Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий Фізико-технічний інститут

ОПЕРАЦІЙНІ СИСТЕМИ

Комп'ютерний практикум
Робота №6

Виконав:
студент групи ФІ-12
Завалій Олександр
Перевірив:
Кірієнко О.В.

Київ-2023

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.

Варіант №5

Зміст індивідуального завдання:

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

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

3. Запустіть програму myforktest. У якій послідовності виконуються батьківський процес і процес-нащадок? Чи завжди цей порядок дотримується?

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

5. Додайте цикл, який забезпечить кількаразове повторення дій після виклику fork(). Які результати показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.

6. 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.

7. 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

8. Проведіть експерименти з викликом різних програм, у тому числі ps, bash, а також з викликами execl() у батьківському процесі. Як запустити фоновий процес-нащадок? Як процес-нащадок дізнається власний PID? PID батьківського процесу?

9. Усі отримані результати і відповіді на запитання, які були задані вище, оформіть у вигляді протоколу.

Task I

Для початку можна взяти демонстраційну програму, запропоновану Greg Ippolito.

```
alex@Oleksandr:~/labs/lab_6$ nano myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ cat myforktest.cpp
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
        pid_t ret;
        ret = fork();
        if (ret == -1) {
        perror("Fork failed");
        exit(-1);
        }
        if (ret == 0) {
        printf("Child Process\n");
        printf("My pid %d\n", getpid());
        printf("My ppid %d\n", getppid());
        exit(0);
        } else {
        printf("Parent Process\n");
        printf("My pid %d\n", getpid());
        printf("My ppid %d\n", getppid());
        exit(0);
        }
        return 0;
}
```

Task II

Скомпілюйте програму
g++ -o myforktest myforktest.cpp

```
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ ls
myforktest  myforktest.cpp
alex@Oleksandr:~/labs/lab_6$
```

Task III

Запустіть програму myforktest. У якій послідовності виконуються батьківський процес і процес-нащадок? Чи завжди цей порядок дотримується?

```
alex@Oleksandr:~/labs/lab_6$ ./myforktest
Parent Process
My pid 4602
My ppid 1889
Child Process
My pid 4603
My ppid 4602
alex@Oleksandr:~/labs/lab_6$
```

«Parent Process» створює копію самого себе, тобто створює «Child Process». Після вони продовжують виконуватися паралельно. Таким чином, батьківський процес завжди продовжує виконуватись першим після виклику fork(). Процес-нащадок розпочинає своє виконання паралельно з батьківським процесом з того місця програмного коду, де була викликана команда.

Task IV

Додайте затримку у виконання одного або обох з цих процесів (функція sleep(), аргумент — затримка у секундах). Чи змінились результати виконання?

```
alex@Oleksandr:~/labs/lab_6$ cat myforktest.cpp
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
        pid_t ret;
        ret = fork();
        if (ret == -1) {
        perror("Fork failed");
        exit(-1);
        }
        if (ret == 0) {
        printf("Child Process\n");
        printf("My pid %d\n", getpid());
        printf("My ppid %d\n", getppid());
        exit(0);
        } else {
        sleep(5);
        printf("Parent Process\n");
        printf("My pid %d\n", getpid());
        printf("My ppid %d\n", getppid());
        exit(0);
        }
        return 0;
}
```

```
alex@Oleksandr:~/labs/lab_6$ ./myforktest
Child Process
My pid 21577
My ppid 21576
Parent Process
My pid 21576
My ppid 1889
alex@Oleksandr:~/labs/lab_6$
```

Затримка у виконанні одного з процесів не вплине на порядок запуску процесів. Батьківський процес все одно розпочне виконання першим, а процес-нащадок - після того, як батьківський процес створить його за допомогою функції fork(). Проте, якщо функція sleep() викликається в процесі-нащадку, то процес-нащадок буде зупинений на визначений період часу, що може вплинути на загальний час виконання батьківського процесу, оскільки батьківський процес буде чекати, доки процес-нащадок відновить свою роботу після затримки.

