Lab. 6 “Межпроцессное взаимодействие Linux”, первая часть

(вспомогательный материал по выполнению Лаб. 6)

Contents

[Требования по выполнению Lab. 6 “Межпроцессное взаимодействие Linux”, первая часть: 3](#_Toc87876478)

[Диаграмма состояний процесса / потока Linux 3](#_Toc87876479)

[Системный вызов fork() 4](#_Toc87876480)

[Просмотр информации о ядре и управление параметрами ядра (команды управления) 5](#_Toc87876481)

[Как же нам узнать, какие модули ядра нужны, а какие можно удалить? 5](#_Toc87876482)

[«u» и «l» опции ps 6](#_Toc87876483)

[Использование top 7](#_Toc87876484)

[Системный вызов kill 7](#_Toc87876485)

[Program to create processes using fork() and check zombie state. 8](#_Toc87876486)

[Code for Program to create processes using fork() and check zombie state in C Programming 8](#_Toc87876487)

[Program to create processes using fork() and check different states i.e. zombie, orphan, etc. 9](#_Toc87876488)

[Code for Program to create processes using fork() and check orphan state in C Programming 9](#_Toc87876489)

[Unix/Linux Pipes 12](#_Toc87876490)

[Creating a pipe 12](#_Toc87876491)

[I/O with a pipe 13](#_Toc87876492)

[Fork and a pipe 13](#_Toc87876493)

[Linux IPC with Pipes 15](#_Toc87876494)

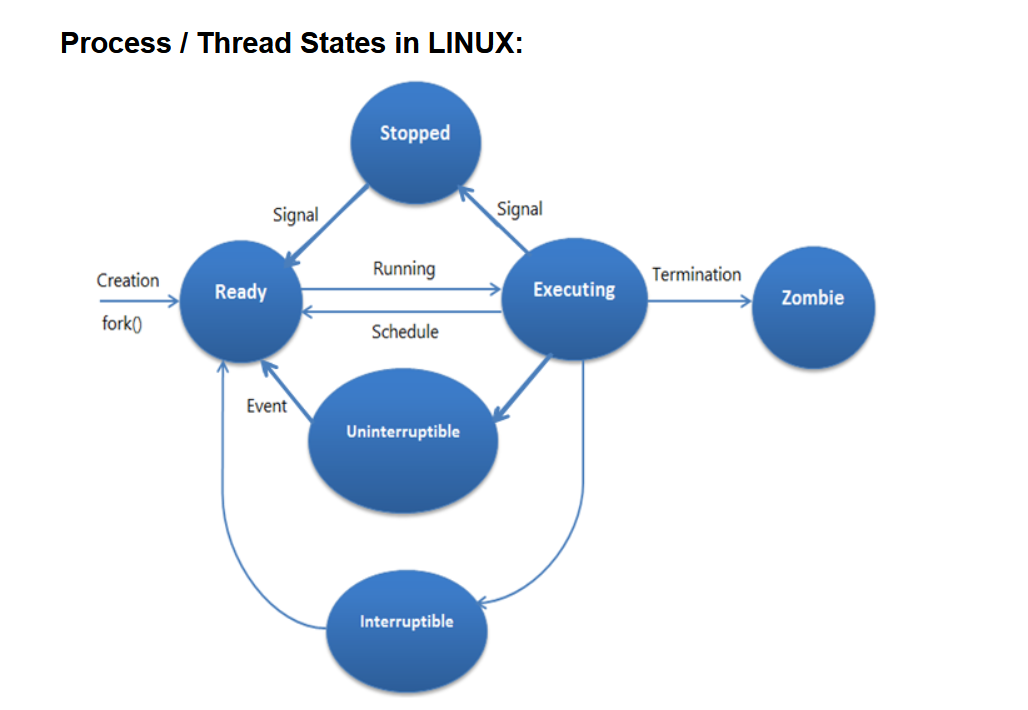
# Требования по выполнению Lab. 6 “Межпроцессное взаимодействие Linux”, первая часть:

1. Просмотр общей информации о ядре: uname, lsmod, modinfo. Состояния процессов Linux/unix. Процессы: Zombie, Сироты*.*
2. **Команды**: gcc, g++, ps (опции), ps x –forest, top, kill; fork(), getpid(), getppid(),getgid(), getuid(), sleep(), usleep(), exec().
3. **Примеры**: wait(), waitpid, pipe(), read(), write()
4. **Дополнительно (не обязательно):** ( mkfifo, fopen/fclose (open/close), shmget).
5. Включительно, inter-process communication, IPC:

* Именованные каналы, FIFO (mkfifo).
* Разделяемый сегмент памяти, shmget.
* Socket.

