



Тема: «Основи програмного керування процесами в *Unix*-подібних ОС»

Викладач: Олександр А. Блажко,

доцент кафедри ІС Одеської політехніки, [blazhko@ieee.org](mailto:blazhko@ieee.org)

**Мета роботи:** отримання навичок в керування процесами в *Unix*-подібних ОС  
з використанням мови програмування *C*.

## План.

### 1 Теоретичні відомості

### 2 Завдання

#### 1 Теоретичні відомості

##### 1.1 Історія взаємовідносин мови програмування *C* та ОС *UNIX*

Відомо, що мова програмування *C* була створена майже одночасно із ОС *Unix*:

- 1971 рік, з'явилась перша редакція ОС *UNIX*, яка була написана на мові програмування *Assembler* (мова *A*), але вже включала мову програмування *B* як спадкоємця першого інтерпретатору компілюючого типу з мови програмування *BCPL* (*Basic Combined Programming Language*);
- 1972 рік, з'явилась 2-я редакція ОС *UNIX* для міні-комп'ютера *PDP-11*, яка була вже повністю переписана на мові програмування *B*;
- 1973 рік, з'явилась 3-тя редакція ОС *UNIX* із вбудованим компілятором мови програмування *C* як розвиток мови програмування *B* але із типізацією змінних;
- 1973 рік, вийшла 4-а редакція ОС *UNIX*, вже частково переписана на мові програмування *C*;
- 1974 рік, вийшла 5-та редакція ОС *UNIX*, вже повністю переписана на мові програмування *C*.

##### 1.2 *GCC - General Public License Compiler Collection*

В *Unix*-подібних ОС для компіляції програм часто використовується *GCC*-набір (*GCC - General Public License Compiler Collection*) – набір компіляторів для різних мов програмування з так званою публічною ліцензією використання, яка стала другою після появи *BSD license* (*Berkeley Software Distribution license*).

Спочатку так називався компілятор з мови програмування C. Пізніше він був розширений для підтримки мов C++, Fortran, Java, Ada, Objective-C та інших.

Компіляція програми на мові програмування C за допомогою команди gcc передбачає такі послідовні етапи:

- препроцесінг (*preprocessing*) або попередня обробка програмного коду;
- компіляція (*compilation*) або трансляція програмного коду у код на мові програмування *Assembler*;
- збирання (*assembly*) або трансляція програмного коду у машинні команди;
- лінковка (*linking*) або встановлення зв'язків із зовнішніми функціями динамічними програмними бібліотеками.

Але для швидкої компіляції програм нема потреби проходити кожний з етапів окремо, достатньо вказати опцію `-o` для визначення назви *executable*-файлу:

```
gcc файл.c -o файл
```

Компілятор автоматично викликає усі програми, які обслуговують вказані етапи, про що можна дізнатися додавши опцію `-v` до команди `gcc`.

Якщо не вказати опцію `-o`, тоді буде створено *executable*-файл з назвою *a.out*

### 1.3 C-функції перегляду ідентифікаторів процесів та їх груп

#### 1.3.1 C-функції перегляду ідентифікатору процесу та користувача

В таблиці 1 наведено приклади функцій перегляду ідентифікаторів процесу та користувачів, які його запустили.

Для функцій заголовочним файлом є файл `<unistd.h>`

Таблиця 1 – Приклади функцій

Назва функції	Опис роботи функції
<code>pid_t getpid (void);</code>	повертає <i>PID</i> процесу, який викликав цю функцію
<code>pid_t getppid (void);</code>	повертає <i>PPID</i> для вказаного процесу, який викликав цю функцію
<code>uid_t getuid (void);</code>	повертає ідентифікатор користувача процесу ( <i>UID – User ID</i> ), який викликав цю функцію
<code>gid_t getgid (void);</code>	повертає ідентифікатор групи користувача ( <i>GID – Group ID</i> ), який викликав цю функцію

На рисунку 1 наведено приклад використання функцій з таблиці 1.

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

int main (void) {
    fprintf(stdout, "I am process with pid=%d\n", getpid());
    fprintf(stdout, "My parent pid=%d\n", getppid());
    fprintf(stdout, "My user id=%d\n", getuid());
    fprintf(stdout, "My group id=%d\n", getgid());
    return 0;
}
```

Рис. 1 – Приклади використання функцій отримання *PID*, *PPID*, *UID*, *GUID*

Результат виконання програми з представленим програмним кодом наведено на рисунку 2.

```
[blazhko@vps6iefe OS-LabWork9-Examples]$ gcc info.c -o info
[blazhko@vps6iefe OS-LabWork9-Examples]$ ./info
I am process with pid=2876952
My parent pid=2876588
My user id=1188
My group id=1188
```

Рис. 2 – Результат виконання програми

### 1.3.2 С-функції роботи з групами процесів

Відомо, що для зменшення трудомісткості керування великою кількістю будь-яких об'єктів часто рекомендується об'єднувати їх в логічні групи, після чого почати керувати окремими групами. Так, наприклад, відбувається під час керування правами доступу до файлів збоку користувачів та їх додаткових груп (група власника, всі інші користувачі). Коли виконання якогось складного завдання вимагає створення великої кількості процесів, їх також об'єднують у логічні групи. Тому в *Unix*-подібній ОС процеси мають не тільки свій *PID*, але й належать до відповідних груп процесів.

*Група процесів* – це набір з одного або більше процесів, зазвичай пов'язаних з виконанням групового завдання, які можуть приймати сигнали від одного псевдотерміналу. Кожна група процесів має унікальний ідентифікатор. Ідентифікатор групи процесів дуже нагадує ідентифікатор процесу: це ціле число, що зберігається у змінній типу *pid\_t*.

Найчастіше процеси об'єднуються у групу командною оболонкою під час конвеєрної обробки даних. Наприклад, виконаємо наступну послідовність команд:

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

В наведеному прикладі буде створено дві групи процесів:

- група процесів для команд *proc1*, *proc2*;

– група процесів для команд *proc3*, *proc4*, *proc5*.

Функція *getpgrp* повертає ідентифікатор групи процесів, до якої належить поточний процес, який викликав цю функцію:

```
pid_t getpgrp(void);
```

На рисунку 3 наведено приклад використання функції *getpgrp*.

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

int main(void) {
    fprintf(stdout, "I am process with pid=%d\n", getpid());
    fprintf(stdout, "My group process id =%d\n", getpgrp());
    fprintf(stdout, "My parent pid=%d\n", getppid());
    return 0;
}
```

Рис. 3 – Фрагмент програми *info\_grp* з функцією *getpgrp*

На рисунку 4 наведено приклад виклику програми *info\_grp*, де видно, що при паралельному виконанні двох процесів програми процеси мають різні ідентифікатори групи. Але при конвеєрному виконанні двох процесів програми значення ідентифікаторів групи співпадають, тому що процеси об'єднано в єдину групу процесів. В конвеєрі перша команда запускається з перенаправленням *stdout*-потoku на *stderr*-потік, щоб можна було побачити її повідомлення, інакше повідомлення автоматично підуть до другої програми.

```
[blazhko@vps6iefe OS-Labwork9-Examples]$ ./info_grp & ./info_grp &
[1] 2551269
[2] 2551270
[blazhko@vps6iefe OS-Labwork9-Examples]$ I am process with pid=2551270
My group process id =2551270
My parent pid=2526523
I am process with pid=2551269
My group process id =2551269
My parent pid=2526523
```

(a) паралельне виконання двох процесів програми *info\_grp*

```
[blazhko@vps6iefe OS-Labwork9-Examples]$ ./info_grp 1>&2 | ./info_grp
I am process with pid=2184298
My group process id =2184297
My parent pid=2174535
I am process with pid=2184297
My group process id =2184297
My parent pid=2174535
[blazhko@vps6iefe OS-Labwork9-Examples]$ |
```

(b) конвеєрне виконання двох процесів програми *info\_grp*

Рис. 4 – Результати виконання програми *info\_grp*

## 1.4 C-функції керування процесами

### 1.4.1 C-функції породження *child*-процесу

В *Unix*-подібних ОС створення *child*-процесу має декілька етапів:

- 1) породження *child*-процесу як програмної копії *parent*-процесу;
- 2) надання особливостей *child*-процесу, як нового вмісту програмного коду.

На першому етапі *parent*-процес повинен «породити» *child*-процес, тобто створити копію свого *програмного образу* (програмного коду в оперативній пам'яті) з копією дескрипторів відкритих файлів. Копіювання дескрипторів відкритих файлів у *child*-процес дозволяє в подальшому забезпечити міжпроцесну взаємодію між *parent*-процесом та *child*-процесом, коли *OUTPUT*-потік *parent*-процесу зв'язується з *INPUT*-потокм *child*-процесом, наприклад, під час:

- перенаправлення потоків даних;
- конвеєризації команд.

*Parent*-процес породжує *child*-процес, використовуючи функцію:

```
pid_t fork (void);
```

Функція *fork()* після виклику повертає у *parent*-процес значення *PID* створеного *child*-процесу, а у *child*-процес функція повертає 0. Ця особливість дозволяє в програмі розділити дії *parent*-процесу та *child*-процесу.

На рисунку 5 представлено приклад програми *fork* з функцією ініціалізації *child*-процесу. *Child*-процес може дізнатися про значення свого *PID* через виклик функції *getpid()*, а про значення *PPID* - через виклик функції *getppid()*.

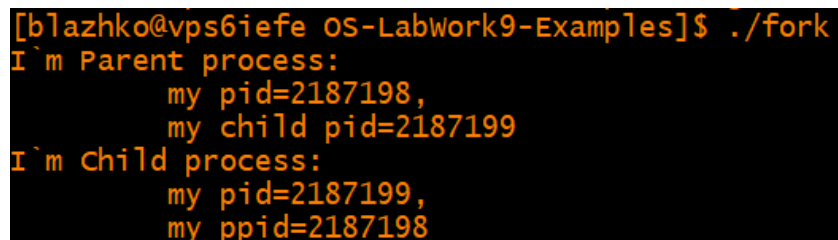
Програмний код обробки *parent*-процесу завершується викликом функції *sleep(1)*, яка вносить секундну затримку, щоб він не завершився раніше *child*-процесу.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
    pid_t pid = fork();
    if (pid == 0) {
        printf("I'm Child process:\n");
        printf("\t my pid=%d, \n\t my ppid=%d\n",getpid(),getppid());
    }
    else {
        printf("I'm Parent process:\n");
        printf("\t my pid=%d, \n\t my child pid=%d\n",getpid(),pid);
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

Рис. 5 – Приклад C-програми з функцією ініціалізації *child*-процесу

Після запуску скомпільованого файлу *fork* будуть отримані одразу два повідомлення, як показано на рисунку 6, які вказують на те, що вже запущено два паралельні процеси (*parent*-процес та *child*-процес), кожен з яких вивів своє власне повідомлення в залежності від значення, що повертається функцією *fork*.



```
[b1azhko@vps6iefe OS-Labwork9-Examples]$ ./fork
I`m Parent process:
    my pid=2187198,
    my child pid=2187199
I`m Child process:
    my pid=2187199,
    my ppid=2187198
```

Рис. 6 – Приклад повідомлень запуску програми *fork*

Після породження *child*-процесу для нього встановлюється ідентифікатор групи процесів, який співпадає зі значенням його *PID*.

Для того, щоб *parent*-процес міг призначити ідентифікатор групи процесів створеному *child*-процесу, а *child*-процес міг встановити свій власний ідентифікатор групи процесів, використовується функції *setpgid*:

```
int setpgid(pid_t pid, pid_t pgid),
```

де *pid* – значенням його *PID*;

*pgid* – ідентифікатор групи процесів.

Функція має наступні особливості використання:

- функція повертає 0 в разі успіху, -1 – в разі помилки;
- якщо аргументи мають однакові значення, то процес, заданий ідентифікатором *pid*, стає лідером групи процесів;
- якщо в аргументі *pid* передається значення 0, тоді ідентифікатор процесу групи співпадає з ідентифікатором процесу, який викликав функцію;
- якщо ж в аргументі *pgid* передається значення 0, то як ідентифікатор групи процесів використовується значення аргументу *pid*.
- *parent*-процес може встановити ідентифікатор групи процесів тільки для себе самого та для своїх *child*-процесів;
- але *parent*-процес не зможе змінити ідентифікатор групи процесів *child*-процесу, який вже викликав одну з функцій заміни програмного образу *child*-процесу.

