Технология запуска и завершения процессов

Linux предоставляет два системных вызова *fork()* и *exec()*
для создания (порождения) процесса, и для запуска новой программы.
Фрагмент кода порождения процесса:

```
#include <sys/types.h>
#include <unistd.h>
. . .
pid_t fork(void);
```

Порожденный или *дочерний* процесс, является точной копией
родительского процесса. В частности, дочерний процесс наследует
такие атрибуты родительского, как:
- идентификаторы пользователя и группы,
- переменные окружения,
- диспозицию сигналов и их обработчики,
- ограничения, накладываемые на процесс,
- все файловые дескрипторы,
- текущий и корневой каталог,
- управляющий терминал.

Виртуальная память дочернего процесса не отличается от
образа родительского:
такие же сегменты кода (*code*), данных (*data*), стека (*stack*),
разделяемой памяти (*shared memory*) и т. д.
После возврата из вызова *fork()* , который происходит
и в родительский и в дочерний процессы,
оба начинают выполнять одну и же инструкцию.

Немногочисленные различия между дочерним родительским процессами:
- дочернему процессу присваивается уникальный идентификатор
- идентификаторы родительского процесса PPID у этих процессов различны,
- значение, возвращаемое системным вызовом *fork()* различно для родителя и потомка.

При этом значение, возвращаемое родителю, равно *PID*
дочернего процесса,
а дочернему процессу *fork() возвращ*ает значение, равное 0.
Если же *fork()* возвращает *-1*, то это свидетельствует о
неудаче в выполнении, т.е. об ошибке
(естественно, возврат *-1* происходит только в тот процесс,
который пытался выполнить системный вызов).

Таким образом, возвращаемое значение *fork()* позволяет определить,
кто является родителем, а кто - потомком, и соответственно
разделить дальнейшую функциональность этих процессов, которые будут
выполняться параллельно..

Фрагмент кода программы:

```
. . .
int pid;
pid = fork();
if (pid == -1) {
        perror("fork"); exit (1);
}

If (pid == 0)       {
/* Эта часть кода выполняется дочерним процессом */
                printf ("процесс-потомок\n");
        }
else                {
/* Эта часть кода выполняется родительским процессом */
                printf ("процесс-родитель\n");
        }
. . .
```

Программа иллюстрирует процедуру клонирования процесса и *асинхронность поведения* процесса-родителя и процесса-потомка.

```
 /* Программа  forkdemo.cpp */
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
main()
{
 int i;
  if (fork()) {    /* Это процесс-родитель */
      for(i=0; i<1000 ; i++)
          printf("\t\tPARENT %d\n", i);
      }
       else {      /* Это процесс-потомок  */
        for(i=0; i<1000 ; i++)
            printf("CHILD %d\n",i);
        }
}
```

Асинхронность и конкурентность процессов в данном примере иллюстрируется случайным характером переключения вывода от процесса к процессу на этапе исполнения.

Невозможно точно предугадать порядок вывода, несмотря на то, что известен исходный текст программы.
Переключение ввода-вывода процессов зависит от длины кванта времени, выделяемого каждому из них планировщиком процессов, от буферизации, выполняемой библиотекой ввода-вывода, и даже от характеристик конкретного компьютера.

Для загрузки другого исполняемого файла из процесса потомка предназначен системный вызов *exec().*

При выполнении *exec()* не создается новый процесс, а образ существующего (процесса-потомка) полностью заменяется на загружаемый из указанного при вызове *exec()* исполняемого файла.

Загружаемая программа наследует от процесса-потомка такие атрибуты как:
- идентификаторы процесса PID и PPID,
- идентификаторы пользователя и группы,
- ограничения, накладываемые на процесс,
- текущий и корневой каталоги,
- управляющий терминал,
- файловые дескрипторы, для которых не установлен флаг FD CLOEXEC.

Наследование характеристик процесса играет важную роль. Так наследование идентификаторов владельцев процесса гарантирует преемственность привилегий и, таким образом, неизменность привилегий пользователя при работе в Linux. Наследование файловых дескрипторов позволяет установить направления ввода/вывода для нового процесса или новой программы.

Системный вызов *exec()* представлен несколькими модификациями, различающимися

- тем, как задан путь к исполняемому файлу загружаемой программы (абсолютно или относительно и используя переменную окружения PATH);

- тем, откуда берутся переменные окружения для загружаемой программы (сохраняются существующие или передается новый набор переменных);

