4. Засоби System V IPC. Організація роботи з розділюваною пам'яттю в Linux. Поняття ниток виконання (thread)

4.1. Переваги та недоліки потокового обміну даними.

Під час вивчення попередньої теми розглядалися механізми, що забезпечують потокову передачу даних між процесами в операційній системі Linux, а саме **`pipe`** та **`FIFO`**. Потокові механізми досить прості в реалізації і зручні для використання, але мають ряд істотних недоліків:

- Операції читання і запису не аналізують вміст даних, що передаються. Процес, що прочитав 20 байт із потоку, не може сказати, чи були вони записані одним процесом або декількома, чи записувалися вони за один раз або, наприклад, виконано 4 операції запису по 5 байт. Дані в потоці ніяк не інтерпретуються системою. Якщо потрібна яка-небудь інтерпретація даних, то процеси, що передають і приймають дані, повинні заздалегідь погодити свої дії і уміти здійснювати її самостійно.
- Для передачі інформації від одного процесу до іншого потрібно, як мінімум, дві операції копіювання даних: перший раз – з адресного простору процесу, який передає дані, в системний буфер, другий раз – із системного буфера в адресний простір процесу, що приймає дані.
- Процеси, що обмінюються інформацією, повинні одночасно існувати в обчислювальній системі. Не можна записати інформацію в потік за допомогою одного процесу, завершити його, а потім, через якийсь час, запустити інший процес і прочитати записану інформацію.

4.2. Поняття про System V IPC

Зазначені вище недоліки потоків даних привели до розробки інших механізмів передачі інформації між процесами. Частина цих механізмів, що вперше з'явилися в UNIX System V і згодом були перенесені практично в усі сучасні версії операційної системи UNIX, одержали загальну назву System V IPC (IPC – скорочення від interprocess communications). У групу System V IPC входять: черги повідомлень, розділювана пам'ять і семафори. Ці засоби організації взаємодії процесів зв'язані не тільки спільним походженням, але й мають схожий інтерфейс для виконання подібних операцій, наприклад, для виділення і звільнення відповідного ресурсу в системі. Будемо розглядати їх у порядку від менш семантично навантажених з погляду операційної системи до більш семантично навантажених. Частину цієї теми присвятимо вивченню розділюваної пам'яті. Семафори будуть розглянуті в темі 5, а черги повідомлень – в темі 6.

4.3. Простір імен. Адресація в System V IPC. Функція ftok()

Усі засоби зв'язку з System V IPC, як і вже розглянуті **pipe** і **FIFO**, є засобами зв'язку з непрямою адресацією. У попередній темі встановлено, що для організації взаємодії нерідних процесів за допомогою засобу зв'язку з непрямою адресацією необхідно, щоб цей засіб зв'язку мав ім'я. Відсутність імен в **pipe** дозволяє процесам одержувати інформацію про розташування **pipe** в системі і його стан тільки через родинні зв'язки. Наявність асоційованого імені в **FIFO** – імені спеціалізованого файлу у файловій системі – дозволяє нерідним процесам одержувати цю інформацію через інтерфейс файлової системи.

Множину всіх можливих імен для об'єктів якого-небудь виду прийнято називати простором імен відповідного виду об'єктів. Для **FIFO** простором імен є множина всіх допустимих імен файлів у файловій системі. Для всіх об'єктів з System V IPC таким простором імен є множина значень деякого цілочисельного типу даних – **key_t** – ключа. Причому програмістові **не дозволено явно присвоювати значення ключа**, це значення задається опосередковано: через комбінацію імені якого-небудь файлу, що вже існує у файловій системі, і невеликого цілого числа – наприклад, номера екземпляра засобу зв'язку.

Такий хитрий спосіб одержання значення ключа пов'язаний із двома міркуваннями:

35/17 Якщо дозволити програмістам самим присвоювати значення ключа для ідентифікації засобів зв'язку, то не виключено, що два програмісти випадково скористаються тим самим значенням, не підозрюючи про це. Тоді їхні процеси будуть несанкціоновано взаємодіяти через той самий засіб комунікації, що може привести до нестандартної поведінки цих процесів. Тому основним компонентом значення ключа є перетворене в числове значення повне ім'я деякого файлу, доступ до якого на читання дозволений процесу. Кожний програміст має можливість використати для цієї мети свій специфічний файл, наприклад виконуваний файл, пов'язаний з одним із взаємодіючих процесів. Зазначимо, що перетворення текстового імені файлу в число ґрунтується на розташуванні зазначеного файлу на жорсткому диску або іншому фізичному носії. Тому для утворення ключа варто застосовувати файли, що не змінюють свого розташування протягом часу взаємодії процесів;

35/17 Другий компонент значення ключа використовується для того, щоб дозволити програмістові зв'язати з іменем файлу більше одного екземпляра кожного засобу зв'язку. Як компонент можна задавати порядковий номер відповідного екземпляра.

