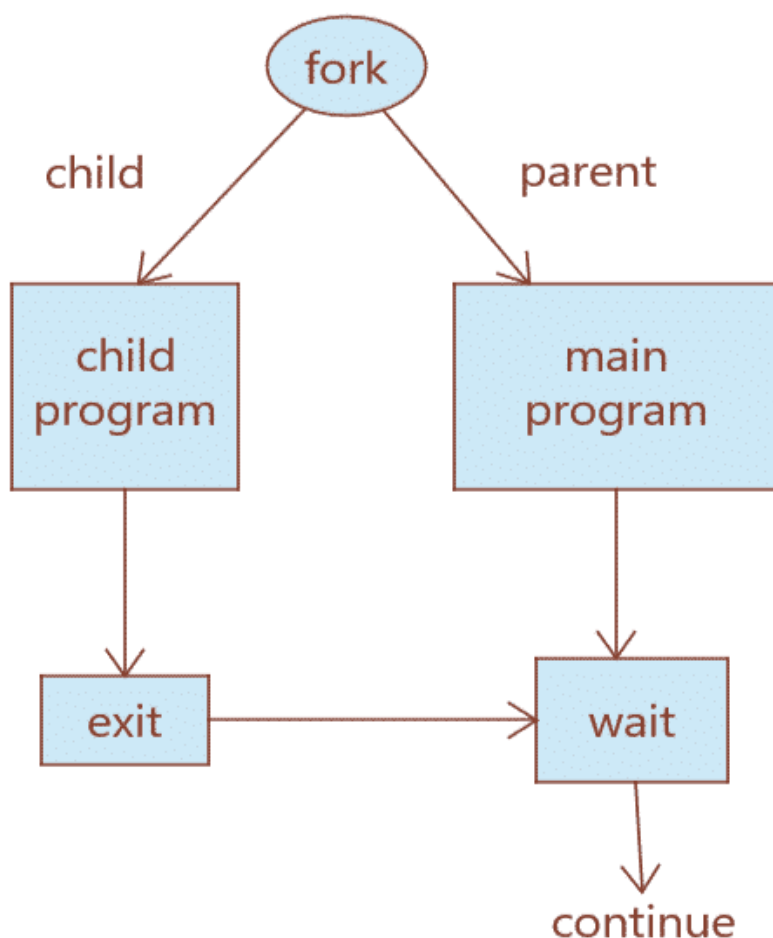


Программирование на С

Системный вызов вилки в С

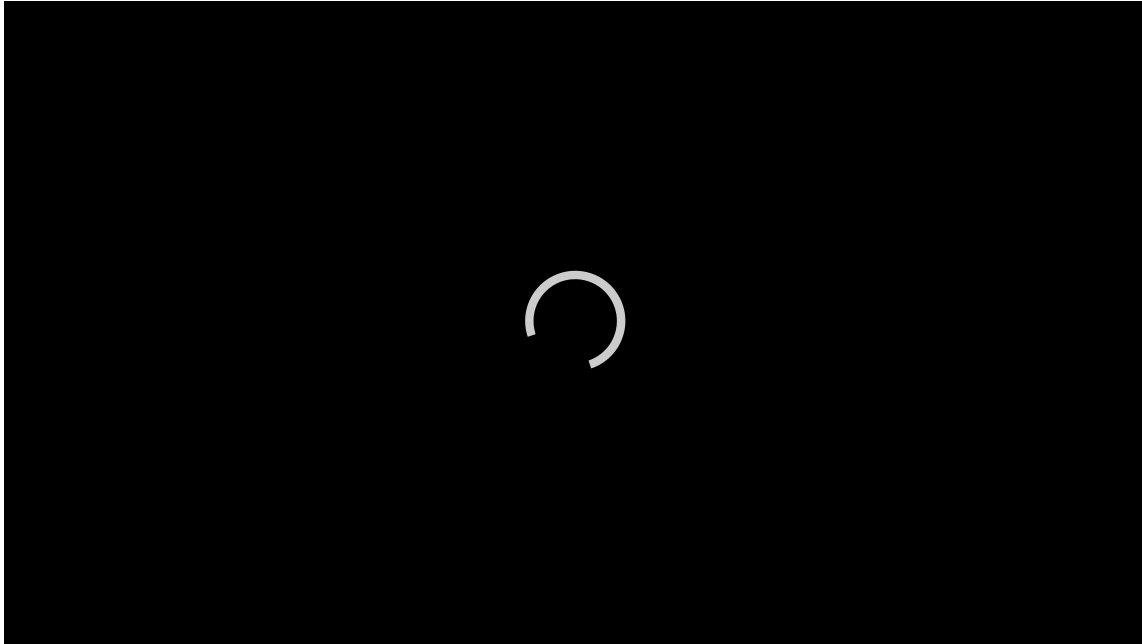
3 года назад • Шахриар Шовон

системный вызов `fork()` используется для создания дочерних процессов в программе на языке Си. `fork()` используется там, где в вашем приложении требуется параллельная обработка. Системная функция `fork()` определена в заголовках **`sys/types.h`** и **`unistd.h`**. В программе, где вы используете `fork`, вы также должны использовать системный вызов `wait()`. Системный вызов `wait()` используется для ожидания завершения дочернего процесса в родительском процессе. Для завершения дочернего процесса в дочернем процессе используется системный вызов `exit()`. Функция `wait()` определена в заголовке **`sys/wait.h`**, а функция `exit()` определена в заголовке **`stdlib.h`**.

Рис. 1. Базовый рабочий процесс `fork()`

В этой статье я покажу вам, как использовать системный вызов `fork()` для создания дочерних процессов в С. Итак, давайте начнем.

МОИ ПОСЛЕДНИЕ ВИДЕО



синтаксис `fork()` и возвращаемое значение:

Синтаксис системной функции `fork()` выглядит следующим образом:

```
pid_t fork(void);
```

Системная функция `fork()` не принимает никаких аргументов. Он возвращает целое число типа `pid_t`.

При успешном выполнении `fork()` возвращает PID дочернего процесса, который больше 0. Внутри дочернего процесса возвращаемое значение равно 0. Если `fork()` терпит неудачу, он возвращает -1 .

Простой пример `fork()`:

Ниже приведен простой пример `fork()`:

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

int main(void) {
    pid_t pid = fork();

    if(pid == 0) {
        printf("Child => PPID: %d PID: %d\n", getppid(), getpid());
        exit(EXIT_SUCCESS);
    }
    else if(pid > 0) {
        printf("Parent => PID: %d\n", getpid());
        printf("Ожидание завершения дочернего процесса.\n");
