



МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»  
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ  
Кафедра Інформаційної Безпеки

## Операційні системи

### Комп'ютерний практикум

#### **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.

Перевірив:

\_\_\_\_\_

Виконав:

студент II курсу

групи ФБ-01

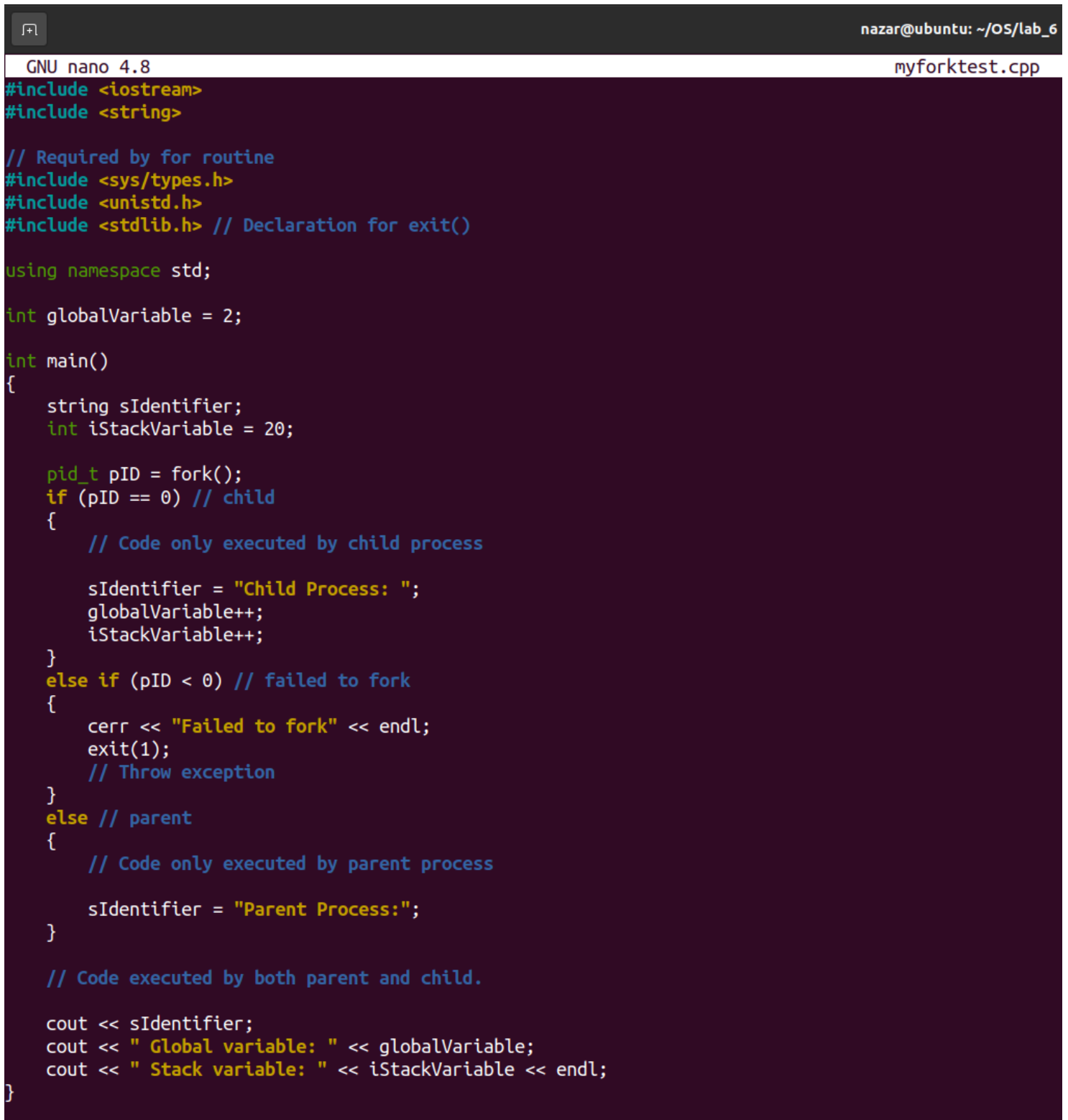
Сахній Н.Р.