Одержання значення ключа із двох компонентів здійснюється функцією **ftok()**.

Функція для генерації ключа System V IPC

Прототип функції

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj);
```

Опис функції

Функція **ftok** призначена для перетворення імені існуючого файлу і невеликого цілого числа, наприклад, порядкового номера екземпляра засобів зв'язку, у ключ System V IPC.

Параметр **pathname** повинен бути покажчиком на ім'я існуючого файлу, доступного для процесу, що викликає функцію.

Параметр **proj** – це невелике ціле число, що характеризує екземпляр засобу зв'язку.

У випадку неможливості генерації ключа функція повертає від'ємне значення, у протилежному випадку вона повертає значення згенерованого ключа. Тип даних **key_t** – 32-бітове ціле.

Ще раз підкреслимо три важливих моменти, пов'язаних з використанням імені файлу для одержання ключа. По-перше, необхідно вказувати ім'я файлу, що **вже існує** у файловій системі і до якого процес **має право доступу на читання** (не плутайте із заданям імені файлу при створенні **FIFO**, де вказувалося ім'я **для створюваного нового** спеціального файлу). По-друге, зазначений файл повинен **зберігати своє розташування на диску** доти, поки всі процеси, що беруть участь у взаємодії, не одержать ключ System V IPC. По-третє, задання імені файлу, як одного з компонентів для одержання ключа, у жодному разі не означає, що інформація, передана за допомогою асоційованого засобу зв'язку, буде розташовуватися в цьому файлі. Інформація буде зберігатися **всередині адресного простору операційної системи**, а задане ім'я файлу лише дозволяє різним процесам згенерувати ідентичні ключі.

4.4. Дескриптори System V IPC

Інформацію про потоки вводу-виводу, з якими має справу поточний процес, зокрема про **pipe** і **FIFO**, операційна система зберігає в таблиці відкритих файлів процесу. Системні виклики, що здійснюють операції над потоком, використовують як параметр індекс елемента таблиці відкритих файлів, що відповідає потоку, – файловий дескриптор. Використання файлових дескрипторів для ідентифікації потоків усередині процесу дозволяє застосовувати до них уже існуючий інтерфейс для роботи з файлами, але в той же час приводить до автоматичного закриття потоків при завершенні процесу. Цим, зокрема, пояснюється один з перерахованих вище недоліків потокової передачі інформації.

При реалізації компонентів System V IPC була прийнята інша

концепція. Ядро операційної системи зберігає інформацію про всі засоби System V IPC, які використовуються у системі, поза контекстом користувацьких процесів. При створенні нового засобу зв'язку або одержанні доступу до вже існуючого процес одержує невід'ємне ціле число – дескриптор (ідентифікатор) **цього засобу зв'язку**, який однозначно ідентифікує його у всій обчислювальній системі. Цей дескриптор повинен передаватися як параметр всім системним викликам, що здійснюють подальші операції над відповідним засобом System V IPC.

Подібна концепція дозволяє усунути один із найістотніших недоліків, властивих потоковим засобам зв'язку – вимогу одночасного існування взаємодіючих процесів, але в той же час вимагає підвищеної обережності для того, щоб процес, що одержує інформацію, не прийняв замість нових старі дані, випадково залишені в механізмі комунікації.

4.5. Розділювана пам'ять в Linux. Системні виклики shmget(), shmat(), shmdt()

Для операційної системи найменш семантично навантаженим засобом System V IPC є розділювана пам'ять (shared memory). Операційна система може дозволити декільком процесам спільно використовувати деяку ділянку адресного простору.

Усі засоби зв'язку System V IPC вимагають попередніх дій ініціалізації (створення) для організації взаємодії процесів.

Для створення області розділюваної пам'яті з певним ключем або доступу за ключем до вже існуючої ділянки застосовується системний виклик **`shmget()`**. Існує два варіанти його використання для створення нової ділянки розділюваної пам'яті.

- **Стандартний спосіб.** Системному виклику задається значення ключа, сформоване функцією **`ftok()`** для деякого імені файлу і номера екземпляра області розділюваної пам'яті. Як прапорці встановлюється комбінація прав доступу до створюваного сегмента і прапорця **`IPC_CREAT`**. Якщо сегмент для даного ключа ще не існує, то система буде намагатися створити його із зазначеними правами доступу. Якщо ж він уже існував, то ми просто одержимо його дескриптор. Можливе додавання до цієї комбінації прапорців прапорця **`IPC_EXCL`**. Цей прапорець гарантує нормальне завершення системного виклику тільки в тому випадку, якщо сегмент дійсно був створений (тобто раніше він не існував), якщо ж сегмент існував, то системний виклик завершиться з помилкою, і значення системної змінної **`errno`**, описаної у файлі **`errno.h`**, буде встановлене в **`EEXIST`**.
