# Лекція 10-11. Process Creation у Linux

## 1. Вступ

Процеси є основною одиницею виконання в Linux. Кожна програма, яка запускається, працює в контексті процесу. Одним з базових механізмів, що дозволяє створювати нові процеси, є системний виклик fork(). Цей механізм відіграє ключову роль у функціонуванні операційної системи, дозволяючи програмам породжувати інші програми або копії самих себе.

### Як працює fork()

Системний виклик fork() створює новий процес, який є точною копією поточного (батьківського) процесу. Цей новий процес називається дочірнім. Важливо, що після виклику fork() обидва процеси — батьківський і дочірній — продовжують виконання з того самого місця, але кожен — у своєму адресному просторі.

Єдиною відмінністю є значення, яке повертається fork():

- у дочірньому процесі — це 0,
- у батьківському — ідентифікатор процесу (PID) дитини,
- у випадку помилки — -1.

### Використання fork()

fork() використовується у програмі для створення паралельного виконання. Це може бути реалізація багатозадачності, створення оболонок, мережевих серверів, обробників запитів тощо. Після fork() програма може реалізувати окрему логіку для батьківського та дочірнього процесів, використовуючи умовну конструкцію на основі повернутого значення.

**Моделювання роботи в батьківському і дочірньому процесах**

Щоб краще зрозуміти роботу fork(), можна зробити експеримент: у кожному процесі реалізувати нескладний цикл, який виводить ідентифікатор процесу та ітерацію. Це дозволяє побачити, що обидва процеси працюють одночасно, хоча незалежно один від одного. Для наочності можна додати затримку sleep(1).

**Копіювання даних при fork()**

Коли процес копіюється за допомогою fork(), у кожного з процесів утворюється **власна копія** всіх змінних. Це означає, що зміни, які внесе дочірній процес у свої змінні, не вплинуть на відповідні змінні в батьківському процесі, і навпаки. Сучасні операційні системи застосовують механізм "копіювання при записі" (copy-on-write), який оптимізує цей процес, копіюючи сторінки пам’яті лише у разі модифікації.

**Процеси та відкриті файли**

Файлові дескриптори теж копіюються при fork(), і батьківський та дочірній процеси можуть мати доступ до одного й того самого файлу. Це означає, що позиція у файлі також спільна — якщо один із процесів читає чи записує, інший це помітить. Це потрібно враховувати при синхронізації доступу до файлів.

**Dumb Shell — мінімальна оболонка**

Щоб закріпити розуміння fork() на практиці, можна реалізувати просту командну оболонку, яка:

1. Зчитує команду від користувача.

2. Створює новий процес (fork()).

3. Використовує execlp() для запуску команди в дочірньому процесі.

4. Очікує завершення дочірнього процесу (wait()).

Це дозволяє імітувати базову роботу Unix shell, яка створює процес для кожної команди.

**Запуск dumbsh**

Програма компілюється звичайним компілятором GCC і запускається як будь-яка інша. Користувач вводить команди (наприклад, ls, whoami, date), і оболонка виконує їх у дочірньому процесі. Введення exit завершує оболонку.

**Очікування завершення процесів: API wait**

Щоб уникнути створення "зомбі-процесів", батьківський процес повинен викликати wait(), щоб дочекатися завершення своїх дочірніх процесів. wait() повертає PID дочірнього процесу, а також дозволяє отримати код його завершення. Якщо wait() не викликається, завершені дочірні процеси залишаються в таблиці процесів як зомбі.

Системний виклик waitpid() дозволяє гнучкіше керувати процесами: чекати на конкретного нащадка, не блокуючи при цьому виконання (WNOHANG), або отримати розширену інформацію про завершення процесу.

**Fork bomb та створення декількох дітей**

fork() може бути використаний не лише конструктивно. При безконтрольному його використанні можна створити так звану **fork bomb** — нескінченне створення процесів, що швидко вичерпує ресурси системи.

Це показує важливість обмежень (ulimit), а також контролю над кількістю створюваних дочірніх процесів у навчальних і реальних програмах.

У практиці часто створюють кількох дітей в циклі, обробляючи кожного з них окремо і очікуючи завершення через wait().