Київ 2022

## Завдання ДО ВИКОНАННЯ:

1. Для початку можна взяти демонстраційну програму, запропоновану *Greg Ippolito*.

```
nazar@ubuntu:~$ cd OS; mkdir lab_6; cd lab_6
nazar@ubuntu:~/OS/lab_6$ nano myforktest.cpp
nazar@ubuntu:~/OS/lab_6$
```



```
GNU nano 4.8 myforktest.cpp
#include <iostream>
#include <string>

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

using namespace std;

int globalVariable = 2;

int main()
{
    string sIdentifier;
    int iStackVariable = 20;

    pid_t pID = fork();
    if (pID == 0) // child
    {
        // Code only executed by child process

        sIdentifier = "Child Process: ";
        globalVariable++;
        iStackVariable++;
    }
    else if (pID < 0) // failed to fork
    {
        cerr << "Failed to fork" << endl;
        exit(1);
        // Throw exception
    }
    else // parent
    {
        // Code only executed by parent process

        sIdentifier = "Parent Process:";
    }

    // Code executed by both parent and child.

    cout << sIdentifier;
    cout << " Global variable: " << globalVariable;
    cout << " Stack variable: " << iStackVariable << endl;
}
```

2. Скомпілюйте програму (вважаємо, текст збережено у файлі `myforktest.cpp`)  
`g++ -o myforktest myforktest.cpp`

```
nazar@ubuntu:~/OS/lab_6$ g++ -o myforktest myforktest.cpp
nazar@ubuntu:~/OS/lab_6$ ls -l
total 24
-rwxrwxr-x 1 nazar nazar 18288 Mar 27 21:53 myforktest
-rw-rw-r-- 1 nazar nazar  920 Mar 27 21:59 myforktest.cpp
nazar@ubuntu:~/OS/lab_6$
```

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

3. Запустіть програму `myforktest`.

```
nazar@ubuntu:~/OS/lab_6$ ./myforktest
Parent Process: Global variable: 2 Stack variable: 20
Child Process:  Global variable: 3 Stack variable: 21
nazar@ubuntu:~/OS/lab_6$
```

- У якій послідовності виконуються батьківський процес і процес-нащадок? Спочатку виконується батьківський процес, а потім процес-нащадок.

- Чи завжди цей порядок дотримується?

У проведених мною запусках програми порядок зберігався кожен раз, проте не завжди і не при будь-яких умовах виконання процесів дотримується такого порядку, так як це багато в чому залежить від алгоритму планування, що використовується ядром.

4. Додайте затримку у виконання одного або обох з цих процесів (функція `sleep()`, аргумент — затримка у секундах).

```
nazar@ubuntu:~/OS/lab_6$ nano myforktest.cpp
else // parent
{
    // Code only executed by parent process

    sleep(5);
    sIdentifier = "Parent Process.";
}
```

```
nazar@ubuntu:~/OS/lab_6$ g++ -o myforktest myforktest.cpp
nazar@ubuntu:~/OS/lab_6$ ./myforktest
Child Process:  Global variable: 3 Stack variable: 21
Parent Process: Global variable: 2 Stack variable: 20
nazar@ubuntu:~/OS/lab_6$
```

- Чи змінились результати виконання?

Із затримкою 5 секунд було виконано батьківський процес, тому можна помітити, що порядок виконання процесів змінився порівняно із попередніми запусками програми. Однак немає ніякої гарантії, що цей прийом спрацює за будь-яких умов, так як можливі випадки, що якщо система сильно завантажена, то навіть після затримки одного процесу, він може отримати керування раніше, ніж той, що її не мав, так як ще досі не завершив свою роботу.

