**Робота №6. Створення процесів у Linux із застосуванням системних викликів fork() і exec()**

### Мета

**Оволодіння практичними навичками застосування системних викликів у програмах, дослідження механізму створення процесів у UNIX-подібних системах.**

### Завдання для самостійної підготовки

1. 1. Ознайомитись з документацією системних викликів fork() і exec():

 man pages;

 книги з числа рекомендованих [1, 5];

 [6] (електронний ресурс на сайті www.ibm.com).

1. 2. Перевірити, чи встановлений у вашій системі Linux компілятор С/С++ (g++). Якщо ні, встановіть за допомогою менеджера пакетів.

# Довідковий матеріал

Тут наведено лише мінімальну інформацію, достатню хіба що для того, щоби зрозуміти, про що йде мова. Решту інформації необхідно здобути з джерел, названих вище.

Створення процесу у UNIX-подібних системах відбувається у два кроки, для чого існують два різних сімейства викликів.

 Перший виклик — це fork() (фактично —

“розгалуження”). Виклик створює тотожню копію існуючого процесу, копіюючи також адресний простір. При цьому “старому” (так званому батьківському) процесу в результаті виклику повертають PID “нового” (так званого дочірнього процесу, або процесу-нащадка), а нащадку повертають значення 0. Якщо виклик завершився з помилкою, повертають 1. Дещо детальніше робота цього виклику була описана вище у Роботі №5. Слід пам’ятати, що є безліч деталей — що у нащадка буде те ж саме, що й у батьківського процесу, а що буде іншим, тому слід уважно перевіряти це за документацією. Після виклику fork() будуть паралельно працювати обидва процеси, виконуючи один і той самий код, але з різними значеннями, що їх повернув виклик.

Існує виклик vfork(), який відрізняється тим, що адресний простір не копіюється (тобто, зокрема, обидва процеси мають доступ до одних і тих самих змінних, а не до різних їх копій), процес-нащадок виконується, а батьківський процес призупиняється до завершення процесу-нащадка.

Другий виклик — exec() — представлений сімейством функцій, які розрізняються наборами параметрів, які вони приймають, і тим, як вони передають їх новому процесу. Спільна ідея — замінити поточний процес на інший. Точніше, у тому самому процесі (що зберігає свій PID) починає виконуватись новий програмний код, що його завантажують з виконуваного файлу, у загальному випадку за допомогою параметрів виклику змінюючи його оточення і передаючи йому параметри командного рядка.

Згідно стандарту POSIX, функції exec() оголошуються у файлі unistd.h (у деяких реалізаціях імена функцій можуть починатись з символу підкреслювання, наприклад \_execl()).

int execl(char const \*path, char const \*arg0, ...);

int execle(char const \*path, char const \*arg0, ..., char const \*envp[]);

int execlp(char const \*file, char const \*arg0, ...); int execv(char const \*path, char const \*argv[]);

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

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

Фактично, назви мають сенс: exec – “виконати”, а далі кожна літера має певне значення:

e – образу нового процесу у явному вигляді передається масив вказівників на змінні оточення (тобто, функції execve() і execle() запускають нову програму в новому оточенні, яке задається аргументами виклику, а інші функції запускають нову програму у тому самому отченні, що існувало до виклику);

l – функції передається список (list) аргументів командного рядка (список має закінчуватись нульовим аргументом);

p – використовує змінну оточення PATH, щоби знайти виконуваний файл названий в аргументі file (з одного боку, не треба задавати повний або відносний шлях до файлу, але з іншого боку якщо поточний каталог явно не вказаний у PATH, то файл з поточного каталогу знайдений і запущений не буде);

v – аргументи командного рядка передаються функції як масив (vector) вказівників.

Якщо передається аргумент path, то наступний аргумент arg0 – це і є їм’я файлу, який треба запустити на виконання. У більшості випадків в якості path і в якості arg0 задають один і той самий рядок — ім’я виконуваного файлу зі шляхом до нього.

# Завдання до виконання

1. Для початку можна взяти демонстраційну програму, запропоновану Greg Ippolito [7][[1]](#footnote-1).

1. #include <iostream>
2. #include <string>

03

1. // Required by for routine
2. #include <sys/types.h>
3. #include <unistd.h>
4. #include <stdlib.h> // Declaration for exit()

08

09 **using** **namespace** std;

10

11 **int** globalVariable = 2;

12

1. Main()
2. {
3. string sIdentifier;
4. **int** iStackVariable = 20;

17

1. pid\_t pID = fork();
2. **if** (pID == 0) // child
3. {
4. // Code only executed by child process

22

1. sIdentifier = "Child Process: ";
2. globalVariable++;
3. iStackVariable++;
4. }
5. **else** **if** (pID < 0) // failed to fork
6. {
7. cerr << "Failed to fork" << endl;
8. **exit**(1);
9. // Throw exception
10. }
11. **else** // parent
12. {
13. // Code only executed by parent process

36

1. sIdentifier = "Parent Process:";
2. } 39

40 // Code executed by both parent and child.

41

1. cout << sIdentifier;
2. cout << " Global variable: " << globalVariable;
3. cout << " Stack variable: " << iStackVariable << endl;
4. }

2. Скомпілюйте програму (вважаємо, текст збережено у файлі myforktest.cpp)

### g++ -o myforktest myforktest.cpp

**Увага!** Пам’ятайте, що не можна називати власні програми просто test! test – це вбудована команда shell (з якою ви вже зустрічалися в Роботі №4).

1. Запустіть програму myforktest. У якій послідовності виконуються батьківський процес і процес-нащадок? Чи завжди цей порядок дотримується?
2. Додайте затримку у виконання одного або обох з цих процесів (функція sleep(), аргумент — затримка у секундах). Чи змінились результати виконання?
3. Додайте цикл, який забезпечить кількаразове повторення дій після виклику fork(). Які результати показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.
4. Спробуйте у первинній програмі (без циклу) замість виклику fork() здійснити виклик vfork(). У чому різниця роботи цих двох викликів? Чи виникає помилка (якщо так, то яка)? У чому причина? Як “змусити” працювати виклик vfork()? Які результати тепер показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.
5. Тепер додайте виклик exec() у код процесу-нащадка. Для початку використайте простішу функцію execl(). Варіант виклику на прикладі утиліти ls:

**execl("/bin/ls", "/bin/ls", "-a", "-l", (char \*) 0);**

У наведеному прикладі передаються аргументи командного рядка.

1. Проведіть експерименти з викликом різних програм, у тому числі ps, bash, а також з викликами execl() у батьківському процесі. Як запустити фоновий процеснащадок? Як процес-нащадок дізнається власний PID? PID батьківського процесу?
2. Усі отримані результати і відповіді на запитання, які були задані вище, оформіть у вигляді протоколу.
1. У цій програмі закладені деякі змінні для використання у наступних програмах. Якщо бажаєте і впевнені у своїх діях, можете програму скоротити або взагалі написати власну із аналогічною функціональністю. [↑](#footnote-ref-1)