        wait(NULL);
        printf("Дочерний процесс завершен.\n");
    }
}
```

```

    }
    else {
        printf("Невозможно создать дочерний процесс.\n");
    }
}

return EXIT_SUCCESS;
}

```

Здесь я использовал `fork()` для создания дочернего процесса из основного / родительского процесса. Затем я напечатал PID (идентификатор процесса) и PPID (идентификатор родительского процесса) из дочернего и родительского процессов. В родительском процессе `wait(NULL)` используется для ожидания завершения дочернего процесса. В дочернем процессе `exit()` используется для завершения дочернего процесса. Как вы можете видеть, PID родительского процесса является PPID дочернего процесса. Таким образом, дочерний процесс **24738** принадлежит родительскому процессу **24731**.

```

01_fork.c - system-programming - Visual Studio Code
C 01_fork.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 int main(void) {
8     pid_t pid = fork();
9
10    if(pid == 0) {
11        printf("Child => PPID: %d PID: %d\n", getppid(), getpid());
12        exit(EXIT_SUCCESS);
13    } else if(pid > 0) {
14        printf("Parent => PID: %d\n", getpid());
15        printf("Waiting for child process to finish.\n");
16        wait(NULL);
17        printf("Child process finished.\n");
18    } else {
19        printf("Unable to create child process.\n");
20    }
21
22    return EXIT_SUCCESS;
23 }
24
PROBLEMS 27 OUTPUT DEBUG CONSOLE TERMINAL
Parent => PID: 24731
Waiting for child process to finish.
Child => PPID: 24731 PID: 24738
Child process finished.
Terminal will be reused by tasks, press any key to close it.

```

Вы также можете использовать функции, чтобы сделать вашу программу более модульной. Здесь я использовал функции `processTask()` и `parentTask()` для дочернего и родительского процессов соответственно. Вот как на самом деле используется `fork()`.

