попытка записи на терминал фоновым процессом	Управление
SIGUSR1	Завершение	Пользовательский сигнал № 1	Пользовательский
SIGUSR2	Завершение	Пользовательский сигнал № 2	Пользовательский
SIGPOLL	Завершение	Событие, отслеживаемое `poll()`	Уведомление
SIGPROF	Завершение	Истечение таймера профилирования	Отладка
SIGSYS	Завершение с дампом памяти	Неправильный системный вызов	Исключение
SIGTRAP	Завершение с дампом памяти	Ловушка трассировки или брейкпоинт	Отладка
SIGURG	Игнорируется	На сокете получены срочные данные	Уведомление
SIGVTALRM	Завершение	Истечение «виртуального таймера»	Уведомление
SIGXCPU	Завершение с дампом памяти	Процесс превысил лимит процессорного времени	Исключение
SIGXFSZ	Завершение с дампом памяти	Процесс превысил допустимый размер файла	Исключение

Сигналы (точнее их номера) описаны в файле singnal.h

Задание 5:

В программу с программным каналом включить собственный обработчик сигнала. Использовать сигнал для изменения хода выполнения программы.

Дополнительная информация

Для нормального завершение процесса используется вызов exit(status);
где status - это целое число, возвращаемое процессу-предку для его информирования о причинах завершения процесса-потомка.
Вызов exit может задаваться в любой точке программы, но может быть и неявным, например при выходе из функции main (при программировании на C) оператор return 0 будет воспринят как системный вызов exit(0);

nohup - игнорирование сигналов прерывания

nohup команда [аргумент]

nohup выполняет запуск команды в режиме игнорирования сигналов. Не игнорируются только сигналы SIGHUP и SIGQUIT.

kill - принудительное завершение процесса

kill [-номер сигнала] PID

где PID - идентификатор процесса, который можно узнать с помощью команды ps.