**Варіанти використання wait**

Існує кілька варіантів використання очікування дочірніх процесів:

- wait(NULL) — очікує будь-який дочірній процес.
- waitpid(pid, &status, 0) — чекає на конкретний PID.
- waitpid(pid, &status, WNOHANG) — не блокує, якщо процес ще не завершено.

Ці варіанти дозволяють будувати неблокуючі або реактивні системи.

**Правила використання fork()**

- fork() створює новий процес, копіюючи пам’ять, дескриптори, середовище.
- Обидва процеси працюють незалежно.
- Дані не синхронізуються — кожен має свою копію.
- Файли спільні до закриття.

- Для уникнення зомбі-процесів слід використовувати wait().
- Після fork() часто викликається exec() — для запуску іншої програми.

## 2. Приклади коду

### 2. 1. Як працює fork()

- fork() створює **копію** поточного процесу (батьківського).
- Обидва процеси (батьківський і дочірній) продовжують виконання **з однакової точки** після виклику fork.
- Вони можуть бути розрізнені за значенням, яке повертає fork():
  - 0 — ви в дочірньому процесі,
  - >0 — ви в батьківському (значення — PID дочірнього),
  - <0 — помилка.

### 2.3. Використання системного виклику fork

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

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

   if (pid < 0) {
      perror("fork failed");
```

```
    } else if (pid == 0) {
        printf("This is the child process\n");
    } else {
        printf("This is the parent process\n");
    }

    return 0;
}
```

### 2.4. Експеримент: імітація роботи в процесах

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

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

    if (pid == 0) {
        for (int i = 0; i < 5; i++) {
            printf("Child: %d\n", i);
            sleep(1);
        }
    } else {
        for (int i = 0; i < 5; i++) {
            printf("Parent: %d\n", i);
            sleep(1);
        }
    }
    return 0;
}
```

### 2.5. Копіювання даних при fork

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

```
int main() {
    int counter = 0;
    pid_t pid = fork();

    if (pid == 0) {
        counter++;
        printf("Child: counter = %d\n", counter);
    } else {
        sleep(1);
        printf("Parent: counter = %d\n", counter);
    }

    return 0;
}
```

Пам’ять **копіюється**, і значення змінної counter розділене.

### 2.6. Процеси та відкриті файли

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd = open("test.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    pid_t pid = fork();

    if (pid == 0) {
        write(fd, "Child\n", 6);
    } else {
        write(fd, "Parent\n", 7);
    }

    close(fd);
    return 0;
```

}

Файлові дескриптори **успадковуються**, але мають спільну таблицю позицій у файлі.

### 2.7. Проєкт: Dumb Shell (DumbSH)

**Ціль:**

Створити просту оболонку, яка запускає команди користувача.

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

#define MAX 1024

int main() {
   char line[MAX];

   while (1) {
      printf("dumbsh> ");
      if (!fgets(line, MAX, stdin)) break;

      line[strcspn(line, "\n")] = 0;

      if (strcmp(line, "exit") == 0) break;

      pid_t pid = fork();
      if (pid == 0) {
         execlp(line, line, NULL);
         perror("exec failed");
         exit(1);
      } else {
         wait(NULL);
```

```
        }
    }

    return 0;
}
```

### 2.8. Запуск dumbsh

1. Зберегти як dumbsh.c

Скомпілювати:

```
gcc dumbsh.c -o dumbsh
```

2.

Запустити:

```
./dumbsh
dumbsh> ls
dumbsh> whoami
dumbsh> exit
```

### 2.9. API wait — деталі

```
#include <sys/wait.h>

int status;
pid_t pid = wait(&status);

if (WIFEXITED(status)) {
    printf("Exited with code %d\n", WEXITSTATUS(status));
}
```

- wait() — чекає на будь-який дочірній процес.
- waitpid(pid, ...) — чекає на конкретний PID.
- WNOHANG — не блокує, якщо дитина ще не завершилась.

### 2.10. Fork Bomb та створення декількох дітей

```
// НЕ ЗАПУСКАЙТЕ ЦЕ! Це fork bomb!
:(){ :|:& };:
```

Це призводить до вичерпання ресурсів.