```

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

void childTask() {
    printf("Hello World\n");
}

void parentTask() {
    printf("Main task.\n");
}

int main(void) {
    pid_t pid = fork();

    if(pid == 0) {
        childTask();
    }
}

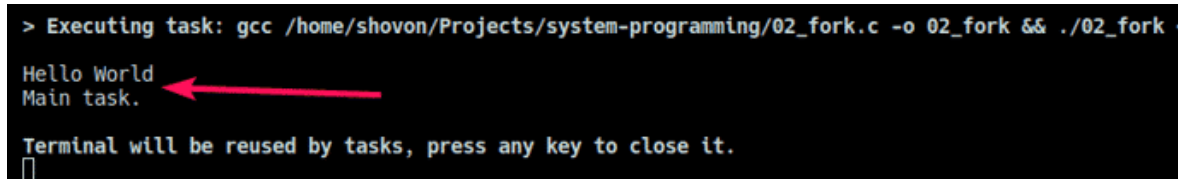
```

```

    exit(EXIT_SUCCESS);
}
else if(pid > 0) {
wait(NULL);
parentTask();
}
else {
    printf("Невозможно создать дочерний процесс");
}
}
return EXIT_SUCCESS;
}

```

Вывод вышеприведенной программы:



```

> Executing task: gcc /home/shovon/Projects/system-programming/02_fork.c -o 02_fork && ./02_fork
Hello World
Main task.
Terminal will be reused by tasks, press any key to close it.

```

Запуск нескольких дочерних процессов с помощью `fork()` и цикла:

Вы также можете использовать цикл для создания столько дочерних процессов, сколько вам нужно. В приведенном ниже примере я создал 5 дочерних процессов, используя цикл `for`. Я также напечатал PID и PPID из дочерних процессов.

```

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

int main(void) {
    for(int i = 1; i <= 5; i++) {
        pid_t pid = fork();

        if(pid == 0) {
            printf("Дочерний процесс => PPID=%d, PID=%d\n", getppid(), getpid());
            exit(0);
        }
        else {
            printf("Родительский процесс =>PID=%d\n", getpid());
            printf("Ожидание завершения дочерних процессов ...\n");
        }
        wait(NULL);
        printf("завершение дочернего процесса.\n");
    }

    return EXIT_SUCCESS;
}

```

Как видите, идентификатор родительского процесса одинаков во всех дочерних процессах. Итак, все они принадлежат одному и тому же родителю. Они также выполняются линейно. Один за другим. Управление дочерними процессами - сложная задача. Если вы узнаете больше о системном программировании Linux и о том, как это работает, вы сможете контролировать поток этих процессов так, как вам нравится.

```

> Executing task: gcc /home/shovon/Projects/system-programming/03_fork.c -o 03_fork && ./03_fork <

Parent process => PID=28751
Waiting for child processes to finish...
Child process => PPID=28751, PID=28758
child process finished.
Parent process => PID=28751
Waiting for child processes to finish...
Child process => PPID=28751, PID=28759
child process finished.
Parent process => PID=28751
Waiting for child processes to finish...
Child process => PPID=28751, PID=28760
child process finished.
Parent process => PID=28751
Waiting for child processes to finish...
Child process => PPID=28751, PID=28761
child process finished.
Parent process => PID=28751
Waiting for child processes to finish...
Child process => PPID=28751, PID=28762
child process finished.

Terminal will be reused by tasks, press any key to close it.

```

Пример из реальной жизни:

Различные сложные математические вычисления, такие как md5, sha256 и т.д. Генерация хэша требует большой вычислительной мощности. Вместо того, чтобы вычислять такие вещи в том же процессе, что и основная программа, вы можете просто вычислить хэш в дочернем процессе и вернуть хэш в основной процесс.

В следующем примере я сгенерировал 4-значный PIN-код в дочернем процессе и отправил его родительскому процессу, основной программе. Затем я напечатал оттуда PIN-код.

```

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

int getPIN() {
    // используйте PPID и PID в качестве начального
    srand(getpid() + getppid());
    int secret = 1000 + rand() % 9000;
    return secret;
}

int main(void) {
    int fd[2];
    pipe(fd);
    pid_t pid = fork();

    if(pid > 0) {
        close(0);
        close(fd[1]);
        dup(fd[0]);

        int secretNumber;
        size_t readBytes = read(fd[0], &secretNumber, sizeof(secretNumber));