# Диаграмма состояний процесса / потока Linux

**В каких же состояниях может находиться процесс в Linux?**



Каждый запущенный процесс в любой момент времени находится в одном из следующих состояний (которое называют еще статусом процесса):

* **Активен (R=Running)** – процесс находится в очереди на выполнение, то есть либо выполняется в данный момент, либо ожидает выделения ему очередного кванта времени центрального процессора.
* **«Спит» (S=Sleeping)** – процесс находится в состоянии прерываемого ожидания, то есть ожидает какого-то события, сигнала или освобождения нужного ресурса.
* Находится в состоянии **непрерываемого ожидания (D=Direct)** – процесс ожидает определенного («прямого») сигнала от аппаратной части и не реагирует на другие сигналы;
* **Приостановлен (T)** – процесс находится в режиме трассировки (обычно такое состояние возникает при отладке программ).
* **«Зомби» (Z=Zombie)** – это процесс, выполнение которого завершилось, но относящиеся к нему структуры ядра по каким-то причинам не освобождены. Одной из причин их появления в системе может быть следующая ситуация. Обычно освобождение структур ядра, относящихся к процессу, выполняет процесс-родитель после получения от потомка сигнала о завершении. Но бывают случаи, когда родительский процесс завершается раньше дочернего. Процессы, не имеющие родителя, называются "**сиротами**". *"****Сироты****" автоматически* ***усыновляются*** *процессом init*, который и принимает сигналы об их завершении. Если процесс-родитель или init по каким-то причинам не может принять сигнал о завершении дочернего процесса, то процесс-потомок превращается в "зомби" и получает статус Z. Процессы-зомби не занимают процессорного времени (т. е. их выполнение прекращается), но соответствующие им структуры ядра не освобождаются. В некотором смысле это «мертвые» процессы. Уничтожение таких процессов — одна из обязанностей системного администратора. появление данных процессов говорит о том, что в системе что-то не в порядке, и скорее всего не в порядке с аппаратной частью. Не исключен вариант и кривого кода программы.

Так же, говоря о процессах в линуксе, можно выделить особый вид процессов - **демоны**. Данный вид процессов работает в фоне (подобно службам в Windows), без терминала и выполняет задачи для других процессов. Данный вид процессов на серверных системах является основным.

# Системный вызов fork()

При выполнении вызова fork():

1. Выделяется память для описателя нового процесса в таблице процессов
2. Назначается идентификатор процесса PID
3. Создается логическая копия процесса, который выполняет fork() - полное копирование содержимого виртуальной памяти родительского процесса, копирование составляющих ядерного статического и динамического контекстов процесса-предка
4. Увеличиваются счетчики открытия файлов (порожденный процесс наследует все открытые файлы родительского процесса).
5. Возвращается PID в точку возврата из системного вызова в родительском процессе и 0 - в процессе-потомке.

# Просмотр информации о ядре и управление параметрами ядра (команды управления)

**Просмотр общей информации о ядре** (версии, имени ОС, аппаратная платформа и т.п.) производится с помощью **команды** [**uname**](http://www.k-max.name/linux/osnovnye-komandy-linux-ili-shpargalka-nachinayushhego-linuksojda#uname).

Просмотреть  список подключенных модулей в данный момент возможно с помощью **команды** [**lsmod**](http://www.k-max.name/linux/osnovnye-komandy-linux-ili-shpargalka-nachinayushhego-linuksojda#lsmod):

Помимо **имени модуля**, команда lsmod показывает также размер, число пользователей модуля и имена пользователей.