- **Нестандартний спосіб.** Як значення ключа вказується спеціальне значення **`IPC_PRIVATE`**. Використання значення **`IPC_PRIVATE`** завжди приводить до спроби створення нового сегмента розділюваної пам'яті із заданими правами доступу та із ключем, що не збігається зі значенням ключа жодного із уже існуючих сегментів і який не може

бути отриманий за допомогою функції **ftok()** ні при одній комбінації її параметрів. Наявність прапорців **IPC_CREAT** і **IPC_EXCL** у цьому випадку ігнорується.

Системний виклик shmget()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

Опис системного виклику

Системний виклик **shmget** призначений для виконання операції доступу до сегмента розділюваної пам'яті і, у випадку його успішного завершення, повертає дескриптор System V IPC для цього сегмента (ціле невід'ємне число, що однозначно характеризує сегмент всередині обчислювальної системи і яке надалі використовується для інших операцій з ним).

Параметр **key** є ключем System V IPC для сегмента, тобто фактично його іменем із простору імен System V IPC. Як значення цього параметра може використовуватися значення ключа, отримане за допомогою функції **ftok()**, або спеціальне значення **IPC_PRIVATE**. Використання значення **IPC_PRIVATE** завжди приводить до спроби створення нового сегмента розділюваної пам'яті із ключем, що не збігається зі значенням ключа жодного із уже існуючих сегментів і який не може бути отриманий за допомогою функції **ftok()** ні при одній комбінації її параметрів.

Параметр **size** визначає розмір створюваного або вже існуючого сегмента в байтах. Якщо сегмент із зазначеним ключем уже існує, але його розмір не збігається із зазначеним у параметрі **size**, вказується на виникнення помилки.

Параметр **shmflg** – прапорці – відіграє роль тільки при створенні нового сегмента розділюваної пам'яті і визначає права різних користувачів при доступі до сегмента, а також необхідність створення нового сегмента і поведінку системного виклику при спробі створення. Він є деякою комбінацією (за допомогою операції побітове або – "|") наступних визначених значень і вісімкових прав доступу:

- **IPC_CREAT** – якщо сегмента для зазначеного ключа не існує, він повинен бути створений;
- **IPC_EXCL** – застосовується разом із прапорцем **IPC_CREAT**. При спільному їхньому використанні та існуванні сегмента із зазначеним ключем, доступ до сегмента не відбувається і констатується помилкова ситуація, при цьому змінна **errno**, описана у файлі **<errno.h>**, прийме значення **EEXIST**;
- **0400** – дозволене читання для користувача, що створив сегмент;

- **0200** – дозволений запис для користувача, що створив сегмент;
- **0040** – дозволене читання для групи користувача, що створив сегмент;
- **0020** – дозволений запис для групи користувача, що створив сегмент;
- **0004** – дозволене читання для всіх інших користувачів;
- **0002** – дозволений запис для всіх інших користувачів.

Значення, що повертається

Системний виклик повертає значення дескриптора System V IPC для сегмента розділюваної пам'яті при нормальному завершенні і значення -1 при виникненні помилки.

Доступ до створеної ділянки розділюваної пам'яті надалі забезпечується її дескриптором, який поверне системний виклик **shmget()**. Доступ до вже існуючої ділянки також може здійснюватися двома способами:

- Якщо відомо її ключ, то, використовуючи виклик **shmget()**, можемо одержати її дескриптор. У цьому випадку не можна вказувати як складову частину прапорців прапорець **IPC_EXCL**, а значення ключа, природно, не може бути **IPC_PRIVATE**. Права доступу ігноруються, а розмір області повинен збігатися з розміром, зазначеним при її створенні.
- Можемо скористатися тим, що дескриптор System V IPC дійсний у рамках всієї операційної системи, і передати його значення від процесу, що створив розділювану пам'ять, поточному процесу. Відзначимо, що при створенні розділюваної пам'яті за допомогою значення **IPC_PRIVATE** – це єдиний можливий спосіб.

Після одержання дескриптора необхідно включити область розділюваної пам'яті в адресний простір поточного процесу. Це здійснюється за допомогою системного виклику **shmat()**. При нормальному завершенні він поверне адресу розділюваної пам'яті в адресному просторі поточного процесу. Подальший доступ до цієї пам'яті здійснюється за допомогою звичайних засобів мови програмування.

Системний виклик shmat()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg);
```