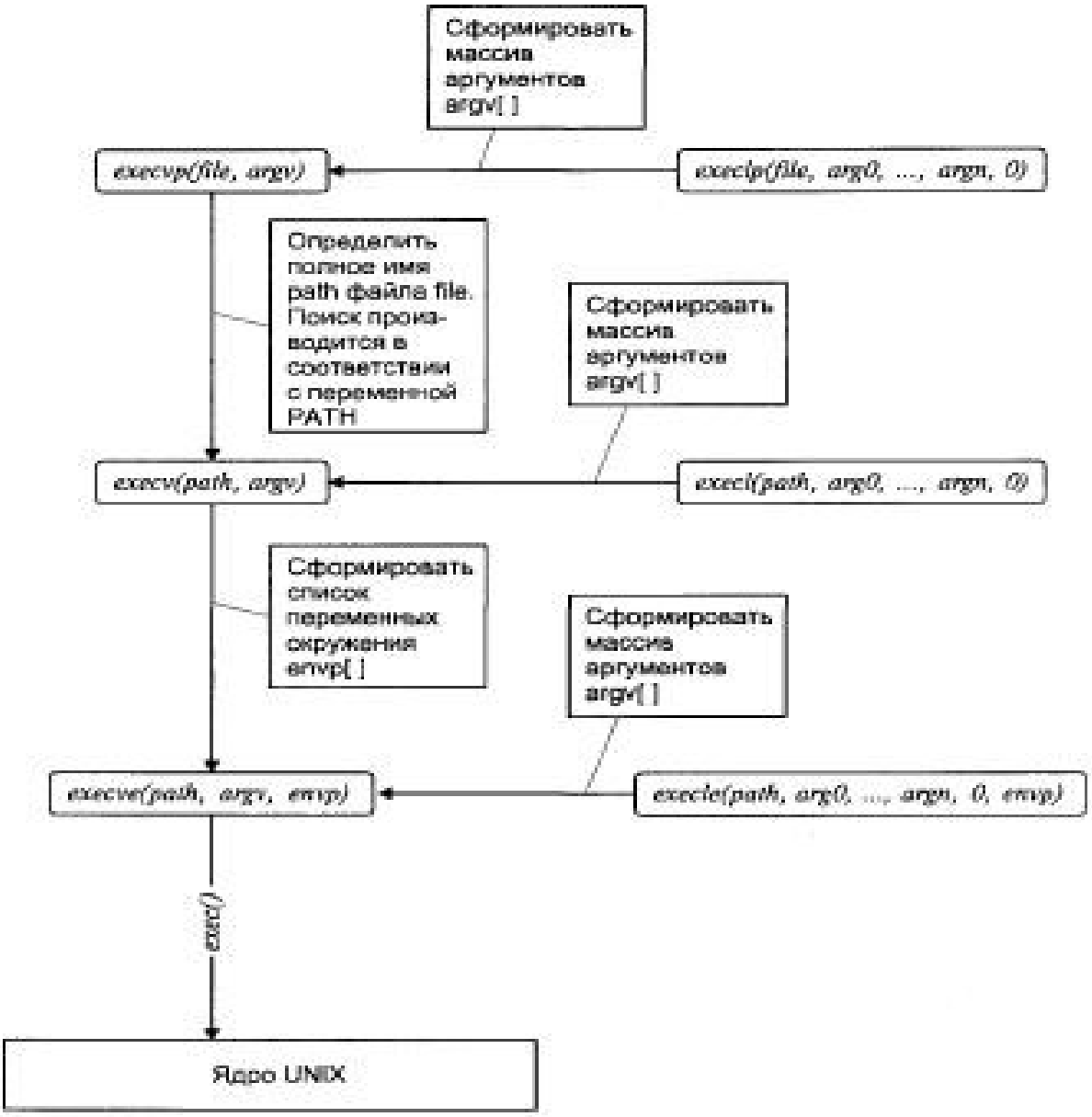
- способом передачи аргументов командной строки (в виде списка параметров при вызове (*explicit list*) или как вектор (*vector*)).

Каждая разновидность вызова *exec()* имеет в мнемонике еще до двух дополнительных символов.

Так, например, простейший в применении вызов *execlp()* , не указывает абсолютный путь, наследует окружение, и передает параметры командной строки в виде списка. Имеет постфикс *lp*.
Такой же по семантике вызов, но передающий параметра командной строки вектором, имеет постфикс *vp (т.е. это вызов execvp()).*
Аналог вызова *execlp()* , но передающий путь к загружаемому файлу в абсолютном виде, имеет мнемонику *execl()*.

Аналог вызова *execvp()* , но передающий путь к загружаемому файлу в абсолютном виде, имеет мнемонику *execv()*.
Наконец, вызовы *execle()* и *execve()* передают путь к загружаемому файлу в абсолютном виде, но не наследуют переменные окружения,
а передают их новый набор, либо вектором, либо списком, соответственно.

Картинка иллюстрирующая многообразие вызовов *exec()* :



Первый параметр вызова *exec()* всегда указывает имя загружаемого файла. Далее следует список аргументов *arg0, arg1, … , argn* ,
либо указатель на вектор с этим списком.
Причем, требуется, чтобы первый элемент списка указывал на имя загружаемого файла (опять), а последний был нулевым указателем.

В программе *tinymenu* используется версия системного вызова *exec*, позволяющая передавать параметры командной строки списком, а переменные окружения наследовать.
С помощью вызова *execlp* в примере запускаются команды интерпретатора *shell*.
При этом, в нормальной ситуации, после вызова *execlp* и отработки соответствующей команды *shell*, управление, обратно, само по себе, не возвращается.

/* Программа tinymenu.cpp */

```
#include<stdio.h>
#include<unistd.h>
main()
{
        /* Фиксированный список команд */
static char *cmd[]={"who","ls","date"};
int i;
        /* Подсказка номера команды */
printf("0=who, 1=ls, 2=date:");
scanf("%d",&i);
        /* Запуск выбранной команды на исполнение */
    execlp(cmd[i], cmd[i], 0);
printf("Command not found\n");
        /* запуск не удачный */
}
```

Как организовать возврат управления в родительский процесс из потомка когда потомок завершается ?

Посредством системного вызова *wait()* родительский процесс переводится в состояние ожидания, выход из которого происходит по событию завершения дочернего процесса, а именно, по выполнению в потомке вызова *exit()*.

/* Программа tinyexit.cpp */

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
main()
{
      /* Фиксированный список команд */
static char *cmd[]={"who","ls","date"};
int i;
   while(1){
      /* Подсказка номера команды */
   printf("0=who, 1=ls, 2=date:");
   scanf("%d",&i);
      /* Если номер неверный, родительский процесс завершается */
      if(i<0 || I>2)
            exit();
      if (fork()==0){                  /* Дочерний процесс */
      /* Процесс-потомок исполняет выбранную команду */
   execlp(cmd[i], cmd[i], 0);
   printf("Command not found\n");
      /* Запуск не удачный */
   exit(1);
   }
   else
      {     /* Родительский процесс дожидается завершения дочернего */
      wait(0);
      }
   }
}
```

Системный вызов *wait()* возвращает значение *PID* идентификатора завершившегося дочернего процесса.
Параметр этого вызова позволяет передавать по ссылке информацию о *статусе завершения* процесса-потомка.

В программе *wait_parent* из процесса родителя запускаются три потомка, и затем системный вызов *wait()* отслеживает завершение каждого из них, сообщая, какой именно процесс закончил свое исполнение и какой *код завершения* был при этом передан.
В качестве процесса-потомка выступает программа *wait_child* , запускаемая из родителя с помощью *execlp* трижды
с разными параметрами командной строки.

В потомках случайным образом формируется *код завершения*
(для случаев нормального завершения),
а также варьируется и сам *способ завершения*.
При завершении по сигналу, номер сигнала передается процессу-родителю
в младшем байте *статуса завершения*.
При нормальном же завершении, формируемый *код завершения*
находится в передаваемой информации о *статусе завершения*,
в байте, следующем после младшего (остальные байты обнуляются).

/* Программа wait_parent.cpp */