Task V

Додайте цикл, який забезпечить кількаразове повторення дій після виклику fork(). Які результати показують процеси (значення глобальної змінної і змінної, що визначена у стеку)? Поясніть.

```
alex@Oleksandr:~/labs/lab_6$ nano myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ cat myforktest.cpp
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;
int global_var = 0;

int main () {
    string processN;
    int local_var = 0;
    pid_t ret = fork();
    for(int i = 0; i < 3; i++) {
        if (ret == -1) {
            perror("Fork failed");
            exit(-1);
        }
        if (ret == 0) {
            global_var++;
            local_var++;
            processN = "Child";
        } else {
            processN = "Parent";
        }
    }
    cout<<"Process: "<<processN<<endl;
    cout<<"lobal varible: "<<global_var<<endl;
    cout<<"Local variable value: "<<local_var<<endl;
    return 0;
}
```

```
alex@Oleksandr:~/labs/lab_6$ ./myforktest
Process: Parent
lobal varible: 0
Local variable value: 0
Process: Child
lobal varible: 3
Local variable value: 3
alex@Oleksandr:~/labs/lab_6$
```

Для батьківського процесу будемо мати порожні значення змінних. Після створення дочірнього процесу він пробіжить по циклу три рази й додасть до цих змінних по одиниці за один прохід. Тому пілся виконання коду бачимо, що значення в дочірньому процесі змінились від нуля до трьох.

Task VI

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.

```
alex@Oleksandr:~/labs/lab_6$ nano myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ cat myforktest.cpp
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;
int global_var = 0;

int main () {
    string processN;
    int local_var = 0;
    pid_t ret = vfork();
        if (ret == -1) {
            perror("Fork failed");
            exit(-1);
        }
        if (ret == 0) {
            global_var++;
            local_var++;
            processN = "Child";
        } else {
            processN = "Parent";
        }
    cout<<"Process: "<<processN<<endl;
    cout<<"lobal varible: "<<global_var<<endl;
    cout<<"Local variable value: "<<local_var<<endl;
    return 0;
}
```

```
alex@Oleksandr:~/labs/lab_6$ ./myforktest
Process: Child
lobal varible: 1
Local variable value: 1
Process: Parent
lobal varible: 1
Local variable value: 1
*** stack smashing detected ***: terminated
Aborted (core dumped)
alex@Oleksandr:~/labs/lab_6$
```

fork() створює новий процес шляхом копіювання вмісту батьківського процесу (включаючи вміст змінних). vfork() створює новий процес, але він використовує спільні змінні з батьківським процесом без його копіювання. Також виклик vfork() блокує батьківський процес до тих пір, поки дочірній процес не викличе одну з функцій з exec.

Так, виникає помилка "stack smashing detected: terminated. Aborted (core dumped)". vfork() створює дочірній процес, який не завершує своє виконання, тому отримали помилку переповнення стеку. Для вирішення цієї проблеми достатньо завершити виконання дочірнього процесу за допомогою команди exit(0);

```
alex@Oleksandr:~/labs/lab_6$ nano myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ cat myforktest.cpp
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;
int global_var = 0;

int main () {
    string processN;
    int local_var = 0;
    pid_t ret = vfork();
    if (ret == -1) {
    perror("VFork failed");
    exit(-1);
    }
    if (ret == 0) {
        global_var++;
        local_var++;
        processN = "Child";
    } else {
        processN = "Parent";
    }
    cout<<"Process: "<<processN<<endl;
    cout<<"lobal varible: "<<global_var<<endl;
    cout<<"Local variable value: "<<local_var<<endl;
    exit(0);
}
```

```
alex@Oleksandr:~/labs/lab_6$ ./myforktest
Process: Child
lobal varible: 1
Local variable value: 1
Process: Parent
lobal varible: 1
Local variable value: 1
alex@Oleksandr:~/labs/lab_6$
```

Оскільки дочірній процес було створено командою vfork(), можемо побачити, що він змінив значення глобальної та локальної змінної на одиницю. Оскільки адресний простір вони мають однаковий, тому і значення для двух процесів рівні.

