**Лабораторная работа № 4**

**ПРОЦЕССЫ В LINUX**

**Часть 2**

# ФУНКЦИИ ДЛЯ СОЗДАНИЯ ПРОЦЕССОВ

## **1. Теоретическая часть**

**Идентификаторы процессов.** Работая с идентификаторами процессов в программах, написанных на языках C и C++, следует объявить соответствующие переменные как имеющие тип **pid\_t** (этот тип определяется в файле **<sys/types.h>)**. Программа может узнать идентификатор своего собственного процесса с помощью системного вызова **getpid()**, а идентификатор своего родительского процесса с помощью системного вызова **getppid()**.

**Пример 1:**

#include<stdio.h>

#include<unistd.h>

int main()

{

printf("Номер процесса: %d\n", (int) getpid() );

printf("Номер родительского процесса: %d\n", (int) getppid());

return 0;

}

Обратите внимание на важную особенность: при каждом вызове программа сообщает о разных идентификаторах, поскольку всякий раз запускается новый процесс. Тем не менее, если программа вызывается из одного и того же интерпретатора команд, то родительский идентификатор оказывается одинаковым.

**Создание процессов**

Существуют два способа создания процессов. Первый из них относительно прост, применяется редко, поскольку неэффективен и связан со значительным риском для безопасности системы. Второй способ сложнее, но избавлен от недостатков первого. Первый способ основан на применении функции **system()**, а второй на основе применения функций **fork()** и **exec()**.

**Функция system()**

Функция **system()** определена в стандартной библиотеке языка C и позволяет вызвать из программы системную команду, как если бы она была набрана в командной строке.

По сути, эта функция запускает стандартный интерпретатор и передает ему команду на выполнение.

**Пример 2:**

#include<stdlib.h>

#include<stdio.h>

int main()

{

int return\_value;

return\_value = system("ls -l /");

printf("Возвращаемое значение: %d\n", return\_value);

return return\_value;

}

Функция **system()** возвращает код завершения указной команды. Если интерпретатор не может быть запущен, возвращается значения 127, а в случае возникновения других ошибок (-1).

Поскольку функция **system()** запускает интерпретатор команд, она подвержена всем тем ограничениям безопасности, что и командный интерпретатор.

В большинстве Unix-системах программа /bin/sh представляет собой символическую ссылку на другой интерпретатор. В Linux это в основном bash.

Вызов из функции **system()** программы с привилегиями пользователя root также может иметь неодинаковые последствия в разных системах. Таким образом, лучше создавать процессы с помощью функций **fork()** и **exec()**.

**Функции fork() и exec()**

В DOS и Windows API имеется семейство функций **spawn()**. Они принимают в качестве аргумента имя программы, создают новый экземпляр ее процесса и запускают его.

В Linux нет такой функции, которая выполнила бы все это за одни заход. Вместо этого имеются функция **fork()**, создающая дочерний процесс, который является точной копией родительского процесса, и семейство функций **exec()**, заставляющих требуемый процесс перестать быть вторым экземпляром одной программы и превратиться в экземпляр другой программы.

Чтобы создать новый процесс, нужно сначала с помощью функции **fork()** создать копию текущего процесса, а затем с помощью функции **exec()** преобразовать одну из копий в экземпляр запускаемой программы.

**Вызов функции fork()**

Вызывая функцию **fork()**, программа создает свой дубликат, называемый дочерним процессом. Родительский процесс продолжает выполнять программу с той точки, где была вызвана функция **fork()**. То же самое делает и дочерний процесс.

Процессы отличаются своими идентификаторами. Таким образом, программа может вызывать функцию **getpid()** и узнать где именно она находится.

Но сама функция **fork()** реализует другой способ: она возвращает разные значения в родительском и дочернем процессах. В родительском процессе функция **fork()** равна идентификатору своего потомка, а в дочернем процессе она равна 0. Рассмотрим данную ситуацию на примере, учтите, что первая часть инструкции **if** выполняется только в родительском процессе, тогда как ветвь **else** – только в дочернем.