```
/* Процесс-родитель дожидается завершения процесса-потомка */
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
main()
{
pid_t pid, w;
int   i, status;
char  value[3];
   for(i=0; i<3; ++i)
   {          /* Запуск трех дочерних процессов */
   if ((pid=fork())==0){
      sprintf(value, "%d", i);
      execlp("wait_child", "wait_child", value, (char *)0);
      }
      else          /* Подразумевается успешный запуск  */
         printf("Forked child %d\n", pid);
   }
                  /* Ожидание завершения дочерних процессов */
   while((w=wait(&status)) && w!=-1){
      if(w!=-1)
         printf("Wait on PID: %d returns status of:
                %04X\n", w, status);
      }
      exit(0);
}
```

/* Программа wait_child.cpp */

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

```
main(int argc, char *argv[])
{
pid_t pid;
int     ret_value;
pid  =  getpid();
ret_value = (int) (pid % 256);
srand((unsigned) pid);
sleep(rand() %5);
if(atoi(*(argv+1)) % 2){
                /* Подразумевается, что при запуске argv[1] существует */
    printf("Child %d is terminating with signal 0009\n", pid);
    kill(pid, 9);                   /* Процесс сам себя завершает - харакири */
    }
    else{
        printf("Child %d is terminating with exit (%04X)\n", ret_value);
        exit(ret_value);
        }
}
```

Байты статуса завершения

Status info:

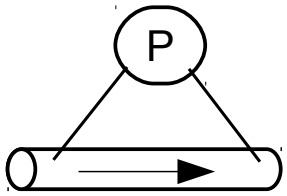
	byte 3	byte 2	byte 1	byte 0
нормальное завершение:	*0*	*0*	*exit code*	*0*
вследствие получения сигнала:	*0*	*0*	*0*	*signal #*

Организация конвейеров

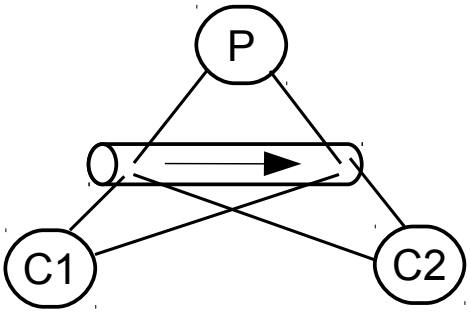
Конвейеры (в оригинале *pipes*) представляют собой традиционный и достаточно распространенный механизм взаимодействия процессов в Linux. Например, простейшая конструкция вида *who* | *sort* , запускаемая в *shell*, создает два конкурентных процесса, соединенных конвейером так, что поток вывода первого из них попадает в поток ввода второго.

Для организации однонаправленных конвейеров в приложениях применяется системный вызов *pipe()*.
В качестве параметра вызова выступает указатель на массив двух файловых дескрипторов.
Первый дескриптор связан с выходом образуемого конвейера, а второй - со входом.

В программе *whosortpipe* реализуется типовой подход к созданию однонаправленного канала передачи данных между процессами потомками.
Вначале родительский процесс *P* вызовом *pipe()* создает конвейер.

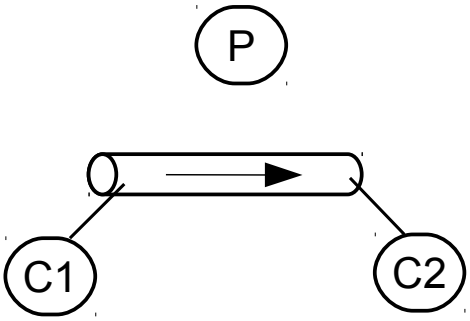


Затем порождает пару дочерних процессов, каждый из которых наследует от родителя оба открытых вызовом *pipe()* дескриптора.



Потомки *C1* и *C2* закрывают дескрипторы, соответствующие тем концам конвейера. В которых они не нуждаются, а на оставшиеся перенастраивают (ассоциируют) свой *стандартный вывод* и *стандартный ввод*, соответственно.

После того, как процесс родитель *P* со своей стороны, закроет оба дескриптора (отключится), в системе образуется однонаправленный канал, по которому в поток ввода одного дочернего процесса *C2* будут поступать данные из выходного потока другого процесса *C1*.



ПРИМЕЧАНИЕ:
Каждая запущенная из командного интерпретатора *shell* программа (команда) получает три открытых потока ввода/вывода:
- стандартный ввод (дескриптор 0),
- стандартный вывод (дескриптор 1),
- стандартный вывод ошибок (дескриптор 2).

По умолчанию все эти потоки ассоциированы с терминалом.
То есть любая программа, не использующая потоки, кроме стандартных, будет ожидать ввода с клавиатуры терминала, а весь вывод этой программы, включая сообщения об ошибках,
будет происходить на экран терминала.

