**ОБЩАЯ ТЕОРИЯ**

**Процесс** — это исполняющийся экземпляр программы.

Создание процесса — это не магия, а строго определённая последовательность:

1. Родитель → Рождает потомка

Процессы не возникают из ниоткуда. Почти все они **порождаются другим процессом.**

Пример: Процесс bash порождает ls

systemd (PID 1) порождает bash

Изначально есть только один первичный процесс — init или systemd, запускаемый ядром после инициализации.

2. Системный вызов fork()

Когда процесс вызывает fork(), происходит следующее:

- В user space:

В коде: pid\_t pid = fork();

Возвращает:

0 — в дочернем процессе

PID дочернего — в родителе

- В kernel space:

Вызов sys\_fork() (или clone() в современных Linux)

Ядро:

Создает новую структуру процесса (task\_struct)

Копирует родительское адресное пространство (copy-on-write)

Копирует открытые файловые дескрипторы, дескрипторы сигналов, состояние регистров и др.

Задаёт новый PID

Вставляет новый процесс в таблицу процессов (process table)

Это называется copy-on-write fork — память копируется только при записи.

**Важно**: fork() не запускает новую программу. Он просто дублирует текущий процесс!

**Copy-on-write** — это ленивая стратегия копирования памяти: копия создается только при попытке записи. До этого момент копии разделяют одну и ту же физическую память.

Когда вызывается fork():

1. Ядро создает новый task\_struct (структура процесса).

2. Адресное пространство не копируется целиком:

Вместо этого обе структуры (родитель и потомок) получают указатели на одну и ту же физическую память.

3. Все страницы помечаются как "только для чтения".

4. Устанавливается флаг COW.

**Что происходит при записи?**

Когда один из процессов (родитель или потомок) попытается записать в память:

1. Происходит page fault (ошибка доступа к памяти).

2. Ядро ловит исключение и понимает: это COW-страница.

3. Ядро:

- Создает новую физическую страницу.

- Копирует туда содержимое оригинальной страницы.

- Обновляет таблицу страниц этого процесса.

- Делает новую страницу доступной для записи.

4. Теперь оба процесса имеют свои независимые страницы.

*1. Жизненный цикл процесса.*

*a. Напишите программу, которая:*

*i. создает и инициализирует переменную (можно две: локальную и глобальную);*

*ii. выводит ее (их) адрес(а) и содержимое;*

*iii. выводит pid;*

*iv. порождает новый процесс (используйте fork(2)).*

*v. в дочернем процессе выводит pid и parent pid.*

*vi. в дочернем процессе выводит адреса и содержимое переменных,*

*созданных в пункте а;*

*vii. в дочернем процессе изменяет содержимое переменных и*

*выводит их значение;*

*viii. в родительском процессе выводит содержимое переменных;*

*ix. в родительском процессе делает sleep(30);*

*x. в дочернем процессе завершается с кодом “5” (exit(2)).*

*xi. в родительском процессе дожидается завершения дочернего, вычитывает код завершения и выводит причину завершения и кодзавершения если он есть. В каком случае кода завершения не*

*будет?*

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int global\_var = 42;

int main() {

int local\_var = 10;

printf("Родительский PID: %d\n", getpid());

printf("Глобальная переменная: адрес=%p, значение=%d\n", &global\_var, global\_var);

printf("Локальная переменная: адрес=%p, значение=%d\n", &local\_var, local\_var);

pid\_t pid = fork();

if (pid == -1) {

perror("Ошибка fork()");

return 1;

}

if (pid == 0) {

printf("Дочерний PID: %d\n", getpid());

printf("Дочерний PPID (родительский PID): %d\n", getppid());

printf("Дочерний: Глобальная переменная: адрес=%p, значение=%d\n", &global\_var, global\_var);

printf("Дочерний: Локальная переменная: адрес=%p, значение=%d\n", &local\_var, local\_var);

global\_var = 100;

local\_var = 50;

printf("Дочерний: Изменённые значения: global=%d, local=%d\n", global\_var, local\_var);

printf("Дочерний: Глобальная переменная: адрес=%p, значение=%d\n", &global\_var, global\_var);

printf("Дочерний: Локальная переменная: адрес=%p, значение=%d\n", &local\_var, local\_var);

sleep(60);

exit(5);

}

else {

printf("\nРодительский: Глобальная переменная: значение=%d\n", global\_var);

printf("Родительский: Локальная переменная: значение=%d\n", local\_var);

sleep(30);

int status;

waitpid(pid, &status, 0);

if (WIFEXITED(status)) {

printf("Дочерний процесс завершился с кодом: %d\n", WEXITSTATUS(status));

} else if (WIFSIGNALED(status)) {

printf("Дочерний процесс убит сигналом: %d\n", WTERMSIG(status));

} else {

printf("Дочерний процесс завершился по неизвестной причине\n");

}

}

return 0;

}

**Что делает fork() с адресным пространством?**

Когда происходит fork(), ОС создаёт новый процесс — дочерний — с копией всех структур родителя:

Копируются:

- Таблицы страниц

- Файловые дескрипторы

- Контекст выполнения (регистры, стек)

- Окружение

**Но не физически: используется Copy-On-Write (COW)!**

int global\_var = 42;

int local\_var = 10;

Обе переменные попадают в адресное пространство процесса:

- global\_var — в .data (сегмент глобальных переменных)

- local\_var — в стеке

**После fork():**

У родителя и дочернего процесса эти переменные находятся по одинаковым виртуальным адресам.