**Команда modinfo** **выдает информацию об одном или нескольких модулях.**

kernel-server:/tmp/123$ /sbin/modinfo ipv6

filename:       /lib/modules/2.6.26-2-686/kernel/net/ipv6/ipv6.ko

alias:          net-pf-10

license:        GPL

description:    IPv6 protocol stack for Linux

author:         Cast of dozens

depends:

vermagic:       2.6.26-2-686 SMP mod\_unload modversions 686

### Как же нам узнать, ****какие модули ядра нужны****, а какие можно удалить?

А все просто: если счетчик **Used By** равен нулю, то модуль ядра никем и ничем не используется. Соответственно, его можно удалить.

**Удаление модуля ядра** происходит командой  ***rmmod*** module\_name.

Удаленный модуль может понадобиться в процессе работы, для загрузки модуля необходимо выполнить команду: ***insmod*** /path/to/module.ko

Интересный пример использования insmod в купе с другими командами:

[root@proxy test]# uname -r

2.6.27-ovz-smp-alt9

[root@proxy test]# insmod /lib/modules/`uname -r`/kernel/drivers/block/floppy.ko

[root@proxy test]# rmmod floppy

[root@proxy test]# modinfo -F filename floppy

/lib/modules/2.6.27-ovz-smp-alt9/kernel/drivers/block/floppy.ko

[root@proxy test]# insmod $(modinfo -F filename floppy)

[root@proxy test]# lsmod | grep floppy

floppy                 58244  0

$ **ps ax**

PID TTY STAT TIME COMMAND

1 ? S 0:04 init [3]

2 ? SW 0:11 [keventd]

3 ? SWN 0:13 [ksoftirqd\_CPU0]

4 ? SW 2:33 [kswapd]

5 ? SW 0:00 [bdflush]

Здесь приведены только первые 5 процессов, поскольку обычно список процессов очень длинный. Команда дает вам «слепок» всего, что в данный момент выполняется на машине, однако в нем много лишней информации. Если бы вы, не указали *ax*, вы бы получили список только тех процессов, которые принадлежат вам, и которые есть в управляющем терминале. Команда *ps x* покажет все процессы, даже те, которых нет в управляющем терминале. Если использовать *ps a*, то будет получен список процессов из терминалов всех пользователей.

Вы также можете просмотреть и другую информацию о каждом процессе. Опция --forest позволяет легко просмотреть иерархию процессов и даст вам представление о том, как различные процессы в системе взаимосвязаны между собой. Если один процесс запускает другой процесс, то запущенный будет называться его потомком. В выводе --forest, родители находятся слева, а потомки появляются как ветки справа:   
  
$ **ps x --forest**

PID TTY STAT TIME COMMAND

927 pts/1 S 0:00 bash

6690 pts/1 S 0:00 \\_ bash

26909 pts/1 R 0:00 \\_ ps x --forest

19930 pts/4 S 0:01 bash

25740 pts/4 S 0:04 \\_ vi processes.txt

### «u» и «l» опции ps

Опции *u* и *l* могут быть использованы в любой комбинации с опциями *a*, *x* с целью получения более подробной информации о процессах:   
  
$ **ps au**

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

agriffis 403 0.0 0.0 2484 72 tty1 S 2001 0:00 -bash

chouser 404 0.0 0.0 2508 92 tty2 S 2001 0:00 -bash

root 408 0.0 0.0 1308 248 tty6 S 2001 0:00 /sbin/agetty 3

agriffis 434 0.0 0.0 1008 4 tty1 S 2001 0:00 /bin/sh /usr/X

chouser 927 0.0 0.0 2540 96 pts/1 S 2001 0:00 bash

$ **ps al**

F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY TIME COMMAND

100 1001 403 1 9 0 2484 72 wait4 S tty1 0:00 -bash

100 1000 404 1 9 0 2508 92 wait4 S tty2 0:00 -bash

000 0 408 1 9 0 1308 248 read\_c S tty6 0:00 /sbin/ag

000 1001 434 403 9 0 1008 4 wait4 S tty1 0:00 /bin/sh

000 1000 927 652 9 0 2540 96 wait4 S pts/1 0:00 bash

### Использование top

Если вы обнаружили, что запускаете ps несколько раз подряд, пытаясь рассмотреть происходящие изменения, возможно вам стоит воспользоваться top. Программа top отображает постоянно обновляющийся список процессов, наряду с другой полезной информацией:   
  