```
/* Программа whosortpipe.cpp */

#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
main()
{
int     fds[2];
pipe(fds);   /* Создание конвейера */
        /* Один дочерний процесс подключает стандартный ввод stdin к
выходу конвейера
        и закрывает другой конец */
    if (fork()==0){
        dup2(fds[0], 0);  /* перенаправление ввода */
        close(fds[1]);
    execlp("sort", "sort", 0);
    }
/* Другой дочерний процесс подключает стандартный вывод stdout ко
входу конвейера
        и закрывает другой конец  */
    else if (fork()==0){
            dup2(fds[1], 1);      /* перенаправление вывода */
            close(fds[0]);
        execlp("who", "who", 0);
```

```
        }
/* Процесс-родитель закрывает оба конца конвейера и ожидает
завершения обоих
        дочерних процессов */
        else{
            close(fds[0]);
            close(fds[1]);
            wait(0);
            wait(0);
            }
}
```

При замещении дочерних процессов другими программами (посредством exec() вызова)
дескрипторы входа и выхода конвейера остаются обычно нетронутыми,
и конвейер продолжает служить для передачи данных от одного процесса другому, посредством обычных системных вызовов *write()* и *read()*.

Конвейер представляется, как разновидность файла,
в котором можно сохранять ограниченный объем данных,
причем доступ к этим данным организован в манере
дисциплины обслуживания очереди типа FIFO (first-in first-out).
В большинстве систем конвейеры ограничены 10-ю логическими блоками по 512 байт.

Заголовочный файл <limits.h> содержит
предопределенную константу PIPE_BUF ,
задающую предельный размер буфера для каждой конкретной реализации.

Система синхронизирует обращения процессов к конвейеру.
При попытке записи в заполненный канал, процесс будет автоматически заблокирован до тех пор, пока не освободится место для получения данных. Попытка чтения из пустого конвейера также приведет к блокировке, пока в конвейере не появятся данные.

Недостатками приведенного механизма передачи данных между процессами являются:
- невозможность аутентификации процесса
на другой стороне конвейера, то есть, процесс, читающий из *pipe*,
не может знать, кто туда пишет;
- неудобно также и то, что конвейеры должны создаваться
предварительно, еще до момента запуска процессов;
- обмен данными возможен лишь между родственными процессами,
имеющими общего предка, передавшего потомкам файловые дескрипторы открытого конвейера;
- конвейеры не позволяют обмен данными по сети,
т.е. оба процесса должны выполняться на одном и том же компьютере.

Частично эти ограничения снимаются применением другой разновидности конвейеров, так называемых, *named pipes*.

Существует и другой путь создания обычных конвейеров для исполнения команд *shell*. Он основан на системных вызовах *popen()* и *pclose()*.

При вызове *popen()* выполняется последовательность сразу нескольких действий:
автоматически генерируется дочерний процесс, который,
в свою очередь, запускает (*exec()*-ом) на исполнение команду *shell*,
которая указана в качестве первого параметра *popen()*. Второй параметр вызова указывает, как интерпретировать дескриптор файла,
указатель на который возвращает *popen()* в случае успешного завершения организации конвейера.

При указании "*w*" родительский процесс может писать данные
в стандартный ввод команды *shell*,
что дает возможность процессу потомку, выполняющему эту команду shell, читать эти данные (как стандартный ввод).

При указании "*r*" в качестве второго параметра *popen()*, наоборот,
процесс родитель читает данные со стандартного вывода команды *shell*, запущенной дочерним процессом.

Программа *cmdpipe* иллюстрирует данный способ создания конвейера.

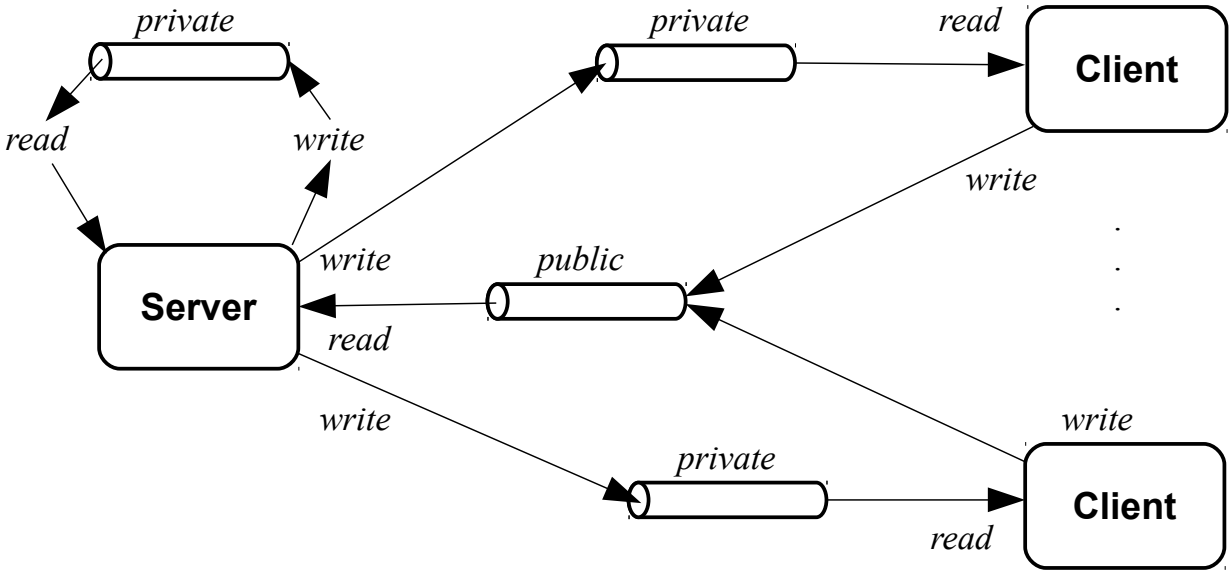
```
/* Программа cmdpipe.cpp */