На рисунку 7 представлено приклад програми *fork\_pgrp* з функцією встановлення нового ідентифікатора групи для *child*-процесу.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
    pid_t pid;
    if ((pid = fork()) < 0) {
        fprintf(stderr, "fork error!");
        return EXIT_FAILURE;
    }
    if (pid == 0) {
        printf("I`m child process:\n");
        printf("\t my pid=%d, \n\t my ppid=%d\n", getpid(), getppid());
        printf("\t my group process id = %d\n", getpgrp());
    }
    else {
        printf("I`m Parent process:\n");
        printf("\t my pid=%d, \n\t my child pid=%d\n", getpid(), pid);
        printf("\t my group process id = %d\n", getpgrp());
        if (setpgid(pid, getpid()) != 0) {
            fprintf(stderr, "setpgid error!");
            return EXIT_FAILURE;
        }
        sleep(1);
    }
    return EXIT_SUCCESS;
}

```

Рис. 7 – Приклад C-програми *fork\_pgrp* з функцією встановлення нового ідентифікатора групи для *child*-процесу

Після запуску скомпільованого файлу *fork\_pgrp* будуть отримані одразу два повідомлення, як показано на рисунку 8:

- *parent*-процес виконується в своїй групі процесів, ідентифікатор якої співпадає з ідентифікатором *parent*-процесу;
- *child*-процес виконується у групі *parent*-процесу.

```

[b]lazhko@vps6iefe OS-LabWork9-Examples]$ ./fork_pgrp
I`m Parent process:
    my pid=2189943,
    my child pid=2189944
    my group process id = 2189943
I`m child process:
    my pid=2189944,
    my ppid=2189943
    my group process id = 2189943

```

Рис. 8 – Приклад повідомлень запуску програми *fork\_2*

Якщо замінити виклик функції *setpgid(pid, getpid())* на *setpgid(pid, 0)*, тоді для *child*-процесу створено свою групу процесів, ідентифікатор якої буде співпадати з ідентифікатором *child*-процесу.

#### 1.4.2 C-функції заміни програмного образу процесу

Після породження *child*-процесу необхідно замінити його програмний образ, тобто замінити програмний код роботи *child*-процесу, який вже не буде співпадати з програмним кодом *parent*-процесу. Як вказувалося раніше, такий двоетапний спосіб запуску нової програми пов'язаний з необхідністю налаштовувати *INPUT/OUTPUT* потоки, які в *Unix*-подібних ОС є файлами, успадкованими від *parent*-процесу. Завдяки цьому забезпечується міжпроцесна взаємодія через конвеєр, коли дескриптор файлу *INPUT*-потoku *parent*-процесу пов'язується з дескриптором файлу *INPUT*-потoku *child*-процесу.

Для заміни образу процесу використовується одна з сімєства *exec*-функцій, які оголошені у файлі *unistd.h*:

```
int execl(char *name, char *arg0, ... /*NULL*/);
int execv(char *name, char *argv[]);
int execlp(char *name, char *arg0, ... /*,NULL, char *envp[] */);
int execve(char *name, char *argv[], char *envp[]);
int execlp(char *name, char *arg0, ... /*NULL*/);
int execvp(char *name, char *argv[]);
```

Суфікси *l*, *v*, *p*, *e* в іменах функцій визначають формат, кількість аргументів функції та каталоги, в яких потрібно шукати файл з програмою для запуску:

- *l (list)*: аргументи функції передаються у вигляді списку *arg0, arg1.... argn, NULL*, тому цю функцію використовують, якщо кількість аргументів заздалегіть не відома;
- *v (vector)*: аргументи функції передаються у вигляді вектора (масиву) *argv[]*, де останній аргумент *argv [n]* має бути *NULL*, тому цю функцію використовують, якщо кількість аргументів заздалегіть відома;
- *p (path)*: файл для виконання шукається у поточному каталозі та у всіх каталогах, визначених змінною середовища *PATH*;
- *e (environment)*: функція чекає на список змінних середовища у вигляді вектора (масива) *envp []* і не використовує поточне середовище.

Функція *execve()*:

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



Розглянемо функцію `execve()`, яка замінює поточний образ процесу програмою з файлу з ім'ям *path*, набором аргументів *argv* і оточенням *envp*.

При цьому:

- *path* - це не просто ім'я програми, а шлях до неї, наприклад, щоб виконати команду *ls*, потрібно в першому аргументі вказати `"/bin/echo"`;
- масиви рядків *argv* та *environ* обов'язково повинні закінчуватися елементом *NULL*;
- перший елемент масиву *argv*, *argv[0]* – це ім'я програми *child*-процесу, а аргументи командного рядку програми розміщуються в елементах, починаючи з *argv[1]*.

У разі успішного завершення функція `execve()` нічого не повертає, оскільки нова програма отримує повне і безповоротне керування поточним процесом. Але якщо сталася помилка, тоді продовжується виконання *parent*-процесу та, за традицією, програма завершується помилкою, повертаючи -1 (константа `EXIT_FAILURE`).

На рисунку 9 представлено приклад програми *exec*, яка замінює свій образ іншою програмою, наприклад *echo*, використовуючи функцію `execv`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
    char* echo_args[] = { "echo",
                          "I`m ECHO - a new image of child process.\n",
                          NULL
                        };

    pid_t pid = fork();
    if (pid != 0) {
        printf("I`m parent!\n");
        execv("/bin/echo", echo_args);
        printf("exec error!");
        return EXIT_FAILURE;
    }
    else
        printf("I`m child!\n");
    return EXIT_SUCCESS;
}
```

Рис. 9 – Приклад програми, яка замінює образ *child*-процесу програмою *echo*

Після запуску скомпільованого файлу *exec* будуть отримані одразу два повідомлення, як показано на рисунку 10. Але повідомлення «*I`m child!*» з'являється пізніше через окремі затримки у процесах, тому для проведення експериментів рекомендується перед функцією `execv` додати функцію затримки, наприклад, `sleep(1)`.

```
[blazhko@vps6iefe OS-Labwork9-Examples]$ gcc exec.c -o exec
[blazhko@vps6iefe OS-Labwork9-Examples]$ ./exec
I`m parent!
I`m ECHO - a new image of child process.

[blazhko@vps6iefe OS-Labwork9-Examples]$ I`m child!
```

Рис. 10 – Приклад повідомлень запуску програми *exec*

### 1.5 C-функції керування сигналами

Функція *kill* надсилає сигнал процесу або групі процесів:

```
int kill (pid_t pid, int signo);
```

Функція повертає 0 в разі успіху, -1 – в разі помилки.

