OS Lab 9

Status	approved
checkbox	☑
class	OS
due date	@Mar 24, 2021

Task

> 9. Создание двух процессов
>
> Напишите программу, которая создает подпроцесс. Этот подпроцесс должен исполнить `cat(1)` длинного файла. Родитель должен вызвать `printf(3)` и распечатать какой-либо текст. После выполнения первой части задания модифицируйте программу так, чтобы последняя строка, распечатанная родителем, выводилась после завершения порожденного процесса. Используйте `wait(2)`, `waitid(2)` или `waitpid(3)`.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>

int main (int argc, char **argv){
    if (argc < 2){
        printf("not enough arguments\n");
        return 0;
    }

    int status;
    pid_t newPid, ret;

    if ((newPid = fork()) == -1) {
        perror("fork failed: ");
        return 1;
    }

    if (newPid == 0){
        printf("I'm a child process with pid: %d\n", newPid);
        if ( execl("/bin/cat", "cat", argv[1], NULL) == -1){
            perror("execl failed: ");
            return 1;
        }
    }

    printf("I'm a parent process waiting for my child to die: %d\n", newPid);
    ret = wait(&status);

    if (ret == -1){
        perror("wait(2) error: ");
        return 1;
    }

    printf("My child died. Now it's time for me to pass away\n");