/* Использование команд popen и pclose  */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<limits.h>
main()
{
FILE  *fin, *fout;
char  buffer[PIPE_BUF};
int    n;
   if(argc<3){
       fprintf(stderr, "Usage %s cmd1 cmd2\n", argv[0]);
       exit(1);
       }
fin  = popen(argv[1], "r");
fout = popen(argv[2], "w");
while((n = read(fileno(fin), buffer, PIPE_BUF))>0)
   write(fileno(fout), buffer, n);
pclose(fin);
pclose(fout);
exit(0);
}
```

Другой тип конвейера, называемый в оригинале *named pipe*, требует для своей организации системного вызова *mknod()*.
Вообще, системный вызов *mknod()* создает специальные файлы блочных или символьных устройств (не только конвейеры).
Именованные конвейеры (*named pipes*) в смысле организации взаимодействия между процессами, близки к обычным конвейерам (*pipes*), однако обладают некоторыми преимуществами. Так, при создании именованного конвейера, запись о нем, с соответствующими правами доступа, попадает в список файлов каталога. Это делает возможным применение таких конвейеров для коммуникации между любыми процессами, а не только между родственными. Процесс должен быть только наделен соответствующими правами. Именованный конвейер может создаваться, как из приложения, так и на уровне *shell*.

Первый параметр системного вызова *mknod()* передает указатель на полное имя создаваемого конвейера в каталоге.
Второй параметр задает его тип. Для создания очереди *FIFO это* тип *S_FIFO*.
Второй параметр может также еще содержать логическое слагаемое, задающее права доступа к создаваемому ресурсу.
В списке каталога информация об атрибутах этого ресурса, помимо прав доступа к нему, содержит еще на первой позиции маркер *"p"*, указывающий на то, что это конвейер (*pipe*).
Третий параметр *mknod()* задает тип устройства (для типа *S_FIFO* это - 0).

В программах *pipe_server* и *pipe_client приводится* реализация и совместное использование существующих механизмов обмена *pipes* и *named pipes* для организации взаимодействия процессов работающих в клиент-серверной парадигме. Заголовочный файл *pipe_local* является общим для обеих программ.

Иллюстрация построенной в данном примере схемы межпроцессного взаимодействия.



Процесс сервер запускается на исполнение в фоновом (*background*) режиме (символ & после имени запускаемого процесса).

Клиентские процессы запускаются последовательно в обычном режиме (*foreground*). Сервер создает именованный конвейер *PUBLIC*, доступный и для любого клиентского процесса.

Каждый же из клиентских процессов генерирует собственный именованный конвейер со своим уникальным именем. Далее клиент в ответ на свой запрос получает с консоли от пользователя команду *shell*, которую необходимо выполнить.

Затем клиент пишет в общедоступный именованный конвейер *PUBLIC* то уникальное имя, которое он присвоил собственному конвейеру при его создании, и пишет затем команду *shell*, полученную от пользователя.

На другом конце *PUBLIC* конвейера сервер читает эти данные от клиента. Сервер выполняет полученную команду *shell* с помощью системного вызова *popen()* (запускающего дочерний процесс для исполнения этой команды). По создаваемому тем же вызовом *popen()* собственному конвейеру сервер получает результат исполнения команды *shell*.
После этого сервер пишет в собственный именованный конвейер клиента результат исполнения команды *shell*, полученной ранее от данного клиента. Клиент читает эти данные и выводит их на *стандартный вывод ошибок* (чтобы можно было бы отличать данные, пришедшие клиенту от сервера).

```
/* Программа pipe_local.h */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<limits.h>
#include<string.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#define PUBLIC "/tmp/PUBLIC"
#define B_SIZ (PIPE_BUF / 2)
struct message{
char  fifo_name[B_SIZ];
char  cmd_line[B_SIZ];
};
```

```
/* Программа  pipe_server.cpp */
#include<pipe_local.h>
void main(void)
{
int     n, done, dummyfifo, privatefifo, publicfifo;
static char buffer[PIPE_BUF];
FILE *fin;
struct message msg;
       /* Создание очереди типа public FIFO */
mknod(PUBLIC, S_IFIFO | 0666, 0);
       /* Открыть public FIFO на чтение и запись */
if ((publicfifo=open(PUBLIC, O_RDONLY))==-1) ||
    (dummyfifo=open(PUBLIC, O_WRONLY | O_NDELAY))==-1){
   perror(PUBLIC);
   exit(1);
   }
            /* Сообщение можно прочитать из public конвейера */
while(read(publicfifo, (char *) &msg, sizeof(msg))>0){
   n = done = 0; /* Очистка счетчиков | флагов */
   do{                      /* Попытка открытия private FIFO */
      if ((privatefifo=open(msg.fifo_name, O_WRONLY | O_DELAY))==-1
      sleep(3);                  /* Задержка по времени  */
      else{                      /* Открытие успешно */
         fin = popen(msg.cmd_line, "r");/* Исполнение shell cmd,
полученной от клиента */
         write(privatefifo, "\n", 1);  /* Подготовка очередного вывода */
         while((n=read(fileno(fin), buffer, PIPE_BUF))>0){
            write(privatefifo, buffer, n);/* Вывод в private FIFO к клиенту */
            memset(buffer, 0x0, PIPE_BUF);    /* Очистка буфера */
            }
            pclose(fin);
            close(privatefifo);
            done = 1;                /* Запись произведена успешно */
         }
      }while(++n<5 && !done);

      if(!done)                        /* Указание на неудачный исход */
         write(fileno(stderr), "\nNOTE: SERVER ** NEVER **
                        accessed private FIFO\n", 48);
   }
}
```