Інтерпретація аргументу *pid* функції *kill* здійснюється відповідно до наступних правил, наведених в таблиці 2.

Таблиця 2 - Інтерпретація значень аргументу *pid* функцією *kill*

Значення аргументу <i>pid</i>	Інтерпретація
$pid > 0$	Сигнал надсилається процесу із ідентифікатором <i>pid</i>
$pid == 0$	Сигнал надсилається всім процесам з таким же ідентифікатором групи процесів, що має поточний процес, який надсилає сигнал. Зверніть увагу, що до поняття “всі процеси” не входять визначені реалізацією системні процеси. У більшості версій <i>UNIX</i> під системними розуміють процеси ядра й процес <i>init</i> (ідентифікатор процесу 1)
$pid < 0$	Сигнал надсилається всім процесам з ідентифікатором групи процесів, який дорівнює абсолютному значенню <i>pid</i> , яким цей процес має право надсилати сигнали. Знову ж до поняття “всі процеси” не входять визначені реалізацією системні процеси
$pid == -1$	Сигнал надсилається всім процесам у системі, яким процес, що надсилає сигнал, має право надсилати сигнали. Тут так само з поняття “всі процеси” виключаються деякі системні процеси

Як ми вже згадували, процес повинен мати певні права, щоб надсилати сигнали іншим процесам. Так адміністратор ОС може надіслати сигнал будь-якому процесу. В інших випадках необхідно дотримуватися основного правила: реальний ідентифікатор користувача процесу, що надсилає сигнал, повинен збігатися з реальним ідентифікатором користувача процесу, що приймає сигнал.

Якщо в результаті виклику функції *kill* генерується сигнал для процесу, який викликає її, та при цьому сигнал не заблокований, тоді або сигнал з номером *signo*, або інший сигнал, що очікує оброблення, буде доставлений процесу ще до того, як функція *kill* поверне керування.

На рисунку 11 показано приклад програми *kill* з викликом функції *kill*, параметром якої є сигнал *SIGKILL* для зупинки процесу з *PID*, який передається через аргумент командного рядку.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    pid_t pid;
    if(argc == 1) {
        fprintf(stderr,"Usage: %s <pid>\n",argv[0]);
        exit(1);
    }
    pid = (pid_t) atoi(argv[1]);
    if (!kill(pid,SIGKILL))
        printf("Process with PID=%d was killed!\n",pid);
    else
        fprintf(stderr,"Kill error for pid=%d\n",pid);
    return 0;
}
```

Рис. 11 – Приклад програми *kill* з викликом функції *kill*

Для обробки сигналів, надісланих процесу, використовується функція *signal*:

```
void* signal(int signo, void (*func)(int))
```

Функція повертає попередню диспозицію сигналу в разі успіху, *SIG\_ERR* – в разі помилки.

Аргумент *signo* – це ім'я сигналу.

Як аргумент *func* можна передавати:

- константу *SIG\_IGN* як повідомлення системі, що сигнал повинен ігноруватися, окрім двох сигналів *SIGKILL* та *SIGSTOP*;
- константу *SIG\_DFL* як повідомлення системі, що з сигналом зв'язується дія за замовчуванням;
- адресу функції, яка буде викликана в момент отримання сигналу.

На рисунку 12 показано приклад програми *signal\_ignor* з викликом функції *signal*, яка забезпечить для процесу ігнорування сигналу *SIGINT* через комбінацію клавіш *Ctrl+C* або через виклик команди *kill -9 <pid>*

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
int main(){
    int i;
    signal(SIGINT,SIG_IGN); // ігнорувати сигнал Ctrl+C = kill -2
    for(i=1;;i++){
        printf("%d : Робоча ітерація процесу\n",i);
        sleep(1);
    }
    return EXIT_SUCCESS;
}

```

Рис. 12 – Приклад програми *signal\_ignor* з викликом функції *signal*

Після запуску скомпільованого файлу *signal\_ignor* буде створено процес, який:

- не можливо зупинити через комбінацію клавіш *Ctrl+C* (сигнал *interrupt*, код = 2)
- можна зупинити через комбінацію клавіш *Ctrl+\* (сигнал *quit*, код = 3), як показано на рисунку 13.

```

[oracle@vpsj3IeQ Lab8]$ gcc signal_ignor.c -o signal_ignor
[oracle@vpsj3IeQ Lab8]$ ./signal_ignor
1 : Робоча ітерація процесу
2 : Робоча ітерація процесу
3 : Робоча ітерація процесу
^C4 : Робоча ітерація процесу
^C5 : Робоча ітерація процесу
^C^C6 : Робоча ітерація процесу
7 : Робоча ітерація процесу
^\\Quit
[oracle@vpsj3IeQ Lab8]$

```

Рис. 13 – Приклад повідомлень запуску програми *signal\_ignor*

Якщо при виклику функції *signal* у її другому параметрі вказано адресу функції, то вона буде викликатися в момент отримання сигналу, тобто буде “перехоплювати” сигнал. Такі функції називають оброблювачами або перехоплювачами сигналів.

Для користувача зареєстровано два сигнали: *SIGUSR1*, *SIGUSR2*

На рисунку 14 наведено приклад створення програми *signal\_get*, яка очікує сигнал *SIGUSR1*.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

/* функція-оброблювач сигналу SIGUSR1 */
static void sig_usr(int signo) {
    if (signo == SIGUSR1)
        printf("Отримано сигнал SIGUSR1\n");
}

int main(void) {
    // визначення реакції на сигнал SIGUSR1
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        fprintf(stderr, "Неможливо перехопити сигнал SIGUSR1\n");
    printf("Мій PID = %d. Чекаю на сигнал SIGUSR1\n", getpid());
    /* безкінечний цикл */
    for ( ; ; )
        // pause() припиняє процес поки не буде отримано сигнал
        pause();
    return EXIT_SUCCESS;
}

```

Рис. 14 – Приклад створення програми *signal\_get*, яка очікує сигнал *SIGUSR1*

Вказаний приклад програми *signal\_get* після компіляції необхідно запустити. На рисунку 15 наведено приклад програми *signal\_send*, яка надсилає процесу сигнал *SIGUSR1*.