    return 0;
}
```

Notes

Что такое процесс?

Создание процесса - fork(2)

В юниксе, есть первый особенный процесс init, от которого форкаются все процессы в системе.

💡 **Процесс** представляет собой исполняющуюся программу вместе с необходимым ей окружением. Окружение процесса состоит из:

. информации о процессе, содержащейся в различных системных структурах данных

. содержимого регистров процессора (контекста процесса)

. пользовательского стека процесса и системного стека, используемого при обработке системных вызовов

. пользовательской области, которая содержит информацию об открытых файлах, текущей директории, обработке сигналов и т.д.

Wait(3C) vs Zombies

💡 Системный вызов `fork(2)` создаёт новый процесс, исполняющий копию исходного процесса. В основном, новый процесс (порождённый или дочерний) идентичен исходному (родителю). В описании `fork(2)` перечислены атрибуты, которые порождённый процесс наследует от родителя, и различия между ними.

Дочерний процесс наследует все отображённые на память файлы и вообще все сегменты адресного пространства, все открытые файлы, идентификаторы группы процессов и сессии, реальный и эффективный идентификаторы пользователя и группы, ограничения `rlimit`, текущий каталог, а также ряд других параметров.

Дочерний процесс НЕ наследует: идентификатор процесса, идентификатор родительского процесса, а также захваченные участки файлов. В большинстве Unix-систем, дочерний процесс не наследует нити исполнения, кроме той, из которой был вызван `fork(2)`. После возврата из `fork(2)`, оба процесса продолжают исполнение с той точки, где `fork(2)` был вызван. Процессы могут узнать, кто из них является родителем, а кто порождённым, на основании значения, возвращённого `fork(2)`.

Родительский процесс получает идентификатор порождённого процесса, положительное число. Порождённый процесс получает нулевое значение. Как правило, за `fork(2)` следует оператор `if` или `switch`, который определяет, какой код исполнять для родительского и какой для порождённого процесса.

Системный вызов `fork(2)` может завершиться **неудачей**, если вы пытаетесь превысить разрешённое количество процессов для каждого пользователя или общее количество процессов в системе. Эти два ограничения устанавливаются при конфигурации операционной системы. Если `fork(2)` завершается неуспехом, он возвращает значение `-1`. Рекомендуется проверять код возврата этого и остальных системные вызовы на предмет неудачного завершения.

В общем случае никогда нельзя сказать точно, какой из двух процессов первым получит управление после вызова функции `fork` — дочерний или родительский. Это во многом зависит от алгоритма планирования, используемого ядром. При необходимости

💡 **Можно делать только на своих потомков!**

Процесс может синхронизоваться с завершением порожденного процесса с помощью системного вызова `wait(2)`. Если вызывается `wait(2)`, а у процесса нет ни одного незавершенного подпроцесса, `wait(2)` немедленно возвращает `-1` и устанавливает `errno` равной `ECHILD`.

Иначе вызывающий процесс:
1. засыпает, если существует незавершившийся подпроцесс.
2. возвращает управление немедленно, если существует подпроцесс, который уже завершился, но к нему не применялся `wait(2)`. В обоих вышеперечисленных случаях, wait(2) возвращает идентификатор завершившегося подпроцесса

Функция `wait` приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "***зомби***"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Зомби это ху?

После завершения по `exit(2)` или по сигналу, процесс переходит в состояние, известное как «зомби». В этом состоянии процесс не исполняется, не имеет пользовательской области, адресного пространства и открытых файлов и не использует большинство других системных ресурсов. Однако «зомби» занимает запись в таблице процессов и сохраняет идентификатор процесса и идентификатор родительского процесса. Эта запись используется для хранения слова состояния процесса, в котором хранится код завершения процесса (параметр `exit(2)`), если процесс завершился по `exit(2)` или номер сигнала, если процесс завершился по сигналу.

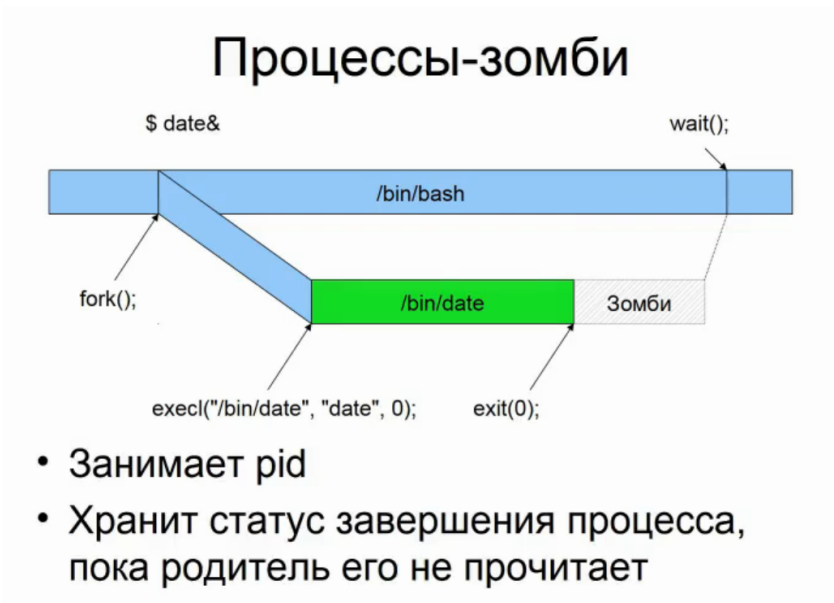
На самом деле, **главным назначением «зомби» является защита идентификатора процесса (pid) от переиспользования.** Дело в том, что родительские процессы идентифицируют

синхронизировать работу родительского и дочернего процессов можно воспользоваться каким-либо механизмом взаимодействий.