Task VII

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);

```
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ ./myforktest
Process: Parent
alex@Oleksandr:~/labs/lab_6$ total 32
drwxrwxr-x 2 alex alex  4096 кві  8 00:05 .
drwxrwxr-x 9 roma alex  4096 кві  6 11:49 ..
-rwxrwxr-x 1 alex alex 17360 кві  8 00:05 myforktest
-rw-rw-r-- 1 alex alex   740 кві  8 00:05 myforktest.cpp

alex@Oleksandr:~/labs/lab_6$
```

Task VIII

Проведіть експерименти з викликом різних програм, у тому числі ps, bash, а також з викликами execl() у батьківському процесі. Як запустити фоновий процес-нащадок? Як процес-нащадок дізнається власний PID? PID батьківського процесу?

```
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ ./myforktest
PID: 8104
PPID: 8103
Process: Child
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 кві07 ?     00:00:05 /sbin/init splash
root           2       0  0 кві07 ?     00:00:00 [kthreadd]
root           3       2  0 кві07 ?     00:00:00 [rcu_gp]
root           4       2  0 кві07 ?     00:00:00 [rcu_par_gp]
root           5       2  0 кві07 ?     00:00:00 [slub_flushwq]
```

```
alex@Oleksandr:~/labs/lab_6$ g++ -o myforktest myforktest.cpp
alex@Oleksandr:~/labs/lab_6$ ./myforktest
PID: 8119
PPID: 8118
Process: Child
Hello, World!
alex@Oleksandr:~/labs/lab_6$
```

```
alex@Oleksandr:~/labs/lab_6$ cat myforktest.cpp
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
using namespace std;

int main () {
    string processN;
    pid_t ret = fork();
    if (ret == -1) {
    perror("Fork failed");
    exit(-1);
    }
    if (ret == 0) {
        processN = "Child";
        cout<<"PID: "<<getpid()<<endl;
        cout<<"PPID: "<<getppid()<<endl;
    } else {
        processN = "Parent";
        //execl("/bin/ls", "/bin/ls", "-a", "-l", (char *) 0);
        //execl("/bin/ps", "/bin/ps", "-ef", (char *) 0);
        execl("/bin/bash", "/bin/bash", "-c", "echo 'Hello, World!'", (char *) 0);
    }
    cout<<"Process: "<<processN<<endl;
    exit(0);
}
alex@Oleksandr:~/labs/lab_6$
```

Щоб перевести процес-нащадок в фоновий режим, можемо скористатись такими командами:

1. Використання символу "&"в кінці команди.
   ./my_program &

2. Використання команди "nohup".
   nohup ./my_program &

Новий процес-нащадок, який створюється з батьківського процесу, отримує свій власний унікальний ідентифікатор процесу (PID), який відмінний від PID процесу-батька. Проте PPID він буде мати такий самий, як і PID батьківського процесу.

Висновки

Процес забезпечує взаємодію між програмами в UNIX-подібних системах. Системні виклики fork() і exec() дозволяють створювати нові процеси та керувати їх роботою.

Системний виклик fork() дозволяє створювати дочірні процеси, які є копіями батьківського процесу. Дочірній процес має власну пам'ять і ресурси.

Системний виклик exec() дозволяє замінити вміст поточного процесу новим процесом. Тобто дозволяє виконувати нові програми в поточному процесі, зберігаючи при цьому його ідентифікатор процесу (PID) і ресурси.

Комбінація викликів fork() і exec() дозволяє створювати програми, які можуть комбінувати різні функції і взаємодіяти між собою.

Важливо також звернути увагу на обробку помилок при використанні системних викликів fork() і exec(). Оскільки неправильна послідовність викликів може призвести до втрати даних або некоректної роботи програми.

Узагальнюючи, розуміння процесів дозволяє забезпечити ефективну взаємодію між різними компонентами програми та оптимальне використання ресурсів системи. Ці навички також можуть бути корисними при розробці системних додатків, демонів, серверів та інших програм, де потрібно контролювати багатопроцесову роботу.