```

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    pid_t pid;
    if(argc == 1) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        return EXIT_FAILURE;
    }
    pid = (pid_t) atoi(argv[1]);
    if (!kill(pid, SIGUSR1))
        printf("Send signal to process with PID=%d!\n", pid);
    else
        fprintf(stderr, "Error!\n");
    return EXIT_SUCCESS;
}

```

Рис. 15 – Приклад програми *signal\_send* з надсилання сигналу *SIGUSR1* процесу

В окремому псевдотерміналі необхідно визначити *pid* запущеного процесу *signal\_get* та виконати програму *signal\_send*, вказавши їй значення *pid*. Після запуску програма *signal\_send* надішле сигнал *SIGUSR1* процесу програми *signal\_get*, яка відреагує на цей сигнал відповідним повідомленням. Після запуску скомпільованого файлу *signal\_get* створено процес, який буде очікувати на сигнал *SIGUSR1*, як показано на рисунку 16. Після

запуску скомпільованого файлу `signal_send` створено процес, який надішле сигнал `SIGUSR1`.

```
blazhko@vps6iefe:~/OS-LabWork9-Examples
[blazhko@vps6iefe OS-LabWork9-Examples]$ gcc signal_get.c -o signal_get
[blazhko@vps6iefe OS-LabWork9-Examples]$ ./signal_get
Мій PID = 2541557. Чекаю на сигнал SIGUSR1
Отримано сигнал SIGUSR1

blazhko@vps6iefe:~/OS-LabWork9-Examples
[blazhko@vps6iefe OS-LabWork9-Examples]$ gcc signal_send.c -o signal_send
[blazhko@vps6iefe OS-LabWork9-Examples]$ ./signal_send 2541557
Send signal to process with PID=2541557!
[blazhko@vps6iefe OS-LabWork9-Examples]$
```

Рис. 16 – Приклад виконання програм `signal_get` та `signal_send`

## 1.6 С-функції керування завершенням процесів

У якийсь момент процеси повинні завершуватися. Питання тільки в тому, який завершиться першим: *parent*-процес або *child*-процес.

### 1.6.1 *Parent*-процес завершується несподівано раніше *child*-процесу

Традиційно *parent*-процес, запустивши *child*-процеси у звичайному режимі, чекає на їх завершення. Для скрипт-файлів, запущених у фоновому режимі, їх *parent*-процес може завершитися раніше, наприклад, як це видно з рисунку 17.

В цьому випадку *child*-процеси стають так званими «сиротами» (*orphan*-процеси).

Але «осиротілі» *child*-процеси повинні тримати новий *parent*-процес, бо у кожного процесу є *parent*-процес та можна повністю відстежити дерево процесів, починаючи з *PID* = 1, який завжди стає новим *parent*-процесом для «осиротілих» *child*-процесів.

```
#!/bin/bash
sleep 1
nice -n 5 ./e2.sh &
sleep 1
nice -n 5 ./e3.sh &
```

(а) приклад скрипт-файлу із запуском двох скрипт-файлів у фоновому режимі

PID	PPID	TT	STAT	NI	%CPU	CMD
358856	358855	pts/4	Ss	0	0.0	-bash
358895	358856	pts/4	R	0	39.2	/bin/bash ./e1.sh
358898	1	pts/4	RN	5	12.7	/bin/bash ./e2.sh
358902	1	pts/4	RN	5	12.7	/bin/bash ./e3.sh

(б) фрагмент таблиці з двома процесами, які отримали новий *parent*-процес, після того, як їх колишній *parent*-процес завершився раніше, а вони «осиротіли»

Рис. 17 – Приклади перевизначення *parent*-процесу для «осиротілих» *child*-процесів

Якщо *parent*-процес буде вимагає завершення своїх *child*-процесів, тоді він повинен надіслати цим процесам сигнал *HANGUP*.

На рисунку 18 показано приклад програми *child\_hangup*, в якій *parent*-процес надсилає своєму *child*-процесу сигнал *HANGUP*.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
int main (void)
{
    pid_t pid = fork();
    if (pid == 0) {
        printf("I`m Child process:\n");
        printf("\t my pid=%d, \n\t my ppid=%d\n",getpid(),getppid());
    }
    else {
        printf("I`m Parent process:\n");
        printf("\t my pid=%d, \n\t my child pid=%d\n",getpid(),pid);
        sleep(2);
        if (!kill(pid,SIGHUP))
            printf("Send signal to process with PID=%d!\n",pid);
        else
            fprintf(stderr,"Error!\n");
    }
    return EXIT_SUCCESS;
}
```

Рис. 18 – Приклад програми *child\_hangup* з надсилання сигналу *HANGUP*

На рисунку 19 представлено приклад програми, яка:

- з *parent*-процесу викликає *fork*;
- *parent*-процес чекає дві секунди з використанням функції *sleep*;
- *parent*-процес нестандартно завершується з використанням функції *\_exit(0)*;
- *child*-процес продовжує працювати, показуючи *PID* свого *parent*-процесу

протягом п'яти секунд.

*Примітка:* функція *\_exit* "негайно" завершує роботу програми:

- всі дескриптори файлів, які належать процесові, закриваються;
- всі його *child*-процеси отримують у якості нового *parent*-процесу процес з *PID*=1;
- процесу надсилається сигнал *SIGCHLD*.

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void)
{
    int i;
    pid_t pid = fork();
    if (pid != 0) {
        printf("I am parent with pid = %d. My child pid = %d\n",
               getpid(),pid);

        sleep(2);
        _exit(0);
    }
    else {
        for (i=0;i<5; i++) {
            printf("I am child with pid=%d. My parent pid = %d\n",
                   getpid(),getppid());
            sleep(1);
        }
    }
    return 0;
}

```

Рис. 19 – Приклад програми *orphan* із завершенням *parent*-процесу раніше *child*-процесу

В результаті виконання програми на екрані з'являться повідомлення, приклад яких представлено на рисунку 20.

