💙

ЛАБ 9

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

void main() {
        FILE* f = fopen("text.txt", "w+");

        for (int i = 0; i < 1000; i++)
                fprintf(f, "%c%c", 'a' + i % ('z' - 'a' + 1), '\n');

        fclose(f);

        pid_t child;

        if ((child = fork()) == -1) {
                perror("Fork error\n");
                exit(EXIT_FAILURE);
        }

        if (child == 0){
                printf("It`s child process: %d\n", child);

                char* argv[3] = {"/bin/cat","text.txt", NULL};
                if (execv("/bin/cat", argv) == -1){
                        perror("Execv error\n");
                        exit(EXIT_FAILURE);
                }
        }

        printf("Waiting for child to finish\n");

        pid_t waitStatus = wait(NULL);

        if (waitStatus == -1){
                perror("Wait error\n");
                exit(EXIT_FAILURE);
        }

        printf("Child finished, my turn!\n");
}
```

▼ **FOPEN()**

FILE *fopen(const char *_path_, const char *_mode_) - Открывает файл с указанными правами доступа и связывает его с потоком

Функция **fopen** открывает файл с именем *path* и связывает его с потоком.

Параметр *mode* указывает на строку, начинающуюся с одной из следующих последовательностей (за ними могут следовать дополнительные символы):

r
Открыть текстовый файл для чтения. Чтение начинается с начала файла.
r+
Открыть для чтения и записи. Чтение или запись начинаются с начала файла.
w
"Урезать" файл до нулевой длины или создать текстовый файл и открыть его для записи. Запись начинается с начала файла.
w+
Открыть для чтения и записи. Файл создается, если до этого его не существовало, в противном случае он "урезается". Чтение или запись начинаются с начала файла.
a
Открыть для дописывания (записи в конец файла). Файл создается, если до этого его не существовало. Запись осуществляется в конец файла.
a+
Открыть для чтения и дописывания (записи в конец файла). Файл создается, если до этого его не существовало. Чтение или запись производятся с конца файла.

Возврат:

- **Успех:**

  FILE* структуру, которая используется для буферизованного ввода-вывода (stdio)

  Возвращает указатель на поток, который используется при последующих операциях с файлом.

- **Иначе**: NULL**+** *errno*

💡 ***Отличие от open():***

- если код придется перенести на другую платформу, которая поддерживает только ANSI C и не поддерживает функцию `open` (возвращает дескриптор файла).
- `fopen` обеспечивает буферизацию ввода-вывода, которая может оказаться намного быстрее, чем `open`

Open() - читает метаданные файла из файловой системы и на их основе создает структуру данных в памяти ядра - **файловый дескриптор. Принимает относительные и абсолютные имена файлов. Делает проверку по эффективному id**

Возвращает **ручка** (дескриптор файла(др), впоследствии может быть использован для чтения или записи)- небольшое целое число, уникальна для каждого процесса. Ручка хранится в памяти процесса и передается дочернему при вызове *fork()*

fclose(FILE ***stream*)** - Закрывает открытый поток.

▼ FORK()

fork() - системный вызов, создающий все нормальные процессы (в списке процессов могут быть имена с [] - это нити)

Возврат:

- **Успех-** родителю- PID процесса-потомка, а процессу-потомку 0.
- **Неудача-** родительскому процессу -1, процесс-потомок не создается + значение errno.

Он создает копию родительского процесса

- отличия - разные pid, разные ppid, код возврата forka, разные адресные пространства
- не наследуется - pid, ppid, захваченные участки файлов, нити исполнения кроме той которая вызвала fork()

Дочерний процесс наследует все **ручки** (дескрипторы файлов, использующиеся для чтения и записи. Хранится в памяти процесса) открытых файлов (но структура FILE* остается общая и если сделать lseek в копии ручки, то начальная ручка это будет видеть + чтобы закрыть файл, нужно закрыть копии ручек во всех процессах) и ряд других параметров (ограничения, pgid, sid, uid, gid)

1) **Сегмент данных**(статические и внешние переменные) у процессов сделаны при помощи **copy-on-write** - система ставит защиту от записи на все страницы.

- При обращении на чтение - читается разделяемая копия.
- При попытки модификации прилетает исключение диспетчера памяти, обработчик в ядре видит, что это **copy-on-write**, дублирует страницу и отдает какому-то процессу

2) **User area** копируется по-честному, чтобы ядро могло видеть атрибуты процесса на разных адресах

▼ **user area**

ОС сохраняет информацию о процессах в структурах данных, размещаемых в памяти ядра: в дескрипторах процесса и пользовательских областях.

Пользовательская область — это системный сегмент данных небольшого фиксированного размера, который содержит информацию, необходимую при исполнении этого процесса, например дескрипторы открытых файлов, реакцию на сигналы, информация о системных ресурсах.