$ **top**

10:02pm up 19 days, 6:24, 8 users, load average: 0.04, 0.05, 0.00

75 processes: 74 sleeping, 1 running, 0 zombie, 0 stopped

CPU states: 1.3% user, 2.5% system, 0.0% nice, 96.0% idle

Mem: 256020K av, 226580K used, 29440K free, 0K shrd, 3804K buff

Swap: 136544K av, 80256K used, 56288K free 101760K cached

PID USER PRI NI SIZE RSS SHARE STAT LIB %CPU %MEM TIME COMMAND

628 root 16 0 213M 31M 2304 S 0 1.9 12.5 91:43 X

26934 chouser 17 0 1272 1272 1076 R 0 1.1 0.4 0:00 top

652 chouser 11 0 12016 8840 1604 S 0 0.5 3.4 3:52 gnome-termin

641 chouser 9 0 2936 2808 1416 S 0 0.1 1.0 2:13 sawfish

$ **ps l 641**

F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY TIME COMMAND

000 1000 641 1 9 10 5876 2808 do\_sel S ? 2:14 sawfish

# Системный вызов kill

Для приостановки, завершения, запуска приостановленного процесса, а также для сообщения процессам другой информации, используются сигналы посылаемые процессам с помощью команды kill, например:

* kill-9 688 – завершение процесса с PID688 в любом случае (сигнал kill);
* kill -15 688 – завершение процесса с PID688 программно (сигнал term);
* kill -19 688 – приостановка выполнения процесса с PID688 (сигнал stop).

Другие опции и сигналы команд управления процессами можно узнать, вызвав справку по данных командам, иногда в них встречаются различия, в зависимости от дистрибутива.

# Program to create processes using fork() and check zombie state.

## Code for Program to create processes using fork() and check zombie state in C Programming

# include <stdio.h>

int main()

{

int pid;

pid=getpid();

printf("Current Process ID is : %d\n",pid);

printf("[ Forking Child Process ... ] \n");

pid=fork(); /\* This will Create Child Process and  
 Returns Child's PID \*/  
if(pid < 0)

{

/\* Process Creation Failed ... \*/

exit(-1);

}

elseif(pid==0)

{

/\* Child Process \*/

printf("Child Process Started ...\n");

printf("Child Process Completed ...\n");

}

else

{

/\* Parent Process \*/

sleep(10);

printf("Parent Process Running ... \n");

printf("I am In Zombie State ...\n");

while(1)

{

/\*   
 Infinite Loop that Keeps the  
 Process Running  
 \*/

}

}

return 0;

}

/\* Output  
  
[divyen@localhost PP-TW1]$ ./Prog01-Z &  
[1] 2320  
Current Process ID is : 2320  
[ Forking Child Process ... ]  
Child Process Started ...  
Child Process Completed ...  
[divyen@localhost PP-TW1]$ ps -l  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 500 2193 2192 0 75 0 - 1078 wait4 pts/2 00:00:00 bash  
0 S 500 2320 2193 0 75 0 - 336 schedu pts/2 00:00:00 Prog01-Z  
1 Z 500 2321 2320 0 75 0 - 0 t> pts/2 00:00:00 Prog01-Z <defunct>  
0 R 500 2322 2193 0 81 0 - 788 - pts/2 00:00:00 ps  
[divyen@localhost PP-TW1]$ Parent Process Running ...  
I am In Zombie State ...  
  