Функция `fork` являет собой яркий пример потенциального источника проблем, связанных с гонкой за ресурсами, если логика выполнения программы явно или неявно зависит от того, кто первым получит управление — родительский процесс или дочерний. В общем случае это невозможно предсказать заранее, но даже если бы мы знали наверняка, какой процесс первым получит управление, все равно дальнейшая работа процесса зависит от степени нагрузки на систему и алгоритма планирования, заложенного в ядре.

своих потомков на основе их pid, а ядро может использовать свободные pid для вновь создаваемых процессов. Поэтому, если бы не существовало записей-«зомби», была бы возможна ситуация, когда потомок с pid=21285 завершается, а родитель, не получив код возврата этого потомка, создает новый подпроцесс и система выделяет ему тот же pid. После этого родитель уже не сможет объяснить системе, какой из потомков с pid-21285 его интересует, и не сможет понять, к какому из потомков относится полученное слово состояния.

Также, если «зомби» является последним из группы процессов или сессии, то соответствующая группа процессов или сессия продолжают существовать, пока существует зомби.



If **wait()** returns because the status of a child process is available, it returns the process **ID** of the child process. If the calling process specified a non-zero value for stat_loc, the status of the child process is stored in the location pointed to by stat_loc. That status can be evaluated with the macros described on the wait.h(3HEAD) manual page.

RETURN VALUES

When **wait()** returns due to a terminated child process, the process **ID** of the child is returned to the calling process. Otherwise, **-1** is returned and **errno** is set to indicate the error.

ERRORS

The **wait()** function will fail if:

ECHILD
The calling process has no existing unwaited-for child processes.

EINTR
The function was interrupted by a signal.

Когда процесс ожидает получения слова состояния своих подпроцессов
wait(2) или waitpid(3C), то это слово может быть проанализировано
определенных в <sys/wait.h>. Эти макросы обсуждаются на странице

WIFEXITED(stat) Ненулевое значение, если это слово сост
подпроцесса, завершившегося по exit(2).

WEXITSTATUS(stat) Если значение WIFEXITED(stat) ненулевое, этот
код завершения, переданный подпроцессом вызову exit(2), или возвращенный его
функцией main(), иначе код возврата не определен.

💡 КАК ВЕДЁТ СЕБЯ ЗАМАПАННАЯ ПАМЯТЬ?

Если сегмент отображался в режиме `MAP_SHARED`, родительский и порождённый процессы используют одну и ту же физическую память; если такой сегмент доступен для модификации, то родитель и потомок будут видеть вносимые в эту память изменения. Большинство разделяемых сегментов в Unix-системах — это сегменты кода, недоступные для модификации.

Если же сегмент отображался как `MAP_PRIVATE` или `MAP_ANON` (по умолчанию, сегменты данных и стека отображаются именно так), процессы получают собственные копии соответствующих страниц. В действительности, копирование происходит при первой записи в страницу. Непосредственно после `fork(2)`, приватные сегменты остаются разделяемыми, но система устанавливает на страницы этих сегментов защиту от записи. Чтение таких страниц происходит без изменений, но при попытке модификации такой страницы, диспетчер памяти генерирует исключение защиты памяти. Ядро перехватывает это исключение, но вместо завершения процесса по `SIGSEGV` (как это происходит при обычных ошибках защиты памяти), создаёт копию приватной страницы и отображает эту копию в адресное пространство того процесса, который пытался произвести запись. Такое поведение называется копированием при записи (`copy-on-write`).

RETURN VALUES

Upon successful completion, **fork()**, **fork1()**, **forkall()**, **forkx()**, and **forkallx()** return **0** to the child process and return the process **ID** of the child process to the parent process. Otherwise, **(pid_t)-1** is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

ERRORS

The **fork()**, **fork1()**, **forkall()**, **forkx()**, and **forkallx()** functions will fail if:

EAGAIN
A resource control or limit on the total number of processes, tasks or LWPs under execution by a single user, task, project, or zone has been exceeded, or the total amount of system memory available is temporarily insufficient to duplicate this process.

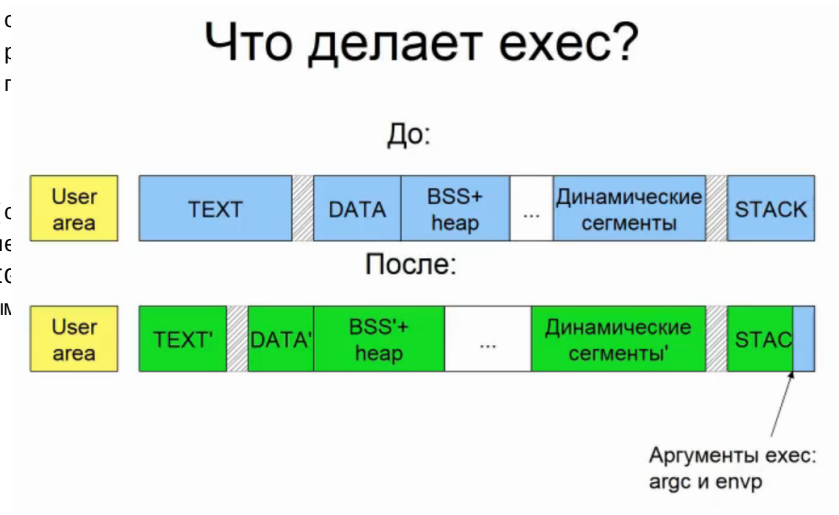
ENOMEM
There is not enough swap space.

EPERM
The {**PRIV_PROC_FORK**} privilege is not asserted in the effective set of the calling process.

The **forkx()** and **forkallx()** functions will fail if:

EINVAL
The flags argument is invalid.