**Пример 3:**

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main()

{

pid\_t child\_pid;

printf("ID процесса основной программы: %d\n", (int) getpid() );

child\_pid = fork();

if (child\_pid)

{

printf("Это родительский процесс, с ID %d\n", (int) getpid() );

printf("Дочерний процесс, с ID %d\n", (int) child\_pid );

}

else

printf("Дочерний процесс с ID %d\n", (int) getpid() );

return 0;

}

**Семейство функций exec()**

Функции семейства **exec()** заменяют программу, выполняющуюся в текущем процессе, другой программой. Когда программа вызывает функцию **exec()**, ее выполнение немедленно прекращается и начинает работу новая программа.

Функции, в название которых присутствует суффикс 'p' (**execvp()** и **execlp()**), принимают в качестве аргумента имя программы и ищут эту программу в каталогах, определяемых переменной среды PATH. Всем остальным функциям нужно передавать полное путевое имя программы.

Функции, в названии которых присутствует суффикс 'v' (**execv()**, **execvp()**, **execve()**), принимают список аргументов программы в виде массива строковых указателей, оканчивающегося NULL-указателем. Функции с суффиксом 'l' (**excevl()**, **execlp()**, **execlve()**), принимают список аргументов переменного размера.

Функции, в названии которых присутствует суффикс 'e' (**execve()**, **execle()**), в качестве дополнительного аргумента принимают массив переменных среды. Этот массив содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид «Переменная = значение».

Поскольку функция **exec()** заменяет одну программу другой, она никогда не возвращает значение – только если вызов программы оказался невозможен в случае ошибки.

Список аргументов, передаваемых программе, аналогичен аргументам командной строки, указываемым при запуске программы в интерактивном режиме. Их тоже можно получить с помощью параметров **argc** и **argv** функции **main()**. Когда программу запускает интерпретатор команд, первый элемент массива **argv** будет содержать имя программы, а далее будут находиться переданные программе аргументы. Аналогичным образом следует поступить, формируя список аргументов для функции **exec()**.

**Совместное использование функций fork() и exec()**

Стандартная методика запуска одной программы из другой такова: сначала с помощью функции **fork()** создается дочерний процесс, затем в нем вызывается функция **exec()**.

Это позволяет главной программе продолжать выполнение в родительском процессе. В качестве примера напишем программу, которая отображает корневой каталог.

**Пример 4:**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include <sys/wait.h>

#include<unistd.h>

int spawn(char\* program, char\*\* arg\_list)

{

pid\_t child\_pid;

child\_pid = fork();

if(child\_pid)

return child\_pid;

else

{

execvp(program, arg\_list);

fprintf(stderr, "an error in execvp\n");

abort();

}

}

int main()

{

int child\_status;

char\* arg\_list[] = {"ls","-l","/",NULL};

spawn("ls", arg\_list);

wait(&child\_status);

printf("done\n");

return 0;

}

**Системные вызовы wait()**

Самая простая функция в семействе называется **wait()**. Она блокирует вызывающий процесс до тех пор, пока один из его дочерних процессов не завершиться (или не произойдет ошибка).

Пример использования данной функции приведен выше.

Функция **waitpid()** позволяет дождаться завершения конкретного дочернего процесса.

Функция **wait3()** возвращает информацию о статистике использования центрального процессора завершившемся дочерним процессом.

Функция **wait4()** позволяет задать дополнительную информацию о том, каких процессов следует дождаться.

## **2. Задание на лабораторную работу**

С помощью компилятора С создать и выполнить программы, исходный текст которых приведен в примерах 1 – 4.

## **Контрольные вопросы**

1. Что является атрибутами процесса?
2. Как организуется взаимодействие процессов?
3. Каким образом программные средства Linux позволяют динамически порождать процессы?
4. Какие существуют формы системного вызова exec()?