[divyen@localhost PP-TW1]$ ps -l  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 500 2193 2192 0 76 0 - 1078 wait4 pts/2 00:00:00 bash  
0 R 500 2320 2193 26 80 0 - 336 - pts/2 00:00:04 Prog01-Z  
1 Z 500 2321 2320 0 75 0 - 0 t> pts/2 00:00:00 Prog01-Z <defunct>  
0 R 500 2323 2193 5 81 0 - 787 - pts/2 00:00:00 ps  
  
  
\*/

# Program to create processes using fork() and check different states i.e. zombie, orphan, etc.

## Code for Program to create processes using fork() and check orphan state in C Programming

# include <stdio.h>

int main()

{

int pid;

pid=getpid();

printf("Current Process ID is : %d\n",pid);

printf("[ Forking Child Process ... ] \n");

pid=fork(); /\* This will Create Child Process and  
 Returns Child's PID \*/  
if(pid < 0)

{

/\* Process Creation Failed ... \*/

exit(-1);

}

elseif(pid==0)

{

/\* Child Process \*/

printf("Child Process is Sleeping ...");

sleep(5);

/\*   
 Orphan Child's Parent ID is 1  
 \*/

printf("\nOrphan Child's Parent ID : %d",getppid());

}

else

{

/\* Parent Process \*/

printf("Parent Process Completed ...");

}

return 0;

}

/\* Output  
  
[divyen@localhost PP-TW1]$ ./Prog01-O &  
[1] 2277  
Current Process ID is : 2277  
[ Forking Child Process ... ]  
Parent Process Completed ...[divyen@localhost PP-TW1]$ ps -l  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 500 2193 2192 0 75 0 - 1078 wait4 pts/2 00:00:00 bash  
1 S 500 2278 1 0 75 0 - 336 schedu pts/2 00:00:00 Prog01-O  
0 R 500 2279 2193 0 81 0 - 787 - pts/2 00:00:00 ps  
[1]+ Done ./Prog01-O  
[divyen@localhost PP-TW1]$ Child Process is Sleeping ...  
Orphan Child's Parent ID : 1  
  
\*/

Для того, чтобы убрать «зомби» **из списка запущенных процессов Linux надо завершить** родительский процесс.

Рассмотрим следующий пример:

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int main()

{

char pid{[}255{]};

fork();

fork();

fork();

sprintf(pid, "PID : %d\n",getpid());

write(STDOUT\_FILENO, pid, strlen(pid));

exit(0);

}

В этом случае будет создано семь процессов-потомков. Первый вызов fork() создает

первого потомка. Как указано выше, процесс наследует положение указателя команд

от родительского процесса. Указатель команд содержит адрес следующего

оператора программы. Это значит, что после первого вызова fork() указатель команд

и родителя, и потомка находится перед вторым вызовом fork().После второго вызова

fork() и родитель, и первый потомок производят потомков второго поколения - в

результате образуется четыре процесса. После третьего вызова fork() каждый

процесс производит своего потомка, увеличивая общее число процессов до восьми.

Так называемые процессы-зомби возникают, если потомок завершился, а

родительский процесс не вызвал wait(). Для завершения процессов используют либо

оператор возврата, либо вызов функции exit() со значением, которое нужно

возвратить операционной системе. Операционная система оставляет процесс

зарегистрированным в своей внутренней таблице данных, пока родительский

процесс не получит кода возврата потомка, либо не закончится сам. В случае

процесса-зомби его код возврата не передается родителю, и запись об этом

процессе не удаляется из таблицы процессов операционной системы. При

дальнейшей работе и появлении новых зомби таблица процессов может быть

заполнена, что приведет к невозможности создания новых процессов.

The following variant of the [Hello World](http://en.wikipedia.org/wiki/Hello_World) program "forks off" a child process that prints a message and exits. The parent process does not perform any useful work; it simply waits for the child process to finish.

#include <sys/types.h> /\* pid\_t \*/

#include <sys/wait.h> /\* waitpid \*/

#include <stdio.h> /\* printf, perror \*/

#include <stdlib.h> /\* exit \*/

#include <unistd.h> /\* \_exit, fork \*/

int main(void)

{

pid\_t pid = fork();

if (pid == -1) {

// When fork() returns -1, an error happened.

perror("fork failed");

exit(EXIT\_FAILURE);

}

else if (pid == 0) {

// When fork() returns 0, we are in the child process.

printf("Hello from the child process!\n");

\_exit(EXIT\_SUCCESS); // exit() is unreliable here, so \_exit must be used

}

else {

// When fork() returns a positive number, we are in the parent process

// and the return value is the PID of the newly created child process.