5. Додайте цикл, який забезпечить кількаразове повторення дій після виклику `fork()`.

```
nazar@ubuntu:~/OS/lab_6$ nano myforktest.cpp
```

```
pid_t pID = fork();
for (int c = 1; c <= 3; c++)
{
    if (pID == 0) // child
    {
        // Code only executed by child process

        sIdentifier = "Child Process: ";
        globalVariable++;
        iStackVariable++;
    }
    else if (pID < 0) // failed to fork
    {
        cerr << "Failed to fork" << endl;
        exit(1);
        // Throw exception
    }
    else // parent
    {
        // Code only executed by parent process

        sleep(5);
        sIdentifier = "Parent Process:";
    }

    // Code executed by both parent and child.

    cout << sIdentifier;
    cout << " Global variable: " << globalVariable;
    cout << " Stack variable: " << iStackVariable << endl;
}
}
```

```
nazar@ubuntu:~/OS/lab_6$ g++ -o myforktest myforktest.cpp
```

```
nazar@ubuntu:~/OS/lab_6$ ./myforktest
```

```
Child Process: Global variable: 3 Stack variable: 21
Child Process: Global variable: 4 Stack variable: 22
Child Process: Global variable: 5 Stack variable: 23
Parent Process: Global variable: 2 Stack variable: 20
Parent Process: Global variable: 2 Stack variable: 20
Parent Process: Global variable: 2 Stack variable: 20
nazar@ubuntu:~/OS/lab_6$
```

- Які результати показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.

Батьківський процес залишається незмінним, а дочірні процеси, які він запускає, різні. Це все тому, що вони використовують свою копію відповідного сегмента коду.

6. Спробуйте у первинній програмі (без циклу) замість виклику `fork()` здійснити виклик `vfork()`.

```
nazar@ubuntu:~/OS/lab_6$ nano myforktest.cpp
int main()
{
    string sIdentifier;
    int iStackVariable = 20;

    pid_t pID = vfork();

nazar@ubuntu:~/OS/lab_6$ g++ -o myforktest myforktest.cpp
nazar@ubuntu:~/OS/lab_6$ ./myforktest
Child Process: Global variable: 3 Stack variable: 21
Parent Process: Global variable: 3 Stack variable: 21
*** stack smashing detected ***: terminated
Aborted (core dumped)
nazar@ubuntu:~/OS/lab_6$
```

- У чому різниця роботи цих двох викликів?

Виклик `vfork()` відрізняється від `fork()` тим, що адресний простір не копіюється (тобто, зокрема, обидва процеси мають доступ до одних і тих самих змінних у пам'яті, а не до різних їх копій, як це у виклику `fork()` )

- Чи виникає помилка (якщо так, то яка)?

Так, виникає помилка розбиття стека, а саме як можна помітити, неправильний результат виконання батьківського процесу.

- У чому причина?

Так як процес-нащадок продовжує виконуватись, а батьківський процес призупиняється до завершення процесу-нащадка, то в адресний простір було занесено дані, які батьківський процес також використав.

- Як “змусити” працювати виклик `vfork()`?

Необхідно викликати функцію `_exit()` або одну із функцій `exec()`, як вихід із процесу.

- Які результати тепер показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.

Значення глобальної змінної та змінної, що визначена у стеку співпадають, що у батьківського, що у дочірнього процесу. Це все через те що процеси виконувались не

одночасно, і процес-нащадок змінив ці змінні перед тим, як передати керування батьківському процесу. Саме тому на практиці, з метою обмеження на користування, дочірній процес не може змінювати ніякі глобальні змінні або навіть загальні змінні, що розділяються з батьківським процесом.

7. Тепер додайте виклик `exec()` у код процесу-нащадка. Для початку використайте простішу функцію `execl()`. Варіант виклику на прикладі утиліти `ls`:

↓ У наведеному прикладі передаються аргументи командного рядка.