```

[oracle@vpsj3IeQ Lab8]$ ./orphan
I am parent with pid = 13963. My child pid = 13964
I am child with pid=13964. My parent pid = 13963
I am child with pid=13964. My parent pid = 13963
[oracle@vpsj3IeQ Lab8]$ I am child with pid=13964. My parent pid = 1
I am child with pid=13964. My parent pid = 1
I am child with pid=13964. My parent pid = 1

```

Рис. 20 - Приклад виведення на екран роботи програми

Як видно на рисунку 20, коли *parent*-процес завершується, у *child*-процесі *PPID* змінюється на 1. Оскільки процес виконується у фоновому режимі, як тільки *parent*-процес завершується, керування повертається до командної оболонки, як видно із запрошення командної оболонки після четвертого рядку.

### 1.6.2 *Child*-процес завершується несподівано раніше *parent*-процесу

Розглянемо приклад ситуації, коли *child*-процес завершується несподівано для свого *parent*-процесу раніше *parent*-процесу. На рисунку 21 представлено приклад такої програми, яка для несподіваного завершення *child*-процесу виконує спеціальну функцію `_exit`.



```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main (void)
{
    pid_t pid = fork();
    if (pid == 0) {
        fprintf(stderr, "the zombie was created!\n");
        _exit(0);
    }
    else {
        fprintf(stderr, "the parent start ...\n");
        sleep(10);
        fprintf(stderr, "the parent finish ...\n");
    }
    return EXIT_SUCCESS;
}

```

Рис. 21 – Приклад програми, коли *child*-процес завершується несподівано раніше *parent*-процесу.

Приклад програми з рисунку 21 потрібно запустити у фоновому режимі, використовуючи оператор `&`. Потім в окремому псевдотерміналі необхідно отримати список процесів, наприклад, по команді: `ps -u oracle -o pid,ppid,stat,cmd`

Як видно на рисунку 22, *child*-процес все ще знаходиться в таблиці процесів за станом *S*, але в таблиці процесів з'явився ще один процес – його *child*-процес зі станом *Z* (*Zombie*). Коли через 60 секунд *parent*-процес завершиться, буде очищена таблиця процесів від цього *child*-процесу.

```

oracle@vpsj3IeQ:~/labs_os/Lab8
[oracle@vpsj3IeQ Lab8]$ gcc zombi.c -o zombi
[oracle@vpsj3IeQ Lab8]$ ./zombi &
[1] 31165
[oracle@vpsj3IeQ Lab8]$ the parent start ...
the zombi was created!
the parent finish ...

oracle@vpsj3IeQ:~/labs_os/Lab8
[oracle@vpsj3IeQ Lab8]$ ps -u oracle -o pid,ppid,stat,cmd
  PID  PPID  STAT  CMD
11218 11211  S      sshd: oracle@pts/14
11219 11218  Ss     -bash
11287 11281  S      sshd: oracle@pts/15
11288 11287  Ss+    -bash
31165 11288  S      ./zombi
31166 31165  Z      [zombi] <defunct>
31169 11219  R+     ps -u oracle -o pid,ppid,stat,cmd
[oracle@vpsj3IeQ Lab8]$ |

```

Рис. 22 – Приклад результатів виведення на екран роботи програми

Нажаль *zombie*-процес не можна видалити стандартною командою *kill*, тому що його реально немає в оперативній пам'яті. Є тільки інформацію про нього в таблиці процесів.

В цьому випадку є два варіанти:

- примусово видалити *parent*-процес, після чого автоматично буде видалено *zombie*-процес як звичайний *child*-процес;
- очікувати самостійного завершення *parent*-процесу;

Але найкращий засіб впоратися із *zombie*-процесами – це гарантувати, що вони не будуть створені. Коли *child*-процес завершується, *parent*-процес йому повинен надіслати сигнал *SIGCHLD* (код =17) – сигнал про зміни стану *child*-процесу, наприклад, завершено, призупинено, відновлено.

З моменту завершення *child*-процесу і до того моменту як *parent*-процес отримує від нього сигнал *SIGCHLD*, *child*-процес може знаходитися у *zombie*-стані, тому *parent*-процес, отримавши цей сигнал повинен на нього відреагувати.

На рисунку 23 представлено приклад програми з обробником сигналу *SIGCHLD*. Програма використовує функції:

- *signal* – визначення функції-обробника сигналу про завершення *child*-процесу;
- *wait* - функція очікування завершення *child*-процесу перед своїм продовженням.

На рисунку 23 функція *signal* встановлює покажчик функції на обробник сигналу.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

void sighandler ( int sig) {
    printf("Signal handler for signal = %d\n", sig);
    wait(0);
}

int main (void)
{
    int i;
    signal(SIGCHLD, &sighandler);
    pid_t pid = fork();
    if (pid == 0) {
        fprintf(stdout,"the child suddenly stopped!\n");
        _exit(0);
    }
    else {
        fprintf(stdout,"the parent start ...\n");
        sleep(10);
        fprintf(stdout,"the parent finish ...\n");
    }
}

return EXIT_SUCCESS;
}
```

Рис. 23 – Приклад програми з перевіркою завершення *child*-процесу

Оскільки *zombie*-процес вже завершено необхідно, щоб ОС отримала підтвердження про завершення цього процесу, тому отримавши сигнал *SIGCHLD* програма повинна викликати функцію *wait(0)* для того, щоб почекати остаточного завершення породженого процесу.

На рисунку 24 наведено результати виконання програми, яка вже не залишає *zombie*-процес, успішно оброблюючи сигнал *SIGCHLD* від *child*-процесу, який може стати *zombie*-процесом.