```
WIFSIGNALED(stat)  Возвращает  ненулевое  значение,  если  это  слово состояния получено
  от  подпроцесса,  который  был принудительно завершен сигналом.

WTERMSIG(stat)  Если значение  WIFSIGNALED(stat)  ненулевое,  этот макрос возвращает
  номер сигнала,  который  вызвал завершение подпроцесса,
  иначе код возврата не определен.

WIFSTOPPED(stat)   Возвращает   ненулевое   значение,   если   слово состояния
  получено     от     приостановленного подпроцесса (wait(2) не реагирует на
  приостановленные подпроцессы, такое слово состояния может быть получено
  только вызовом waitpid(2)).

WSTOPSIG(stat)     Если  значение  WIFSTOPPED(stat)  ненулевое,  этот макрос возвращает
   номер сигнала,  который  вызвал приостановку подпроцесса, иначе код возврата не
   определен.

WIFCONTINUED(stat) Возвращает   ненулевое   значение,   если   с
  получено   от  процесса,  продолжившего исполнение (wait(2) не р
  приостановленные подпроцессы, такое слово состояния может быть п
  вызовом waitpid(2)).

WCOREDUMP(stat)    Если значение  WIFSIGNALED(stat)  ненулевое,
  ненулевое  значение,  если  был создан посмертный дамп памяти (с
  завершившегося подпроцесса. Факт создания дампа памяти определяе
  номеру сигнала; завершение по некоторым сигналам, таким, как SIG
  всегда приводит к созданию дампа памяти, завершение по остальным
  никогда не создает такой дамп.
```



Exec(2)

Процесс может заменить текущую программу на новую, исполнив системный вызов `exec(2)`. Этот вызов заменяет текст, данные, стек и остальные сегменты виртуального адресного пространства текущей программы на соответствующие сегменты новой программы. Однако пользовательская область при этом вызове сохраняется

NAME

exec, execl, execle, execlp, execv, execve, execvp - execute a file

SYNOPSIS

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...
     /* const char *argn, (char *)0 */);
```

- **path** указывает на строку, которая содержит абсолютное или относительное имя загрузочного модуля.
- **file** указывает на строку, содержащую имя загружаемого файла, который находится в одной из директорий, перечисленных в переменной `PATH`.
- **arg0,...,argn** указывают на строки -значения параметров, которые надо передать новой программе. Эти значения помещаются в вектор `argv[]` -параметр функции main() новой программы. Количество параметров помещается в параметр `argc` функции `main()`. Список параметров должен завершаться нулевым указателем.

Имя команды передаём как `path` и как `arg0`, то есть теоретически, можно передать разные значения, если это сделать, то можно нарваться на неприятность, потому что одна команда может иметь несколько имён и в зависимости от имени под которым её позвали делать разные вещи

все три команды один и тот же бинарник но завсист от того под каким именем мы ее позвали:

```
d.khaetskaya@fit-main:~$ ls -li `which cp` `which mv` `which ln`
     6760 -r-xr-xr-x   3 root     bin        41108 Feb  4  2020 /usr/bin/cp
     6760 -r-xr-xr-x   3 root     bin        41108 Feb  4  2020 /usr/bin/ln
     6760 -r-xr-xr-x   3 root     bin        41108 Feb  4  2020 /usr/bin/mv
d.khaetskaya@fit-main:~$
```

> Команда `ls-li` выводит номер инода (уникального идентификатора) файла. Видно, что оба файла имеют одинаковый номер инода. Это означает, что и `/usr/bin/cp`, и `/usr/bin/mv` – жесткие связи одного и того же файла.Таким образом, программа `cp(1)` определяет, удалять ли старый файл после копирования, именно на основании `argv[0]`

`arg0` или `argv[0]`, следует устанавливать равным последней компоненте path или параметру file, то есть равным имени загружаемой программы. Некоторые программы воспринимают нулевой аргумент как значимый параметр. Так, программы `gzip(1)` и `gunzip(1)` (потоковые архиватор и деархиватор) обычно представляют собой один и тот же бинарный файл, который определяет, что ему делать (упаковывать или распаковывать) по