int status;

(void)waitpid(pid, &status, 0);

}

return EXIT\_SUCCESS;

}

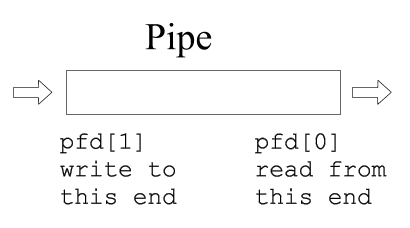
# Unix/Linux Pipes

A pipe is used for one-way communication of a stream of bytes. The command to create a pipe is ***pipe()***, which takes an array of two integers. It fills in the array with two file descriptors that can be used for low-level I/O.

### Creating a pipe

int pfd[2];

pipe(pfd);



### I/O with a pipe

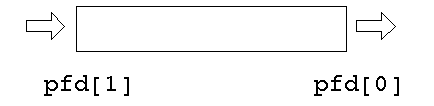
These two file descriptors can be used for block I/O

write(pfd[1], buf, size);

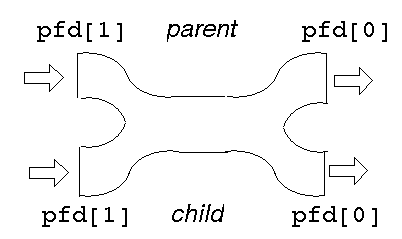
read(pfd[0], buf, SIZE);

### Fork and a pipe

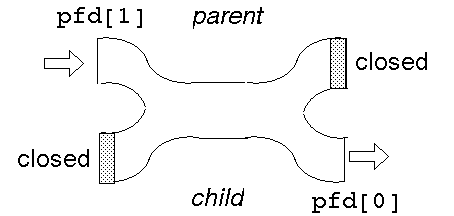
A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using ***fork()***. A pipe opened before the fork becomes shared between the two processes.   
  
Before fork



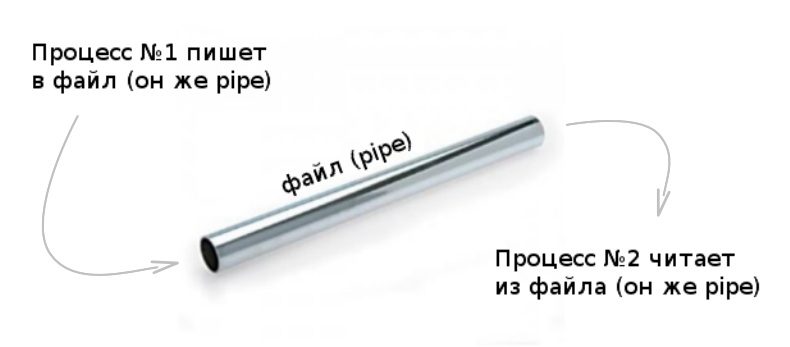
After fork



This gives *two* read ends and *two* write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.   
  
Either process can write into the pipe, and either can read from it. Which process will get what is not known.   
  
For predictable behaviour, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.



Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end.

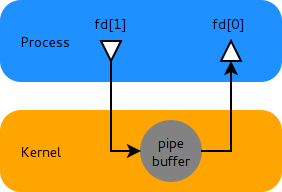


# Linux IPC with Pipes

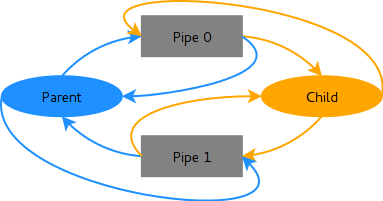
27 Jul 2012

Category: [programming](http://hzqtc.github.io/categories.html#category_programming). Tags: [ipc](http://hzqtc.github.io/tags.html#tag_ipc), [pipe](http://hzqtc.github.io/tags.html#tag_pipe).