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

```
nazar@ubuntu:~/OS/lab_6$ nano myforktest.cpp
pid_t pID = vfork();
if (pID == 0) // child
{
    // Code only executed by child process
    int exec();
    execl("/bin/ls", "/bin/ls", "-a", "-l", (char *) 0);

    sIdentifier = "Child Process: ";
    globalVariable++;
    iStackVariable++;
}
```

```
nazar@ubuntu:~/OS/lab_6$ g++ -o myforktest myforktest.cpp
nazar@ubuntu:~/OS/lab_6$ ./myforktest
Parent Process: Global variable: 2 Stack variable: 20
nazar@ubuntu:~/OS/lab_6$ total 32
drwxrwxr-x 2 nazar nazar 4096 Mar 28 00:45 .
drwxrwxr-x 7 nazar nazar 4096 Mar 27 21:24 ..
-rwxrwxr-x 1 nazar nazar 18336 Mar 28 00:45 myforktest
-rw-rw-r-- 1 nazar nazar 1065 Mar 28 00:45 myforktest.cpp

nazar@ubuntu:~/OS/lab_6$
```

8. Проведіть експерименти з викликом різних програм, у тому числі `ps`, `bash`, а також з викликами `execl()` у батьківському процесі.

```
nazar@ubuntu:~/OS/lab_6$ nano myforktest.cpp
else // parent
{
    // Code only executed by parent process
    int exec();
    execl("/bin/ps", "/bin/ps", "-u", (char *) 0);

    sIdentifier = "Parent Process:";
}
```

```

nazar@ubuntu:~/OS/lab_6$ g++ -o myforktest myforktest.cpp
nazar@ubuntu:~/OS/lab_6$ ./myforktest
Child Process: Global variable: 3 Stack variable: 21
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
nazar    1454  0.0  0.1 172652   2928 tty2     Ssl+  Mar27    0:00 /usr/lib/gd
nazar    1463  0.6  1.8 319328  37464 tty2     Sl+   Mar27    8:31 /usr/lib/xo
nazar    1508  0.0  0.2 199236   4120 tty2     Sl+   Mar27    0:00 /usr/libexe
nazar    34610  0.0  0.1  10988   3060 pts/0    Ss    Mar27    0:01 bash
nazar    71519  0.0  0.2   9976   4200 pts/0    T     Mar27    0:00 nano myfork
nazar    72440  0.0  0.1  11736   3592 pts/0    R+    00:15    0:00 /bin/ps -u
nazar    72441  0.0  0.0      0      0 pts/0    Z+    00:15    0:00 [myforktest
nazar@ubuntu:~/OS/lab_6$

```

- Як запустити фоновий процес-нащадок?

Можна запустити фоновий процес-нащадок використовуючи при цьому **waitpid()** (або будь-яка функція очікування), щоб вловити сигнал про закінчення його виконання.

- Як процес-нащадок дізнається власний **PID**?

За допомогою функції **getpid()**, яка повертає ідентифікатор процесу, що викликає її.

- **PID** батьківського процесу?

За допомогою функції **getppid()**, яка повертає ідентифікатор процесу батьківського процесу, що викликає її.

## Висновки:

Під час виконання лабораторної роботи я засвоїв основні базові аспекти створення процесів у Linux із застосуванням системних викликів **fork()**, **vfork()** і **exec()**. Я зрозумів, що системні виклики **vfork()** і **fork()** відрізняються тим, що перший використовує спільний адресний простір для процесів, і тому такі процеси будуть виконуватися послідовно, що може більше затратити час на виконання, але економить більше ресурсів пам'яті. Щодо виклику функцій **exec()**, то після їх виконання процес завершує свою роботу навіть якщо ще були якісь команди в фрагменті коду.

Отже, після виконання та аналізу усіх завдань цієї роботи я оволодів практичними навичками застосування системних викликів у програмах, та ознайомився із цим самим механізмом створення процесів у UNIX-подібних системах.