Насправді, *parent*-процесу необхідно зробити більше, ніж просто підтвердити сигнал. Йому слід було б також очистити дані про *child*-процес. Після виконання програми, перевіряється список процесів. Оброблювач сигналу отримує значення 17 (*SIGCHLD*) як підтвердження про завершення *child*-процесу, і *parent*-процес повертається в стан очікування через функцію *sleep(10)* та завершує свою роботу.

```
[blazhko@vps6iefe OS-Labwork9-Examples]$ gcc zombi_stop.c -o zombi_stop
[blazhko@vps6iefe OS-Labwork9-Examples]$ ./zombi_stop
the parent start ...
the child suddenly stopped!
Signal handler for signal = 17
the parent finish ...
[blazhko@vps6iefe OS-Labwork9-Examples]$ ps -u
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
blazhko	2866512	0.0	0.2	25416	5008	pts/7	Ss+	07:04	0:00	-bash
blazhko	2876588	0.0	0.2	25548	5356	pts/8	Ss	07:46	0:00	-bash
blazhko	2877469	0.0	0.2	58736	3964	pts/8	R+	07:49	0:00	ps -u

Рис. 24 – Приклад результатів виведення на екран роботи програми

*Zombie*-процес не виконується і не споживає ресурсів центрального процесору, він тільки займає простір в таблиці процесів. Тому єдиний недолік його присутності – це можливе переповнення таблиці процесів користувача, який запускає процеси, що в пальшому може призвести до збільшення оперативної пам'яті та перезавантаження процедури керування процесами.

## 2. Завдання

Виконати наступні дії з підготовки до виконання завдань роботи:

- 1) встановити з'єднання з віддаленим *Linux*-сервером з IP-адресою = 46.175.148.116, логіном, наданим вам викладачем, та паролем, зміненим вами у попередній роботі;
- 2) перейти до каталогу *Git*-репозиторія;
- 3) створити нову *Git*-гілку з назвою «*Laboratory-work-9*»;
- 4) перейти до роботи зі створеною гілкою;
- 5) створити каталог з назвою «*Laboratory-work-9*»;
- 6) перейти до каталогу «*Laboratory-work-9*»;
- 7) в каталозі «*Laboratory-work-9*» створити файл *README.md* та додати до файлу рядок тексту із темою лабораторної роботи «*Основи програмного керування процесами в Unix-подібних ОС*» як заголовок 2-го рівня *Markdown*-форматування;
- 8) виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add*, *commit* із коментарем «*Changed by Local Git*» та *push*;
- 9) на веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-9*»;
- 10) перейти до каталогу «*Laboratory-work-9*» та розпочати процес редагування файлу *README.md* ;
- 11) в подальшому за результатами рішень кожного наступного розділу завдань, який містить перегляд результатів роботи створених програм, до файлу *README.md* додавати рядки як заголовки 3-го рівня *Markdown*-форматування з назвами розділу, а також знімки екрані та підписи до знімків екранів з описом пунктів завдань.

### 2.1 Перегляд інформації про процес

2.1.1 Створити *C*-програму з назвою за шаблоном «*ваше прізвище\_process\_info.c*», наприклад, *blazhko\_process\_info.c* яка виводить на екран таку інформацію:

- ідентифікатор групи процесів, до якої належить процес;
- ідентифікатор процесу, що викликав цю функцію;
- ідентифікатор *parent*-процесу;
- ідентифікатор користувача процесу, який викликав цю функцію;
- ідентифікатор групи користувача процесу, який викликав цю функцію.

Скомпілювати створену *C*-програму та перевірити її роботу.

2.1.2 Запустити дві копії програми у двох режимах за прикладами з рисунку 4:

- паралельне виконання двох процесів;
- конвеєрне виконання двох процесів.

Переглянути результат роботи програми у двох режимах, порівняйте значення ідентифікаторів груп процесів для різних процесів та зробити відповідні висновки за результатами порівнянь, які вказати у підпису до відповідного рисунку зі знімком екрану.

## 2.2 Створення *child*-процесу

2.2.1 Створити C-програму, яка породжує процес та замінює образ процесу на команду у відповідності із варіантом з таблиці 3. Назва програми повинна співпадати з назвою команди з таблиці 3, але з додатком у вигляді транслітерації вашого прізвища, наприклад, *touch\_blazhko.c*

Перед включенням відповідної команди з таблиці 3 до програмного коду перевірте правильність її роботи зі звичайного командного рядку, щоб зрозуміти, що вона повинна виводити на екран.

У програмі необхідно виконати наступне:

- 1) *child*-процес повинен вивести на екран повідомлення типу «*the child of Students` Surname executes: команда*», наприклад, «*the child of Blazhko executes: echo*»;
- 2) заміна програмного образу *child*-процесу через функцію *execv* з урахуванням аргументів команди, приклади яких наведено у стовпчику таблиці 3 (функція повинна забезпечити виконання відповідної команди з урахуванням двох аргументів, значення яких буде розміщено у двох елементах масиву);
- 3) щоб повідомлення *child*-процесу не з'являлося пізніше виконання функції *execv* через окремі затримки у процесах рекомендується перед функцією *execv* додати функцію затримки, наприклад, *sleep(1)*.

Таблиця 3 – Варіанти команд

№ варіанту	Команда	Приклад команди з аргументами
1.	<i>ls</i>	<i>ls -l -a</i>
2.	<i>sum</i>	<i>sum -r /etc/passwd</i>
3.	<i>who</i>	<i>who -b -t</i>
4.	<i>ps</i>	<i>ps -o pid,cmd</i>
5.	<i>pstree</i>	<i>pstree -u -s</i>
6.	<i>cat</i>	<i>cat -E /etc/hosts</i>
7.	<i>free</i>	<i>free -k -w</i>
8.	<i>pwd</i>	<i>pwd -L -P</i>
9.	<i>cp</i>	<i>cp /etc/hosts ./</i>
10.	<i>pushd</i>	<i>pushd -n ./</i>
11.	<i>touch</i>	<i>touch -a file.txt</i>
12.	<i>cd</i>	<i>cd -P ./</i>
13.	<i>paste</i>	<i>paste /etc/hosts /etc/hosts</i>

Таблиця 3 – продовження

№ варіанту	Команда	Приклад команди з аргументами
14.	<i>who</i>	<i>who -u -p</i>
15.	<i>ps</i>	<i>ps -o pid,ppid</i>