Опис системного виклику

Системний виклик **shmat** призначений для включення ділянки розділюваної пам'яті в адресний простір поточного процесу.

Параметр **shmid** є дескриптором System V IPC для сегмента розділюваної пам'яті, тобто значенням, що повернув системний виклик **shmget()** при створенні сегмента або при його пошуку за ключем.

Як параметр **shmaddr** у рамках нашого курсу ми завжди будемо

передавати значення **NULL**, дозволяючи операційній системі самій розмістити розділювану пам'ять в адресному просторі нашого процесу.

Параметр **shmflg** у нашому курсі може приймати тільки два значення: **0** – для здійснення операцій читання і запису над сегментом і **SHM_RDONLY** – якщо ми хочемо тільки читати з нього. При цьому процес повинен мати відповідні права доступу до сегмента.

Значення, що повертається

Системний виклик повертає адресу сегмента розділюваної пам'яті в адресному просторі процесу при нормальному завершенні і значення -1 при виникненні помилки.

Після закінчення використання розділюваної пам'яті процес може зменшити розмір свого адресного простору, виключивши з нього цю область за допомогою системного виклику **shmdt()**. Відзначимо, що як параметр системний виклик **shmdt()** вимагає адресу початку ділянки розділюваної пам'яті в адресному просторі процесу, тобто значення, що повернув системний виклик **shmat()**, тому дане значення варто зберігати протягом усього часу використання розділюваної пам'яті.

Системний виклик shmdt()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

Опис системного виклику

Системний виклик **shmdt** призначений для виключення області розділюваної пам'яті з адресного простору поточного процесу.

Параметр **shmaddr** є адресою сегмента розділюваної пам'яті, тобто значенням, яке повернув системний виклик **shmat()**.

Значення, що повертається

Системний виклик повертає значення 0 при нормальному завершенні і значення -1 при виникненні помилки.