Физическая память одна до первой записи (Copy-On-Write).

Когда дочерний процесс меняет значения (global\_var = 100; local\_var = 50;), происходит:

- Page fault (ошибка записи)

- Ядро копирует соответствующую страницу

Теперь у родителя и дочернего процесса разные физические страницы



**Код возврата (exit code)** — это целое число, которое процесс возвращает операционной системе при завершении. Он сообщает родительскому процессу или оболочке, как именно завершился процесс:

- Успешно?

- С ошибкой?

- Был прерван?

- Убит сигналом?

Ты явно указываешь, что процесс завершается с кодом 5. Это может обозначать что угодно — ОС не интерпретирует это значение автоматически как ошибку или успех.

WIFEXITED(status) → проверяет, завершился ли нормально (exit, return)

WEXITSTATUS(status) → извлекает код возврата

Если процесс завершён нештатно, то WIFEXITED(status) будет false.

Примеры:

Программа упала с segfault (например, разыменование NULL)

Убита сигналом: kill(pid, SIGKILL)

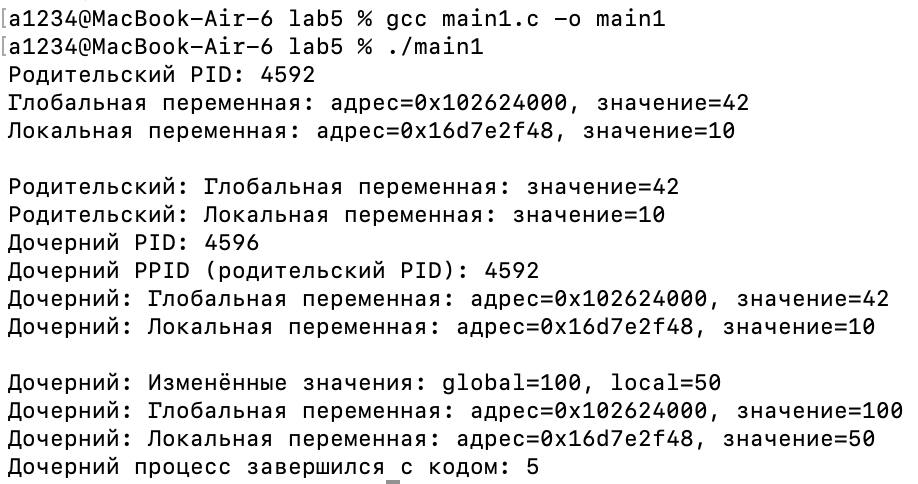
Обработка в родителе:

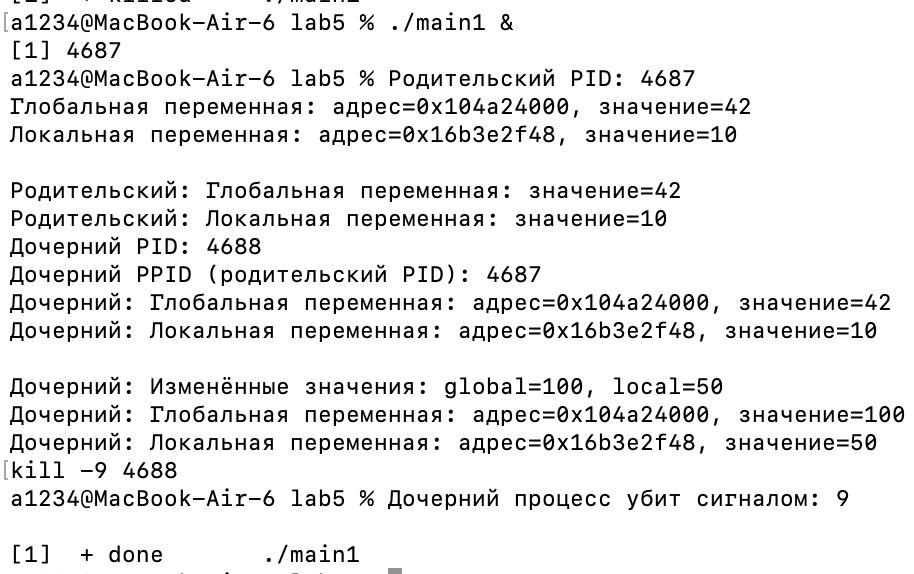
if (WIFSIGNALED(status)) {

printf("Процесс убит сигналом: %d\n", WTERMSIG(status));

}

Код завершения процесса (exit code) не будет доступен в тех случаях, когда **процесс не завершился нормально**, то есть не вызвал exit() или return, а был принудительно **завершён сигналом**.