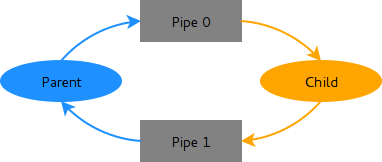
Inter-Process Communication (IPC) is a set of methods for exchanging data among multiple processes. IPC is a very common mechanism in Linux and Pipe maybe one of the most widely used IPC methods. When you type cat foo | grep bar, you create a pipe to connect stdout of cat to stdin of grep. A pipe, as its name states, can be understood as a channel with two ends. Pipe is actually implemented using a piece of kernel memory. The system call pipe always create a pipe and two associated file descriptions, fd[0] for reading from the pipe and fd[1] for writing to the pipe.



Pipe is always used with fork, because using pipe within one process is meaningless. In the example, two pipes are created.



After pipe and fork, both parent process and child process can read and write to both pipes. However, a pipe is unidirectional. If the parent process write the pipe1 and then read from pipe1, it will get the same data written before. And that’s why two pipes are created, pipe1 for data flow from parent to child and pipe2 for data flow from child to parent. Unneeded pipe descriptions must be closed.



After that, we use dup2 to connect stdin of the child to pipe1 and stdout of the child to pipe2. The pipes are transparent for the child process, it has no idea that it’s reading from a pipe and writing to another pipe. Then the parent process can use the two pipes to communicate with the child. In the example program, the child process executes rev, which reverse each line it reads from stdin and print the result to stdout. The parent process passed everything it reads from stdin to the child and redirects everything the child prints to stdout.

Pipe example:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int randomNumber(int j) {

srand(j);

return rand()%20;

}

int main( )

{

int fd[2];

int df[2];

int child1, child2;

int a,b;

int a\_rec=0 ;

int b\_rec=0 ;

char buffer1[255];

char buffer2[255];

char buffer3[255];

char buffer4[255];

pipe(fd);

pipe(df);

child1 = fork();

if (child1 == -1) {

perror("fork");

exit(1);

}

if (child1 == 0) {

close(fd[0]);

a = 0;

a = randomNumber(1);

sprintf(buffer1," %i \n",a);

printf("Child1 sends a number %s \n", buffer1);

write( fd[1], buffer1, sizeof(buffer1) );

exit(0);

}

else {

close(fd[1]);

read(fd[0], buffer2, sizeof(buffer1));

a\_rec = atoi( buffer2 );

printf("Father received from child1 a number: %d \n", a\_rec);

child2 = fork();

if (child2 == 0) {

close(df[0]);

b = randomNumber(2);

sprintf(buffer3,"%i \n",b);

printf(" Child2 number is %s \n ", buffer3);

write(df[1], buffer3, sizeof(buffer3));

exit(0);

}else {

/\* Parent process closes up output side of pipe \*/

close(df[1]);

/\* Read in a string from the pipe \*/

read(df[0], buffer4, sizeof(buffer3));

b\_rec= atoi(buffer4);

printf("Father gets from child2 a number: %d \n ", b\_rec);

}

if (a\_rec > b\_rec)

printf( "Childs1 number is the biggest\n");

else

printf("Childs2 number is the biggest\n");

}

return 0;

}

Results:

$ gcc fork2.c -o f2

$ ./f2

Child1 sends a number 3

Father received from child1 a number: 3

Child2 number is 10

Father gets from child2 a number: 10

Childs2 number is the biggest

Fork example:

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

pid\_t pid;

int a=0;

if( (pid=fork() ) == -1)

{

perror("Error at fork");

//exit(1);

}

if (pid == 0)

{ /\* child \*/

printf("Child process id=%d, with parent: id=%d\n",getpid(),getppid());

printf("Child process: after fork, variable a=%d\n", a);

a = 5;

printf("Child process: after modifying, variable a=%d\n", a);

}

else

{ /\* parent \*/

printf("Parent process id=%d, with its parent: id=%d and child: id=%d\n",getpid(),getppid(),pid);

sleep(2);

printf("Parent process: variable a=%d\n", a);

}

/\* the common code \*/

printf("Common code, executed by %s.\n", pid==0?"child":"parent");

return 0;

}

Results:

$ gcc fork1.c -o f

$ ./f

Child process id=19294, with parent: id=19293

Child process: after fork, variable a=0

Child process: after modifying, variable a=5

Common code, executed by child.

Parent process id=19293, with its parent: id=18717 and child: id=19294

Parent process: variable a=0

Common code, executed by parent.