4.6. Програми з використанням розділюваної пам'яті

Для ілюстрації використання розділюваної пам'яті розглянемо дві взаємодіючі програми.

```
/* Організуємо розділювану пам'ять для масиву із трьох
цілих чисел. Перший елемент масиву є лічильником кількості
запусків програми 1, тобто даної програми, другий елемент
масиву - лічильником кількості запусків програми 2, третій
елемент масиву - лічильником кількості запусків обох
програм */
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    int *array;     /* Покажчик на розділювану пам'ять */
    int shmid;      /* IPC дескриптор для області
                    розділюваної пам'яті */
    int new = 1;    /* Прапорець необхідності
                    ініціалізації елементів масиву */
    char pathname[] = "04-1a.c"; /* Ім'я файлу,
        використовуване для генерації ключа. Файл із
        таким іменем повинен існувати в поточній
        директорії */
    key_t key;      /* IPC ключ */
    /* Генеруємо IPC ключ із імені файлу 04-1a.c в
       поточній директорії і номера екземпляра області
       розділюваної пам'яті 0 */
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    /* Намагаємося створити розділювану пам'ять для
    згенерованого ключа, тобто якщо для цього ключа вона
    уже існує, системний виклик поверне від'ємне
    значення. Розмір пам'яті визначаємо як розмір масиву
    із трьох цілих змінних, права доступу 0666 - читання
    і запис дозволені для всіх */
    if((shmid = shmget(key, 3*sizeof(int),
        0666|IPC_CREAT|IPC_EXCL)) < 0){
    /* У випадку помилки намагаємося визначити: чи
    виникла вона через те, що сегмент розділюваної
    пам'яті вже існує або з іншої причини */
        if(errno != EEXIST){
           /* Якщо з іншої причини - припиняємо роботу*/
            printf("Can\'t create shared memory\n");
            exit(-1);
        } else {
            /* Якщо через те, що розділювана пам'ять уже
            існує, то намагаємося одержати її IPC
            дескриптор і, у випадку успіху, скидаємо
            прапорець необхідності ініціалізації
            елементів масиву */
            if((shmid=shmget(key,3*sizeof(int),0)) < 0){
                printf("Can\'t find shared memory\n");
                exit(-1);
```

```
            }
            new = 0;
        }
    }
    /* Намагаємося відобразити розділювану пам'ять в
    адресний простір поточного процесу. Зверніть увагу
    на те, що для правильного порівняння ми явно
    перетворюємо значення -1 до вказівника на ціле.*/
    if((array=(int*)shmat(shmid,NULL,0))==(int*)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    /* Залежно від значення прапорця new або
       ініціалізуємо масив, або збільшуємо відповідні
       лічильники */
    if(new){
        array[0] = 1; array[1] = 0; array[2] = 1;
    } else {
        array[0] += 1; array[2] += 1;
    }
    /* Друкуємо нові значення лічильників, видаляємо
    розділювану пам'ять із адресного простору
    поточного процесу і завершуємо роботу */
    printf("Program 1 was spawn %d times,
        program 2 - %d times, total - %d times\n",
        array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Лістинг 4.1а. Програма 1 (04-1a.c) для ілюстрації роботи з розділюваною пам'яттю

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    int *array;
    int shmid;
    int new = 1;
    char pathname[] = "04-1a.c";
    key_t key;
```

```
    if((key = ftok(pathname,0)) < 0){
        printf("Can\'t generate key\n");
        exit(-1);
    }
    if((shmid = shmget(key, 3*sizeof(int),
        0666|IPC_CREAT|IPC_EXCL)) < 0){
        if(errno != EEXIST){
            printf("Can\'t create shared memory\n");
            exit(-1);
        } else {
            if((shmid = shmget(key,3*sizeof(int),0))<0){
                printf("Can\'t find shared memory\n");
                exit(-1);
            }
            new = 0;
        }
    }
    if((array=(int*)shmat(shmid,NULL,0))==(int*)(-1)){
        printf("Can't attach shared memory\n");
        exit(-1);
    }
    if(new){
        array[0] = 0; array[1] = 1; array[2] = 1;
    } else {
        array[1] += 1; array[2] += 1;
    }
    printf("Program 1 was spawn %d times,
        program 2 - %d times, total - %d times\n",
    array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        printf("Can't detach shared memory\n");
        exit(-1);
    }
    return 0;
}
```

Лістинг 4.1b. Програма 2 (04-1b.c) для ілюстрації роботи з розділюваною пам'яттю

Ці програми дуже схожі одна на одну і використовують розділювану пам'ять для зберігання кількості запусків кожної із програм і їхньої суми. У розділюваній пам'яті розміщується масив із трьох цілих чисел. Перший елемент масиву використовується як лічильник для програми 1, другий елемент – для програми 2, третій елемент – для обох програм сумарно. Додатковий нюанс у програмах виникає через необхідність ініціалізації елементів масиву при створенні розділюваної пам'яті. Для цього нам потрібно, щоб програми могли розрізняти випадок, коли вони створили її, і випадок,

коли вона вже існувала. Ми досягаємо відмінності, використовуючи спочатку системний виклик **shmget()** із прапорцями **IPC_CREAT** і **IPC_EXCL**. Якщо виклик завершується нормально, то ми створили розділювану пам'ять. Якщо виклик завершується з констатацією помилки і значення змінної **errno** дорівнює **EEXIST**, то розділювана пам'ять уже існує, і ми можемо одержати її IPC дескриптор, використовуючи той же самий виклик з нульовим значенням прапорців.