```
/* Программа  pipe_client.cpp */
#include<pipe_local.h>
void main(void)
{
int     n, privatefifo, publicfifo;
static char buffer[PIPE_BUF];
struct message msg;
                /* Создание имени для очереди типа private FIFO */
sprintf(msg.fifo_name, "/tmp/fifo %d", getpid());
                /* Создание очереди private FIFO */
if (mknod(msg.fifo_name, S_IFIFO | 0666, 0)<0){
    perror(msg.fifo_name);
    exit(1);
    }
                /* Открытие очереди типа public FIFO на запись */
if ((publicfifo=open(PUBLIC, O_WRONLY))==-1){
    perror(PUBLIC);
    exit(2);
    }
while(1){   /* Зацикливание */
    write(fileno(stdout), "\ncmd>", 6);
    memset(msg.cmd_line, 0x0, B_SIZ);          /* Очистка */
    n = read(fileno(stdin), msg.cmd_line, B_SIZ);      /* Чтение с консоли
shell cmd */
        if(!strncmp("quit", msg.cmd_line, n-1))
        break;                              /*Завершение? */
    write(publicfifo, (char *) &msg, sizeof(msg)); /* to PUBLIC */
        /* Открытие private FIFO для чтения вывода исполненной команды
(shell cmd) */
        if((privatefifo = open(msg.fifo_name, O_RDONLY))==-1){
            perror(msg.fifo_name);
            exit(3);
            }
        /* Чтение private FIFO и вывод на результата на
стандартный_вывод_ошибок */
    while((n=read(privatefifo, buffer, PIPE_BUF))>0){
        write(fileno(stderr), buffer, n);
        }
        close(privatefifo);
    }
    close(publicfifo);
    unlink(msg.fifo_name);
}
```

Сигналы

Сигналы являются способом передачи уведомлений между процессами или между ядром системы и процессами.
Уведомление о некотором произошедшем событии.
Сигналы можно рассматривать, как простейшую форму межпроцессного взаимодействия, хотя на самом деле они больше напоминают программные прерывания, которые вторгаются в ход нормального выполнения процесса.

Событие, порождающее сигнал, может быть действием пользователя, может быть вызвано другим процессом или ядром.

Сигналы используются, как простейшее, по мощное средство межпроцессного взаимодействия.

Действия, вызываемые для обработки сигналов, являются принципиально *асинхронными*.

Каждый сигнал имеет уникальное символьное имя и соответствующий ему номер.
Например, сигнал прерывания, посылаемый процессу при нажатии пользователем клавиш <Ctrl>+<C>, имеет имя SIGINT.
Сигнал, генерируемый комбинацией <Ctrl>+<\>, называется SIGQUIT.
Современные версии Linux насчитывают несколько десятков различных сигналов.

Сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова *kill()*.
Первым параметром вызова *kill()* является PID процесса, которому данный сигнал предназначен,
а вторым параметром указывается какой именно сигнал надо отправить.

В число ситуаций при которых ядро отправляет процессу (или целой группе) определенные сигналы, входят и так называемые аппаратные *особые ситуации*.
Например деление на ноль, обращение к недопустимой области памяти и др.
В зависимости от типа сигнала (события его породившего), реакцией системы на его получение (обработкой по умолчанию) может быть:
Exit – выполнение действий совпадающих с семантикой системного вызова *exit()*;
Core – создание файла снимка текущего состояния ядра, затем выполнение *exit()*; файл *core,* хранящий образ памяти процесса, может быть впоследствии проанализирован программой отладчиком для определения состояния процесса непосредственно перед завершением;
Stop – остановка и подвешивание процесса;
Ignore – игнорировать, отбросить сигнал.

Вместо того, чтобы предоставлять системе самой обрабатывать сигнал, процесс может обладать собственным *обработчиком*, выполняющим при получении сигнала какие-либо специфические действия в контексте требований самого этого процесса.
Если для сигнала устанавливается функция-обработчик, то говорят, что сигнал перехватывается (относительно стандартного действия).

Для организации собственной обаботки используется вызов *signal() .*
Системный вызов *signal() инициализирует* собственный обработчик сигнала,
номер которого указан первым параметром вызова.
Второй параметр вызова *signal()* указывает на этот собственный обработчик.

Программа *signal_catch* демонстрирует функционирование собственных обработчиков сигналов SIGINT (отправляем нажатием Ctrl-C) и
SIGQUIT (отправляем нажатием Ctrl-\).