- **argv[ ]** вектор указателей на строки, содержащие параметры, которые нужно передать новой программе. Преимущество использования `argv` состоит в том, что список параметров может быть построен динамически. Последний элемент вектора должен содержать нулевой адрес.
- **envp[ ]** вектор указателей на строки, представляющие новые переменные среды для новой программы. Значения элементов этого массива копируются в параметр `envp[]` функции `main()` новой программы. Аналогично, `environ` новой программы указывает на `envp[0]` новой программы. Последний элемент `envp[]` должен содержать нулевой адрес.
- **cnup** указатель на вектор указателей на строки, представляющие новые переменные среды новой программы; в действительности то же что и `envp[]`.
- **arg0 или argv[0]**, следует устанавливать равным последней компоненте `path` или параметру `file`, то есть равным имени загружаемой программы.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Все семь функций возвращают –1 в случае ошибки, не возвращают управление в случае успеха

Одно из отличий между этими функциями заключается в том, что первые четыре принимают полный путь к файлу, следующие две — только имя файла и последняя — дескриптор файла. Аргумент *filename* интерпретируется следующим образом:

- если аргумент *filename* содержит символ слеша, он интерпретируется как полный путь к файлу;
- иначе производится поиск выполняемого файла в каталогах, перечисленных в переменной окружения PATH.

Переменная окружения PATH содержит список каталогов, разделенных двоеточиями; они называются префиксами пути. Например, строка окружения в формате *name=value*

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

определяет четыре каталога, в которых будет производиться поиск выполняемых файлов. Последним указан текущий каталог. (Пустой префикс также означает текущий каталог. Он может быть определен двоеточием в начале, двумя двоеточиями в середине или двоеточием в конце подстроки *value*.)

> *По причинам, связанным с безопасностью системы, никогда не включайте текущий каталог в переменную окружения PATH. Подробности в [Garfinkel et al., 2003].*

Если функция `execlp` или `execvp` находит выполняемый файл, используя один из префиксов пути, но этот файл не является двоичным выполняемым файлом, сгенерированным редактором связей, функция предположит, что найденный файл является сценарием командной оболочки, и попытается вызывать `/bin/sh` с именем файла в качестве аргумента.

имени команды, которой он был запущен, то есть по `argv[0]`.

Hash bang

Когда скрипт с шебангом выполняется как программа в Unix-подобных операционных системах, загрузчик программ рассматривает остаток строки после шебанга как имя файла программы-интерпретатора. Загрузчик запускает эту программу и передаёт ей в качестве параметра имя файла скрипта с шебангом.[8] Например, если полное имя файла скрипта "`path/to/script`" и первая строка этого файла:

```
#!/bin/sh
```

то загрузчик запускает на выполнение "`/bin/sh`" (обычно это Bourne shell или совместимый интерпретатор командной строки) и передаёт "`path/to/script`" как первый параметр.

Таблица 8.6. Различия между семью функциями семейства exec

Функция	pathname	filename	fd	Список аргументов	argv[]	environ	envp[]
execl	✓			✓		✓	
execlp		✓		✓		✓	
execle	✓			✓			✓
execv	✓				✓	✓	
execvp		✓			✓	✓	
execve	✓				✓		✓
fexecve			✓		✓		✓
буква в имени		p	f	l	v		e

Аргументы всех семи функций семейства exec достаточно сложно запомнить. Но буквы в именах функций немного помогают в этом. Буква `p` означает, что функция принимает аргумент `filename` и использует переменную окружения PATH, чтобы найти выполняемый файл. Буква `l` означает, что функция принимает список аргументов, а буква `v` — что она принимает массив (вектор) `argv[]`. Наконец, буква `e` означает, что функция принимает массив `envp[]` вместо текущего окружения. В табл. 8.6 показаны различия между этими семью функциями.

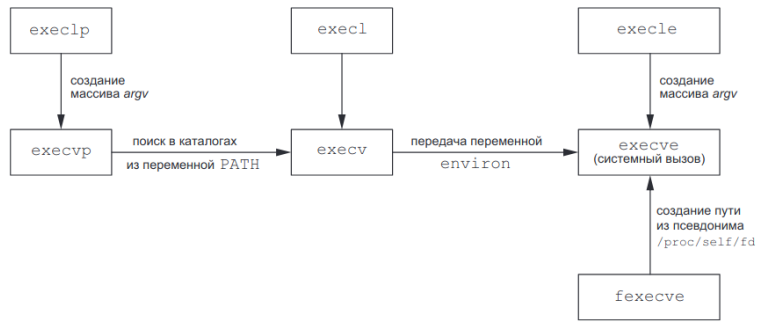


Рис. 8.2. Взаимоотношения между семью функциями exec

Reading list

- [ ]