4.7. Команди ipcs і ipcrm

Створена область розділюваної пам'яті зберігається в операційній системі навіть тоді, коли немає жодного процесу, що включає її у свій адресний простір. З одного боку це має певні переваги, оскільки не вимагає одночасного існування взаємодіючих процесів. З іншого боку – може створювати істотні незручності. Припустимо, що попередні програми потрібно використати таким чином, щоб підраховувати кількість запусків протягом одного поточного сеансу роботи в системі. Однак у створеному сегменті розділюваної пам'яті залишається інформація від попереднього сеансу, і програми будуть видавати загальну кількість запусків за увесь час роботи з моменту завантаження операційної системи. Можна було б створювати для нового сеансу новий сегмент розділюваної пам'яті, але кількість ресурсів у системі не безмежна. Існують способи вилучати невикористовувані ресурси System V IPC як за допомогою команд операційної системи, так і за допомогою системних викликів. Всі засоби System V IPC вимагають певних дій для звільнення займаних ресурсів після закінчення взаємодії процесів. Для того, щоб вилучати ресурси System V IPC з командного рядка, використаємо дві команди: **ipcs** і **ipcrm**.

Команда **ipcs** видає інформацію про всі засоби System V IPC, що існують у системі, для яких користувач має права на читання: області розділюваної пам'яті, семафори і черги повідомлень.

Команда ipcs

Синтаксис команди

```
ipcs [-asmq] [-tclup]
ipcs [-smq] -i id
ipcs -h
```

Опис команди

Команда **ipcs** призначена для одержання інформації про засоби System V IPC, до яких користувач має право доступу на читання.

Опція **-i** дозволяє вказати ідентифікатор ресурсів. Буде видаватися тільки інформація для ресурсів, що мають цей ідентифікатор.

Види IPC ресурсів можуть бути задані за допомогою наступних опцій:

- **-s** для семафорів;
- **-m** для сегментів розділюваної пам'яті;
- **-q** для черг повідомлень;
- **-a** для всіх ресурсів (за замовчуванням).

Опції **[-tclup]** використовуються для зміни складу вихідної інформації. За замовчуванням для кожного засобу виводяться його ключ, ідентифікатор IPC, ідентифікатор власника, права доступу та ряд інших характеристик. Застосування опцій дозволяє вивести:

- **-t** час здійснення останніх операцій над засобами IPC;
- **-p** ідентифікатори процесу, що створив ресурс, і процесу, що виконав над ним останню операцію;
- **-c** ідентифікатори користувача і групи для творця ресурсу і його власника;
- **-l** системні обмеження для засобів System V IPC;
- **-u** загальний стан IPC ресурсів у системі.

Опція **-h** використовується для одержання короткої довідкової інформації.

Із усієї наведеної інформації нас будуть цікавити тільки IPC ідентифікатори для створених нами засобів. Ці ідентифікатори будуть використовуватися в команді **ipcrm**, що дозволяє вилучити необхідний ресурс із системи. Для вилучення сегмента розділюваної пам'яті ця команда має вигляд:

ipcrm shm <IPC ідентифікатор>

Команда ipcrm

Синтаксис команди

ipcrm [shm | msg | sem] id

Опис команди

Команда **ipcrm** призначена для вилучення ресурсу System V IPC з операційної системи. Параметр **id** задає IPC ідентифікатор для ресурсу, що вилучається, параметр **shm** використовується для сегментів розділюваної пам'яті, параметр **msg** – для черг повідомлень, параметр **sem** – для семафорів.

Якщо поведінка програм, що використовують засоби System V IPC, базується на припущенні, що ці засоби були створені при їх роботі, не забувайте перед їх запуском вилучати вже існуючі ресурси.

4.8. Використання системного виклику shmctl()

Для тієї ж мети – вилучити область розділюваної пам'яті із системи – можна скористатися і системним викликом **shmctl()**. Цей системний виклик дозволяє повністю ліквідувати область розділюваної пам'яті в операційній системі за заданим дескриптором засобу IPC, якщо у Вас вистачає для цього повноважень. Системний виклик **shmctl()** дозволяє виконувати і інші дії над сегментом розділюваної пам'яті, але це в даному курсі не розглядається.

Системний виклик shmctl()

Прототип системного виклику

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Опис системного виклику

Системний виклик **shmctl** призначений для одержання інформації про області розділюваної пам'яті, зміни її атрибутів і вилучення із системи.