**Приклад створення кількох дітей:**

```
for (int i = 0; i < 3; i++) {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child %d\n", i);
        exit(0);
    }
}
while (wait(NULL) > 0); // чекає всіх
```

### 2.11. Варіації wait

```
wait(NULL); // будь-який дочірній
waitpid(-1, &status, 0); // еквівалент wait
waitpid(pid, &status, 0); // конкретний процес
waitpid(pid, &status, WNOHANG); // не блокує
```

## 3. Правила fork() — підсумок

fork() копіює:

- пам'ять
- файлові дескриптори
- сигнали

не копіює:

- потоки
- файлові блокування
- pthread-mutex

**Завдання для практики**

1. Створіть програму, яка після виклику fork() змінює значення глобальної змінної у батьківському процесі. Виведіть значення змінної в обох процесах, щоб переконатися, що дані не спільні.

2. Напишіть програму, в якій дочірній процес виводить свій PID та PID батьківського процесу, а потім завершується.

3. Використайте fork() двічі поспіль у програмі. Порахуйте кількість процесів, що виникнуть. Виведіть їхні PID.

4. Створіть файл, відкрийте його у програмі, виконайте fork() і зробіть запис у файл як у батьківському, так і в дочірньому процесі. Перевірте результат у вмісті файлу.

5. Зробіть програму, в якій дочірній процес завершується з певним кодом (exit(7), наприклад), а батьківський читає цей код через wait()

та виводить його.

6. У циклі створіть 5 дочірніх процесів. Кожен з них має вивести свій номер та завершитись. Батьківський процес чекає завершення всіх.

7. Реалізуйте послідовне породження процесів, де кожен створений процес створює ще один дочірній, усього п’ять разів. Прослідкуйте порядок завершення.

8. Створіть програму, де кожен процес породжує два нових дочірніх процеси. Побудуйте дерево з 3 рівнів.

9. Напишіть програму, яка створює 10 дочірніх процесів, зберігає їхні PID у масиві та чекає завершення кожного через wait().

10. Напишіть програму, яка у кожній ітерації циклу спочатку створює одного дочірнього процесу, чекає його завершення, і лише тоді створює наступного.

11. Створіть програму, де після fork() дочірній процес виконує команду ls -l за допомогою execlp().

12. Змусьте дочірній процес виконати команду echo Hello from child, передаючи параметри через execlp().

13. Розширте просту оболонку так, щоб вона дозволяла вводити команду з аргументами, розділеними пробілами, і запускати їх у дочірньому процесі.

14. Напишіть дві програми: одна просто виводить повідомлення, інша — викликає її після fork() за допомогою exec().

15. Створіть декілька процесів, які завершуються з різною затримкою. Спостерігайте в батьківському процесі, як wait() повертає PID завершених процесів у довільному порядку.

16. Змусьте батьківський процес чекати завершення лише певного дочірнього процесу, використовуючи waitpid() з відповідним PID.

17. Використайте waitpid() з прапором WNOHANG, щоб не блокувати батьківський процес при очікуванні на завершення дочірнього з затримкою.

18. Додайте до програми обробник сигналу SIGCHLD, який буде виводити повідомлення про завершення дочірнього процесу.

19. Створіть цикл, у якому викликається fork(), поки це можливо. Визначте, скільки процесів вдалося створити до помилки.

20. Створіть дерево процесів, у якому кожен нащадок створює ще одного. Переконайтесь, що всі процеси завершуються правильно, навіть якщо батьківський процес завершується раніше.
21. Реалізуйте програму, яка створює два дочірні процеси, кожен з яких виводить своє повідомлення.

22. Удоскональте dumbsh — додайте обробку аргументів до команд.

23. Напишіть програму, яка створює 3 дочірніх процеси і чекає їх завершення в довільному порядку.

24. Змініть приклад із відкритим файлом: спробуйте писати в нього з обох процесів із затримкою.

25. Спробуйте реалізувати псевдо-fork bomb з обмеженням кількості створених процесів (наприклад, 5).