Для получения/установки информации, хранящейся в user area, использоваться системные вызовы.
Кроме атрибутов процесса, пользовательская область содержит стек, которым ядро пользуется при выполнении системных вызовов. Функции ядра не могут размещать свои записи активации на пользовательском стеке, ведь тогда другая нить пользовательского процесса могла бы подменить параметры функций или адрес возврата и вмешаться в работу кода ядра.

Если в рамках процесса исполняется несколько нитей, то пользовательская область должна содержать соответствующее количество стеков, чтобы каждая нить могла исполнять системные вызовы независимо от других.

*Пользовательская область не является непосредственно доступной для пользователя, тк находится в памяти ядра. Даже если ядро отображено в адресное пространство процесса, на соответствующих страницах памяти стоят атрибуты, делающие эти страницы доступными только в системном режиме.

▼ **EXEC()**

Очень часто сразу же после вызова fork() следует вызов функции из семейства exec() - отличаются они переданными параметрами, но выполняют одну и ту же функцию - полное замещение текущего процесса новой программой, переданной в качестве аргумента. В UNIX это единственный вариант запустить новую программу.

exec() - исполняет новую программу в текущем процессе

Если он завершился успехом, то все текущее АП заменяется на созданное на основе бинарника, который был передан в качестве 1ого параметра (если бинарник на C - то не с main(), запускается некий код CRuntimeLibrary - не является частью стандартной библиотеки и лежит в отдельном объектном файле)

- execl - позиционные аргументы передаются как отдельные аргументы char(если еще 1 аргумент - переписывать исходник, в конце нужно ставить нулевой указатель, тк это функция с переменным числом аргументов)
- execv - аргументы передаются как char*, в конце 0ой указатель (v - vector)

  1ый аргумент - имя файла. Оно должно совпадать с первым позиционным аргументом
- exec без p - имя файла, ищет его как open (если начинатеся с / - абсолютный путь, если не с него - относительный, если не содержит вообще / - путь в текущем каталоге)
- execp - если имя содержит / - работает как exec, если не содержит - через PATH (Ищется первый каталог, где есть такое имя)

Сначала устанавливается новая среда, затем делается exec. Если не передать в новое окружение PATH, то используется PATH зашитый в библиотеку дефолтный PATH

Выполнение:

Если exec нашел программу, она исполняема и верного формата: выкидывает АП процесса, копирует в новое АП аргументы и запускает новую программу. Создает сегменты в соответсвии с содержимым бинарника: сегменты текста, данных - mmap соответствующих секций исполняемого файла. На дно стека кладутся аргументы argc и envp.

Содержимое User area у процесса почти не меняется: все файлы остаются открытыми, те будет много октрытых ручек. Можно поставить флаг CLOEXEC(fcntl)

Наследуется:

- Почти все содержимое User area
- Открытые файлы (кроме CLOEXEC)
- Захваченные участки файлов (на лекции было сказано, что не наследуются)
- euid/egid (если файл не setuid/setgid)

Не наследуются:

- обработчики сигналов
- АП и отображенные на память файлы(mmap)
- контекст процесса (регистры ЦПУ)
- нити исполнения

Копируются: аргументы argc, environ

▼ **WAIT()**

pid_t wait(int *_status_) - Функция, 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 потомок завершился раньше вызова wait(), то wait() уничтожает зомби-запись этого процесса. Чтобы избежать зомби - надо делать wait) на своих потомков

Возврат:

- **Успех** - pid завершившегося подпроцесса
- **Неуспех** - -1, errno установлена

int* status- 2 байта: Младший байт хранить номер сигнала / 0, если процесс завершился по exit. Во 2 байте: код завершение процесса / 0 - если процесс умер по сигналу. Содержимое старших двух байтов целочисленного значения не определено, на практике оно
обычно 0.

Параметр wait(2) - указатель на целое число, по которому размещается слово состояния подпроцесса

Слово состояния процесса - в регистре содержатся биты кода условия, устанавливаемые инструкциями сравнения, а также биты управления приоритетом центрального процессора, режимом (пользовательским или ядра) и другие служебные биты.

Обычно режимом(системный или пользовательский) управляет специальный бит в слове состояния программы, его нельзя менять в пользовательском режиме

Обычно пользовательские программы могут считывать весь регистр целиком, но записывать — только в некоторые из его полей.

▼ **WAITID, WAITPID**

waitid (idtype_t idtype, id_t id, siginfo_t *infop, int options) - позволяет отбирать потомков по тому, что с ними произошло

Успех: pid процесса

waitpid(pid_t pid, int *stat_loc, int options) - ожидание изменения состояния процесса

Успех: pid процесса