Будемо користуватися системним викликом **shmctl** тільки для вилучення області розділюваної пам'яті із системи. Параметр **shmid** є дескриптором System V IPC для сегмента розділюваної пам'яті, тобто значенням, яке повернув системний виклик **shmget()** при створенні сегмента або при його пошуку за ключем.

Як параметр **cmd** будемо передавати значення **IPC_RMID** – команду для вилучення сегмента розділюваної пам'яті із заданим ідентифікатором. Параметр **buf** для цієї команди не використовується, тому будемо підставляти туди значення NULL.

Значення, що повертається

Системний виклик повертає значення 0 при нормальному завершенні і значення -1 при виникненні помилки.

4.9. Розділювана пам'ять і системні виклики fork(), exec() та функція exit()

Важливим питанням є поведінка сегментів розділюваної пам'яті при виконанні процесом системних викликів **fork()**, **exec()** і функції **exit()**.

При виконанні системного виклику **fork()** всі області розділюваної пам'яті, розміщені в адресному просторі процесу, успадковуються породженим процесом.

При виконанні системних викликів **exec()** і функції **exit()** всі області розділюваної пам'яті, розміщені в адресному просторі процесу, виключаються з його адресного простору, але продовжують існувати в операційній системі.

4.10. Поняття про нитки виконання (thread) в Linux. Ідентифікатор нитки виконання. Функція pthread_self()

У багатьох сучасних операційних системах існує розширена реалізація поняття процесу, коли процес є сукупністю виділених йому ресурсів і набору ниток виконання. Нитки процесу розділяють його програмний код, глобальні змінні і системні ресурси, але кожна нитка має власний програмний лічильник, свій вміст регістрів і свій стек. Оскільки глобальні змінні в ниток виконання є загальними, то вони можуть використовувати їх як елементи розділюваної пам'яті, не застосовуючи механізм, описаний вище.

У різних версіях операційної системи UNIX існують різні інтерфейси, що забезпечують роботу з нитками виконання. Коротко ознайомимося з деякими функціями, що дозволяють розділити процес на нитки і керувати їхньою поведінкою у відповідності зі стандартом POSIX. Нитки виконання, що задовольняють стандарту POSIX, прийнято називати **POSIX threads** або, коротко, **pthreads**.

Операційна система Linux не повністю підтримує нитки виконання на рівні ядра системи. При створенні нової нитки запускається новий традиційний процес, що розділяє з батьківським традиційним процесом його ресурси, програмний код і дані, розташовані поза стеком, тобто фактично створюється нова нитка, але ядро не вміє визначати, що ці нитки є складовими частинами одного цілого. Про це "знає" тільки спеціальний процес-координатор, що працює на користувацькому рівні і стартує при першому виклику функцій, що забезпечують POSIX інтерфейс для ниток виконання. Тому надалі ми зможемо спостерігати не всі переваги використання ниток виконання (зокрема, прискорити виконання завдання на однопроцесорному комп'ютері з їхньою допомогою не вийде), але навіть у цьому випадку нитки можна задіяти як дуже зручний спосіб для створення процесів із спільними ресурсами, програмним кодом і розділюваною пам'яттю.

Кожна нитка виконання, як і процес, має в системі унікальний номер – ідентифікатор нитки. Оскільки традиційний процес у концепції ниток виконання трактується як процес, що містить єдину нитку виконання, то ми можемо довідатися ідентифікатор цієї нитки і для будь-якого звичайного процесу. Для цього використовується функція **pthread_self()**. Нитку виконання, яка створюється при народженні нового процесу, прийнято називати **початковою** або **головною** ниткою виконання цього процесу.

Функція pthread_self()

Прототип функції

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Опис функції

Функція **pthread_self** повертає ідентифікатор поточної нитки виконання.

Тип даних **pthread_t** є синонімом для одного із цілочисельних типів мови С.

4.11. Створення і завершення ниток. Функції pthread_create(), pthread_exit(), pthread_join()

Нитки виконання, як і традиційні процеси, можуть породжувати нитки-нащадки і повинні бути функцію із прототипом

```
void *thread(void *arg);
```

Параметр **arg** передається цій функції при створенні нитки і може розглядатися як аналог параметрів функції **main()**, які розглядались під час вивчення теми 2. Значення, що повертається функцією, може інтерпретуватися як аналог інформації, яку батьківський процес може одержати після завершення процесу-нащадка. Для створення нової нитки виконання застосовується функція **pthread_create()**, але тільки всередині свого процесу.