```
/* Программа signal_catch.cpp */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
main(void)
{
int     i;
void   signal_catcher(int);
if(signal(SIGINT, signal_catcher)==SIG_ERR){
    perror("SIGINT");
    exit(1);
    }
if(signal(SIGQUIT, signal_catcher)==SIG_ERR){
    perror("SIGQUIT");
    exit(2);
    }
for(i=0; ;++i){          /* Зацикливание */
    printf("%i\n", i);      /* Индикация счетчика */
    sleep(1);
    }
}
void
signal_catcher(int the_sig){
    signal(the_sig, signal_catcher);  /* Сброс */
    printf("\nSignal %d received.\n", the_sig);
    if(the_sig==SIGQIUT)
    exit(3);
}
```

Рассмотрим некоторые виды сигналов
и реакции системы на их возникновение:

Название	***Действие по умолчанию***	***Значение сигнала***
SIGILL	Завершить +core	Сигнал посылается ядром, если процесс попытался выполнить недопустимую инструкцию.
SIGINT	Завершить	Сигнал посылается ядром всем процессам текущей группы при нажатии клавиши прерывания (<Ctrl> или <Ctrl>+<C>)
SIGKILL	Завершить	Сигнал, при получении которого выполнение процесса завершается. Этот сигнал нельзя ни перехватить, ни игнорировать.
SIGPIPE	Завершить	Сигнал посылается при попытке записи в канал или сокет, получатель данных которого завершил выполнение (закрыл соответствующий дескриптор).
SIGPOLL	Завершить	Сигнал отправляется при наступлении определенного события для устройства, которое является опрашиваемым.
SIGPWR	Игнорировать	Сигнал генерируется при угрозе потери питания. Обычно отправляется, когда питание системы переключается на ИБП (UPS).
SIGQUIT	Завершить +core	Сигнал посылается ядром всем процессам текущей группы при нажатии <Ctrl>+<\>.
SIGSTOP	Остановить	Сигнал отправляется всем процессам текущей группы при нажатии <Ctrl>+<Z>. Получение сигнала вызывает останов выполн процесса.
SIGTERM	Завершить	Сигнал обычно представляет предупреждение, что процесс вскоре будет уничтожен, что позволяет процессу подготовиться к этому - удалить временные файлы, завершить необходимые транзакции и т. д.
SIGTTIN	Остановить	Сигнал генерируется ядром (драйвером терминала) при попытке процесса фоновой группы осуществить чтение с терминала.
SIGTTOU	Остановить	Сигнал генерируется ядром (драйвером терминала) при попытке процесса фоновой группы осуществить запись на терминал.
SIGUSR1	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессного взаимодействия.
SIGUSR2	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессного взаимодействия.

Системный вызов *signal()* организует обработку сигнала отличную от указанной по умолчанию, т.е. определяет новую *диспозицию* сигнала, в качестве которой может быть:
- пользовательская функция-обработчик;
либо одно из следующих значений:
SIG_IGN - указывает ядру, что сигнал следует игнорировать,
SIG_DFL - указывает, что при получении процессом сигнала необходимо вызвать системный обработчик, т. е. выполнить действие по умолчанию.

В случае необходимости измененная (новая) диспозиция сигнала может быть восстановлена. Это достигается благодаря тому, что системный вызов *signal()* при успешном выполнении возвращает ту диспозицию, что была до изменения.

Группы процессов и сеансы

При создании процесса ему присваивается уникальный идентификатор, возвращаемый системным вызовом *fork()* его родителю.
Дополнительно ядро назначает процессу *идентификатор группы процессов* (*process group ID*).
Группа процессов включает один или более процессов и существует, пока в системе присутствует хотя бы один процесс этой группппы.
Временной интервал, начинающийся с создания группы и заканчивающийся, когда последний процесс ее покинет, называется *временем жизни группы*. Последний процесс может либо завершить свое выполнение, либо перейти в другую группу.
Многие системные вызовы могут быть применены как к единичному процессу, так и ко всем процессам группы.
Например, системный вызов *kill()* может отправить сигнал как одному процессу, так и всем процессам указанной группы.
Точно так же функция *waitpid()* позволяет родительскому процессу ожидать завершения конкретного процесса или любого процесса группы.
Каждый процесс, помимо этого, является членом *сеанса* (*session*), являющегося набором одной нескольких групп процессов.
Понятие сеанса введено для логического объединения процессов, (а точнее, групп процессов), созданных в результате регистрации и последующей работы пользователя в системе.
Таким образом, термин "сеанс работы" в системе тесно связан с понятием сеанса, описывающего набор процессов, которые порождены пользователем за время пребывания в системе.
Все процессы в группе должны находиться в одном сеансе.
У сеанса может быть несколько групп процессов, активных одновременно, причем всегда одна из этих групп находится на переднем плане, т. е. имеет доступ к терминалу.
Другие активные группы процессов находятся в фоновом режиме. Когда фоновый процесс попытается обратиться к терминалу, он получает сигнал SIGTTIN или SIGTTOUT.

Ограничения процессов

Поскольку Linux многозадачная система, то это значит, что несколько процессов конкурируют между собой при доступе к различным ресурсам. Для "справедливого" распределения разделяемых процессами ресурсов, таких как память, дисковое пространство и т. п., каждому процессу установлен набор ограничений.
Эти ограничения не носят общесистемного характера, как, например, максимальное число процессов, а устанавливаются для каждого процесса отдельно. Для получения информации о текущих ограничениях и их изменения предназначены системные вызовы *getrlimit()* и *setrlimit()*.

```
#include <sys/time.h>
#include <sys/resource.h>
int  getlimit(int resource, struct rlimit *rlp);
int  setlimit(int resource, const struct rlimit *rlp);
```

Аргумент *resource* определяет вид ресурса, для которого надо узнать или изменить ограничения процесса.
Структура *rlimit* состоит из двух полей:
rlim_cur;
rlim_max;
определяющих, соответственно, изменяемое (*soft*) и жесткое (*hard*) ограничение.
Первое определяет *текущее* ограничение процесса на данный ресурс, а второе - максимально возможный предел потребления ресурса.

Например, изменяемое ограничение на число открытых процессом файлов может составлять 64, в то время как жесткое ограничение равно 1024. Любой процесс может изменить значение текущего ограничения вплоть до максимально возможного предела.
Жесткое ограничение может быть изменено в сторону увеличения предела потребления ресурса только процессом с привилегиями суперпользователя. Обычные процессы могут только уменьшать значение жесткого ограничения.

Обычно ограничения устанавливаются при инициализации системы и затем наследуются порожденными процессами (хотя в дальнейшем могут быть изменены).
Вообще говоря, максимально возможный предел потребления ресурса может иметь бесконечное значение.
Для этого необходимо установить значение *rlim_max* равным *RLIM_INFINITE* . В этом случае физические ограничения системы (например, объем памяти и дискового пространства) будут определять реальный предел использования того или иного ресурса.
Некоторые ограничения и связанные с ними типы ресурсов приведены в таблице (значения аргумента *resource*):

Ограничение	***Тип ресурса***	***Эффект***
RLIMIT_CORE	Максимальный размер создаваемого файла *core*, содержащего образ памяти процесса. Если предел установлен 0, *core* создаваться не будет.	После создания файла *core* запись в этот файл будет остановлена при достижении предельного размера.
RLIMIT_CPU процес-	Максимальное время использования процессора в сек.	При превышении предела су отправляется сигнал SIGXCPU
.		
RLIMIT_DATA	Максимальный размер сегмента данных процесса в байтах, т. е. максимальное значение брейк-адреса.	При достижении этого предела последующие вызовы функции *brk()* завершатся с ошибкой ENOMEM .
RLIMIT_FSIZE перехваты-	Максимальный размер файла который может создать процесс. Если значение этого	При достижении этого предела процессу отправляется сигнал SIGXFSZ. Если сигнал
	предела равно 0, то процесс может создавать файлы.	вается или игнорируется процесс- ком, последующие попытки увеличить файл закончатся с ошибкой EFBIG .
RLIMIT_NOFILE	Максимальное количество назначенных файловых дескрипторов процесса.	При достижении этого предела, последующие попытки получить новый файловый дескриптор закончатся с ошибкой EMFILE.
RLIMIT_STACK	Максимальный размер стека процесса.	При попытке расширить стек за установленный предел отправляется сигнал SIGSEGV. Если процесс перехватывает или игнорирует сигнал и не использует альтернативный стек с помощью функции *sigaltstack()* диспозиция сигнала устанавливается на действие по умолчанию перед отправкой процессу.
RLIMIT_VMEM	Максимальный размер отображаемой памяти процесса в байтах.	При достижении этого предела последующие вызовы *brk()* или *mmap()* завершатся с ошибкой ENOMEM.
RLIMIT_NPROC	Максимальное число процессов с одним реальным UID. Определяет макс число процессов, которые может запустить пользователь.	При достижении этого предела последующие вызовы *fork()* для порождения нового процесса завершатся с ошибкой EAGAIN.

RLIMIT_RSS	Максимальный размер в байтах резидентной части процесса (RSS - Resident Set Size). Определяет макс количество физической памяти, предоставляемой процессу.	Если система ощущает недостаток памяти, ядро освободит память за счет процессов, превысивших свой RSS.
RLIMIT_MEMLOCK	Максимальный объем физической памяти (физических страниц) в байтах, который процесс может заблокировать с помощью системного вызова *mlock()*.	При превышении предела сис темный вызов *mlock()* завершится с ошибкой EAGAIN.

Программа, выводящая на экран установленные ограничения для процесса:

```
/* Программа вывода на экран текущего и
   максимального пределов потребления ресурса */

#include <sys/time.h>
#include <sys/resource.h>
void disp_limit(int resource, char  *rname);
{
        struct rlimit rlm;
        getrlimit(resource, &rlm);
        printf("%-13s ", rname);
        /* Значение изменяемого ограничения */
        if (rlm.rlim_cur == RLIM_INFINITY)
                printf("infinite   ");
        else
                printf("%101d  ", rlm.rlim_cur);
        /* Значение жесткого ограничения */
        if (rlm.rlim_max == RLIM_INFINITY)
                printf("infinite  \n");
        else
                printf("%101d  ", rlm.rlim_max);
}

main()
{
        disp_limit(RLIMIT_CORE, "RLIMIT_CORE");
        disp_limit(RLIMIT_CPU, "RLIMIT_CPU");
        disp_limit(RLIMIT_DATA, "RLIMIT_DATA");
        disp_limit(RLIMIT_FSIZE, "RLIMIT_FSIZE");
        disp_limit(RLIMIT_NOFILE, "RLIMIT_NOFILE");
        disp_limit(RLIMIT_STACK, "RLIMIT_STACK");

        /* BSD */
```

```
        #ifdef RLIMIT_NPROC
                disp_limit(RLIMIT_NPROC, "RLIMIT_NPROC");
        #endif

        /* BSD */
        #ifdef RLIMIT_RSS
                disp_limit(RLIMIT_RSS, "RLIMIT_RSS");
        #endif
        /* BSD */
        #ifdef RLIMIT_MEMLOCK
                disp_limit(RLIMIT_MEMLOCK, "RLIMIT_MEMLOCK");
        #endif
        /* System V */
        #ifdef RLIMIT_VMEM
                disp_limit(RLIMIT_VMEM, "RLIMIT_VMEM");
        #endif
}
```

Для практических занятий

1. перенаправление ввода/вывода

Синтаксис перенаправления ввода/вывода при работе в *shell*:

> file	Перенаправление стандартного потока вывода в файл *file.*
>> file	Добавление в файл *file* данных из стандартного потока вывода.
< file	Получение стандартного потока ввода из файла *file.*
p1 | p2	Передача стандартного потока вывода программы *p1* в поток ввода программы *p2*.
n > file	Переключение потока вывода из файла с дескриптором *n* в файл *file.*
n >> file	То же, но записи добавляются в файл *file.*

2. листинги программ из лекции

Детально проанализировать листинги примеров программ запуска процессов и организации конвейеров из текста лекции для выполнения и модификации их на практике.

3. обработка сигналов

Выполнить программу *signal_catch* из текста лекции и программу *signal_alarm .*

Программа *signal_alarm* содержит собственный обработчик сигнала SIGALRM от системных часов.

```
/* Программа signal_alarm.cpp */
/* Иллюстрация использования SIGALRM, setjmp и longjmp */
/* для воплощения тайм-аута */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<setjmp.h>
#include<signal.h>
main(void)
{
char    buffer[100];
int     v;
while(1){
   printf("enter a string:");
   v = t_gets(buffer, 5);
   switch(v){
        case -1:        exit(1);/* Возможно EOF */
        case -2:        printf("timed out!\n");
        break;
        default:        printf("you typed %d caracters\n", v);
        }
   }
}
```

```
jmp_buf_timeout_point;
                    /* Это обработчик сигнала SIGALRM */
void timeout_handler(int sigtype)
}
longjmp(timeout_point, 1);
}
   /* This is the important bit */
int t_gets(char *s, int t)
                    /* Буфер для значения тайм-аута в секундах */
{
char *ret;
signal (SIGALRM, timeout_handler);
if(setjmp(timeout_point) != 0)
   return -2;                       /* Тайм-аут */
alarm(t);
ret = gets(s);
alarm(0);       /* Снять звуковой сигнал */
if (ret==NULL) return -1;      /* EOF */
   else return strlen(s);
}                   /* Завершение  ^D */
```