16.	<i>stat</i>	<i>stat -t /etc/passwd</i>
17.	<i>pstree</i>	<i>pstree -p -a</i>
18.	<i>cat</i>	<i>cat /etc/hosts /etc/hosts</i>
19.	<i>head</i>	<i>head --lines=3 /etc/group</i>
20.	<i>tail</i>	<i>tail --lines=3 /etc/group</i>
21.	<i>cd</i>	<i>cd -L ./</i>
22.	<i>command</i>	<i>command -v tail</i>
23.	<i>date</i>	<i>date -d -u</i>
24.	<i>seq</i>	<i>seq 1 5</i>
25.	<i>wc</i>	<i>wc -c /etc/group</i>
26.	<i>uniq</i>	<i>uniq -d /etc/group</i>
27.	<i>uname</i>	<i>uname -s -n</i>
28.	<i>truncate</i>	<i>truncate --size=10 ./file1</i>
29.	<i>ls</i>	<i>ls -l -i</i>

2.2.2 Скомпілювати програму та перевірити її роботу.

Програма повинна виконуватися без помилок.

### 2.3 Обмін сигналами між процесами

2.3.1 Створити C-програму з назвою «*ваше прізвище\_get\_signal*», в якій процес очікує отримання сигналу *SIGUSR2* та виводить повідомлення типу «*Process of Students` Surname got signal*» після отримання сигналу, де замість слова *Students` Surname* в повідомленні повинно бути ваше прізвище в транслітерації.

2.3.2 Скомпілювати програму та запустити її.

2.3.3 Використовуючи інший псевдотермінал, створити C-програму з назвою «*ваше прізвище\_set\_signal*», яка надсилає сигнал *SIGUSR2* процесу, запущеному в попередньому пункті завдання.

2.3.4 Скомпілювати другу створену C-програму та запустити її в іншому псевдотерміналі, проаналізувавши повідомлення, які в першому псевдотерміналі виводить перша програма.

Завершити процес, запущений в попередньому пункті завдання.

### 2.4 Створення процесу-сироти

2.4.1 Створити C-програму з назвою «*ваше прізвище\_orphan*», в якій *parent*-процес несподівано завершується раніше *child*-процесу. *Parent*-процес повинен очікувати завершення  $n+1$  секунд.

*Child*-процес повинен в циклі  $(2*n+1)$  разів із затримкою в 1 секунду виводити повідомлення, наприклад, «*Parent of Students` Surname*», за шаблоном як в попередньому завданні, і додатково виводити *PPID parent*-процесу.

Значення  $n$  – номер вашого варіанту.

2.4.2 Скомпілювати програму та перевірити її роботу.

Переглянути вміст таблиці процесів зі станами процесів та зробити висновки.

## 2.5 Створення *zombie*-процесу

2.5.1 Створити *C*-програму з назвою «*ваше прізвище\_zombie.c*», в якій *child*-процес несподівано завершується раніше *parent*-процесу, перетворюється на *zombie*-процес, виводячи в результаті повідомлення, наприклад, «*I am Zombie-process of Students` Surname*», за шаблоном як в попередньому завданні.

2.5.2 Скомпілювати програму та запустити її у фоновому режимі.

Переглянути вміст таблиці процесів зі станами процесів та зробити висновки.

## 2.6 Попередження створення *zombie*-процесу

2.6.1 Створити *C*-програму з назвою «*ваше прізвище\_zombie\_stop.c*», в якій *child*-процес також як в попередньому завданні може перетворитися на *zombie*-процес, але ця подія вже повинна контролюватися *parent*-процесом.

*Child*-процес повинен виводити повідомлення, наприклад, «*Child of Students` Surname is finished*», за шаблоном як в попередньому завданні.

*Parent*-процес повинен очікувати  $(3*n)$  секунд.

Значення  $n$  – номер вашого варіанту.

2.6.2 Скомпілювати програму та запустити її у фоновому режимі.

Переглянути вміст таблиці процесів зі станами процесів та зробити висновки.

## 2.7 Підготовка процесу *Code Review* для надання рішень завдань лабораторної роботи на перевірку викладачем

Примітка: Рішення на завдання 2.1 та 2.2 можуть бути надані під час *Online*-заняття для отримання відповідних балів. За межами *Online*-заняття виконуються всі рішення.

2.7.1 На веб-сервісі *GitHub* зафіксувати зміни у файлі *README.md*

2.7.2 Скопіювати файли, які було створено у попередніх розділах завдань, в каталог «*Laboratory-work-9*» *Git*-репозиторію.

2.7.3 Оновити *Git*-репозиторій змінами нової гілки «*Laboratory-work-9*» з *GitHub*-репозиторію.

2.7.4 Оновити *GitHub*-репозиторій змінами нової гілки «*Laboratory-work-9*» *Git*-репозиторію.

2.7.5 Виконати запит *Pull Request*.

*Примітка: Увага! Не натискайте кнопку «Merge pull request»!*

*Це повинен зробити лише рецензент, який є вашим викладачем!*

Може бути виконано два запити *Pull Request*: під час *Online*-заняття та за межами *Online*-заняття.

2.7.6 Якщо запит *Pull Request* було зроблено під час *Online*-заняття, тоді рецензент-викладач перегляне рішення, надасть оцінку та закриє *Pull Request* без операції *Merge*.

Коли буде зроблено остаточний запит *Pull Request*, тоді рецензент-викладач перегляне ваше рішення та виконає злиття нової гілки та основної гілки через операцію *Merge*. Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з'являться на сторінці *Pull request*.

## 2.8 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+2 бали	під час <i>Online</i> -заняття виконано правильні рішення завдань 2.1, 2.2 або на наступному <i>Online</i> -занятті з лабораторної роботи чи на консультації отримано правильну відповідь на два запитання, які стосуються виконаних завдань
+2 бали	1) всі рішення роботи відповідають завданням 2) <i>Pull Request</i> представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну помилку	в рішенні є помилка, про яку вказано в <i>Code Review</i>
-1 бал	<i>Pull Request</i> представлено пізніше часу завершення найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	на наступному <i>Online</i> -занятті з лабораторної роботи або на консультації отримано правильну відповідь на два запитання, які стосуються виконаних завдань

## Література

1. Блажко О.А. Відео-запис лекції «Основи програмного керування процесами в Unix-подібних ОС». URL: <https://youtu.be/VSc-yY9azvo>