Создание процессов

Системный вызов **fork()** используется для создания новых процессов linux путём копирования существующего процесса.

*#include <stdio.h>*

*#include <sys/types.h>*

*#include <unistd.h>*

*int main()*

*{*

*// make two process which run same*

*// program after this instruction*

*fork();*

*printf("Hello world!\n");*

*return 0;*

*}*

Выводит «Hello world! Hello world!»

*#include <stdio.h>*

*#include <sys/types.h>*

*int main()*

*{*

*fork();*

*fork();*

*fork();*

*printf("hello\n");*

*return 0;*

*}*

Выводит «hello hello hello hello hello hello hello hello»

Процесс, образованный с помощью fork («ребёнок»), имеет собственный уникальный PID, не совпадающий с ID любой существующей группы процессов.

PPID ребёнка – такой же, как PID процесса-родителя.

Ребёнок не наследует memory locks и semaphore adjustments родителя.

Ребёнок не наследует асинхронные I/O операции и контексты от родителя.

В случае успеха выполнения fork() PID ребёнка возвращается родителю, а ребёнку возвращается 0. В случае неудачи родителю возвращается -1, ребёнок не создан, устанавливается errno.

Когда вызывается **exec()**, программа, определённая в параметре exec(), заменяет весь процесс (замена одного процесса другим).

*//EXEC.c*

*#include<stdio.h>*

*#include<unistd.h>*

*int main()*

*{*

*int i;*

*printf("I am EXEC.c called by execvp() ");*

*printf("\n");*

*return 0;*

*}*

//execDemo.c

*#include<stdio.h>*

*#include<stdlib.h>*

*#include<unistd.h>*

*int main()*

*{*

*//A null terminated array of character*

*//pointers*

*char \*args[]={"./EXEC",NULL};*

*execvp(args[0],args);*

*/\*All statements are ignored after execvp() call as this whole*

*process(execDemo.c) is replaced by another process (EXEC.c)*

*\*/*

*printf("Ending-----");*

*return 0;*

*}*

Выводит «I AM EXEC.c called by execvp()»

exec() – это семейство функций:

* int execl(const char \*path, const char \*arg, ...);
* int execv(const char \*path, char \*const argv[]);
* int execlp(const char \*file, const char \*arg, ...);
* int execvp(const char \*file, char \*const argv[]);
* int execle(const char \*path, const char \*arg, ..., char \* const envp[]);
* int execvpe(const char \*file, char \*const argv[], char \*const envp[]);

Литература:

* https://man7.org/linux/man-pages/man3/exec.3.html
* https://man7.org/linux/man-pages/man2/fork.2.html
* <https://www.geeksforgeeks.org/fork-system-call/>

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, а 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.

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. 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.

01 SIGHUP Освобождение линии (hangup).  
 02 SIGINT Прерывание (interrupt).   
03 SIGQUIT Выход (quit).   
04 SIGILL Некорректная команда (illegal instruction). Не переустанавливается при перехвате.   
05 SIGTRAP Трассировочное прерывание (trace trap). Не переустанавливается при перехвате.   
06 SIGIOT или SIGABRT Машинная команда IOT.   
07 SIGEMT Машинная команда EMT.   
08 SIGFPE Исключительная ситуация при выполнении операции с вещественными числами (floating-point exception)   
09 SIGKILL Уничтожение процесса (kill). Не перехватывается и не игнорируется.   
10 SIGBUS Ошибка шины (bus error).   
11 SIGSEGV Некорректное обращение к сегменту памяти (segmentation violation).   
12 SIGSYS Некорректный параметр системного вызова (bad argument to system call).   
13 SIGPIPE Запись в канал, из которого некому читать (write on a pipe with no one to read it).   
14 SIGALRM Будильник   
15 SIGTERM Программный сигнал завершения   
16 SIGUSR1 Определяемый пользователем сигнал 1   
17 SIGUSR2 Определяемый пользователем сигнал 2   
18 SIGCLD Завершение порожденного процесса (death of a child).   
19 SIGPWR Ошибка питания 22 Регистрация выборочного события

*#include<stdio.h>*

*#include<signal.h>*

*// Handler for SIGINT, caused by*

*// Ctrl-C at keyboard*

*void handle\_sigint(int sig)*

*{*

*printf("Caught signal %d\n", sig);*

*}*

*int main()*

*{*

*signal(SIGINT, handle\_sigint);*

*while (1) ;*

*return 0;*

*}*

Вывод:

^CCaught signal 2 // when user presses ctrl-c ^CCaught signal 2

Литература:

* http://www.opennet.ru/docs/RUS/lnx\_process/process2.html
* https://man7.org/linux/man-pages/man2/signal.2.html
* https://man7.org/linux/man-pages/man2/kill.2.html