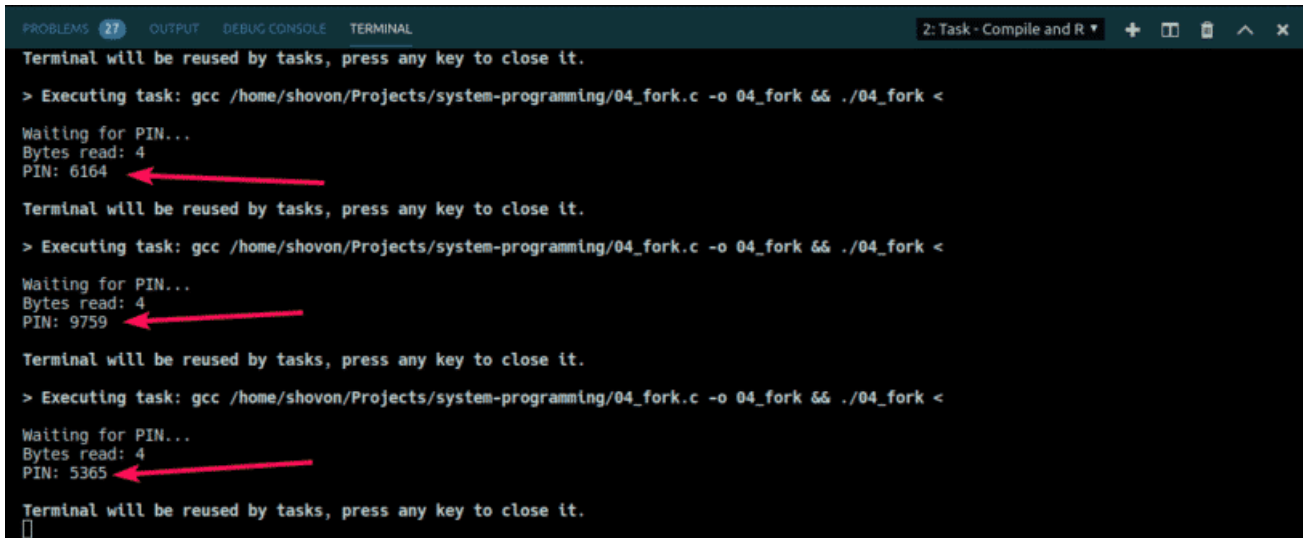
        printf("Ожидание PIN-кода ...\n");
        wait(NULL);
        printf("Чтение байтов: %ld\n", readBytes);
        printf("PIN: %d\n", secretNumber);
    }
    else if(pid == 0) {
        close(1);
        close(fd[0]);
        dup(fd[1]);

        int secret = getPIN();
        write(fd[1], &secret, sizeof(secret));
        exit(EXIT_SUCCESS);
    }
}

```

```
} return EXIT_SUCCESS;  
}
```

Как видите, каждый раз, когда я запускаю программу, я получаю другой 4-значный PIN-код.



```
PROBLEMS 27 OUTPUT DEBUG CONSOLE TERMINAL 2: Task - Compile and R + [ ] [ ] ^ x  
Terminal will be reused by tasks, press any key to close it.  
> Executing task: gcc /home/shovon/Projects/system-programming/04_fork.c -o 04_fork && ./04_fork <  
Waiting for PIN...  
Bytes read: 4  
PIN: 6164  
Terminal will be reused by tasks, press any key to close it.  
> Executing task: gcc /home/shovon/Projects/system-programming/04_fork.c -o 04_fork && ./04_fork <  
Waiting for PIN...  
Bytes read: 4  
PIN: 9759  
Terminal will be reused by tasks, press any key to close it.  
> Executing task: gcc /home/shovon/Projects/system-programming/04_fork.c -o 04_fork && ./04_fork <  
Waiting for PIN...  
Bytes read: 4  
PIN: 5365  
Terminal will be reused by tasks, press any key to close it.  
[ ]
```

Итак, вот как вы используете системный вызов `fork()` в Linux. Спасибо, что прочитали эту статью.

ОБ АВТОРЕ



Шахриар Шовон

Фрилансер и системный администратор Linux. Также любит разработку веб-API с Node.js и JavaScript. Я родился в Бангладеш. В настоящее время я изучаю электронику и коммуникационную инженерию в Университете инженерии и технологий Кхулны (KUET), одном из требовательных государственных инженерных университетов Бангладеш.

[Посмотреть все сообщения](#)

**СВЯЗАННЫЕ СООБЩЕНИЯ ПОДСКАЗОК
LINUX**

C С верхнего на нижний регистр
Как перевернуть бит в C
Как перевернуть массив в C
Преобразование строки в целое число в C
Как отлаживать ошибки сегментации в C?
Strchr Method в C
Статические функции в компьютерном языке Си

Linux Hint LLC, editor@linuxhint.com
1309 S Mary Ave Suite 210, Sunnyvale, CA
94087

ЭЛИТНЫЙ ИЗДАТЕЛЬ CAFEMEDIA