*c. Понаблюдайте за адресными пространствами в procfs.*

Что увидим у родителя и дочернего процесса сразу после fork()?

Будет почти полное совпадение!

У процессов:

- Те же адреса (виртуальные!) для кода, стека, кучи

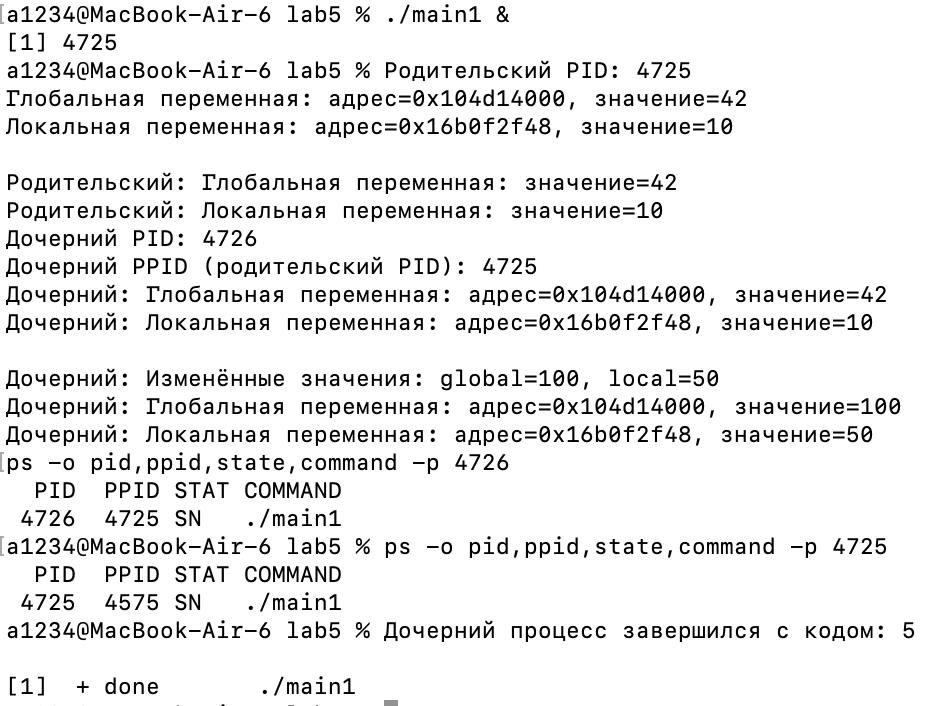
- Те же участки памяти (/proc/<PID>/maps почти идентичны)

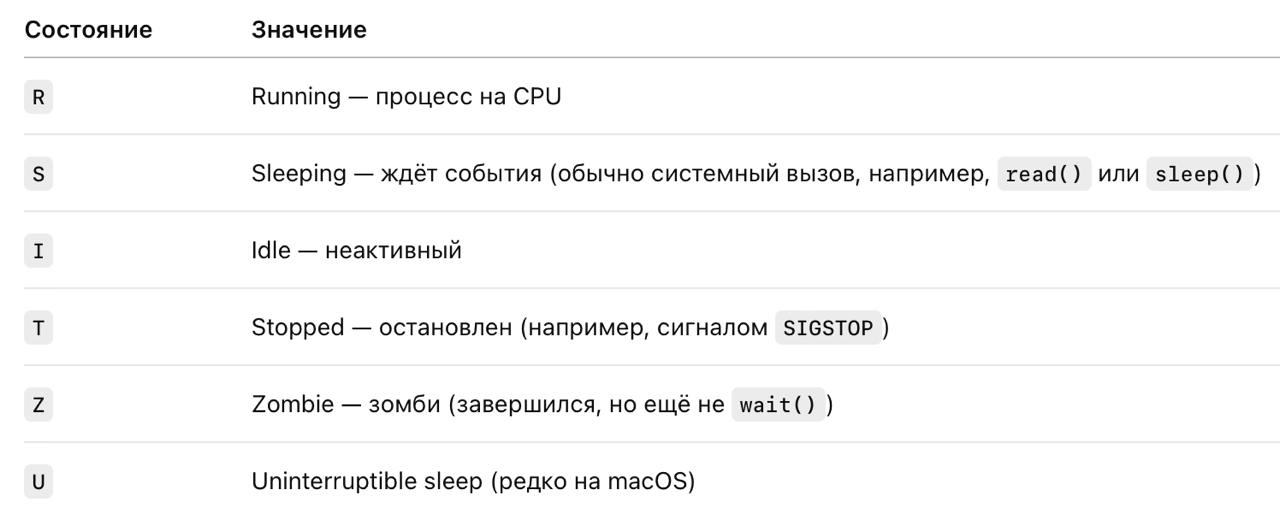
- Те же отображения библиотек

- Одни и те же значения переменных

Почему?

fork() не копирует память сразу, он создаёт новый процесс с таким же адресным пространством, но использует механизм "copy-on-write" (COW).





*d. Понаблюдайте за состояниями процесса в procfs или с помощью утилиты*

*ps.*

ps -o pid,ppid,state,command -p <PID>