Функція для створення нитки виконання

Прототип функції

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t
    *attr, void * (*start_routine)(void *), void *arg);
```

Опис функції

Функція **pthread_create** призначена для створення нової нитки виконання всередині поточного процесу.

Нова нитка буде виконувати функцію **start_routine** із прототипом

```
void *start_routine(void *)
```

передаючи їй як аргумент параметр **arg**. Якщо потрібно передати більше одного параметра, вони збираються в структуру, і передається адреса цієї структури. Значення, що повертається функцією **start_routine**, не повинно вказувати на динамічний об'єкт даної нитки.

Параметр **attr** призначений для задання різних атрибутів створюваної нитки. Будемо вважати їх заданими за замовчуванням, підставляючи як аргумент значення NULL.

Значення, що повертається

При вдалому завершенні функція повертає значення 0 і поміщає ідентифікатор нової нитки виконання за адресою, на яку вказує параметр **thread**. У випадку помилки повертається **додатне значення** (а не від'ємне, як у більшості системних викликів і функцій!), яке визначає код помилки, описаний у файлі **<errno.h>**. Значення системної змінної **errno** при

цьому не встановлюється.

Результатом виконання цієї функції є поява в системі нової нитки виконання, що буде виконувати функцію, асоційовану з ниткою, передавши їй специфікований параметр, паралельно із уже існуючими нитками виконання процесу.

Створений thread може завершити свою діяльність трьома способами:

- За допомогою виконання функції **`pthread_exit()`**. Функція ніколи не повертається в нитку виконання, що викликала її. Об'єкт, на який вказує параметр цієї функції, може бути використаний в іншій нитці виконання, наприклад, у нитці, яка породила завершену нитку. Цей параметр повинен вказувати на об'єкт, що не є локальним для завершеної нитки, наприклад, на статичну змінну;
- За допомогою повернення з функції, асоційованої з ниткою виконання. Об'єкт, на який вказує адреса, що повертається функцією, як і в попередньому випадку, може бути в іншій нитці виконання. Наприклад, у нитці, яка породила завершену нитку. Він повинен вказувати на об'єкт, що не є локальним для завершеної нитки;
- Якщо в процесі виконується повернення з функції **`main()`** або де-небудь у процесі (у будь-якій нитці виконання) здійснюється виклик функції **`exit()`**, це приводить до завершення всіх ниток процесу.

Функція для завершення нитки виконання

Прототип функції

```
#include <pthread.h>
void pthread_exit(void *status);
```

Опис функції

Функція **`pthread_exit`** призначена для завершення нитки виконання (thread) поточного процесу.

Функція ніколи не повертається у thread, що викликав її. Об'єкт, на який указує параметр **`status`**, може бути згодом використаний в іншій нитці виконання, наприклад у нитці, що породила завершену нитку. Тому він не повинен вказувати на динамічний об'єкт завершеної нитки.

Одним з варіантів одержання адреси, яка повертається завершеною ниткою, з одночасним очікуванням її завершення є використання функції **`pthread_join()`**. Нитка виконання, що викликала цю функцію, переходить у стан **очікування** до завершення заданої нитки. Функція дозволяє також одержати покажчик, який повернула завершена нитка в операційну систему.

Функція pthread_join()

Прототип функції

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **status_addr);
```

Опис функції

Функція **pthread_join** блокує роботу нитки виконання, яка викликала її, до завершення нитки з ідентифікатором **thread**. Після розблокування в покажчик, розташований за адресою **status_addr**, заноситься адреса, яку повернула завершена нитка при виході з асоційованої з нею функції або при виконанні функції **pthread_exit()**. Якщо нас не цікавить, що повернула нитка виконання, то в ролі значення цього параметра можна використати значення **NULL**.

Значення, що повертається

Функція повертає значення 0 при успішному завершенні. У випадку помилки повертається **додатне значення**, яке визначає код помилки, описаний у файлі **<errno.h>**. Значення системної змінної **errno** при цьому не встановлюється.

4.12. Програма з використанням двох ниток виконання

Розглянемо програму, у якій працюють дві нитки виконання.

```
/* Кожна нитка виконання просто збільшує на 1 розділювану
змінну a.*/
#include <pthread.h>
#include <stdio.h>
int a = 0;
/* Змінна a є глобальною статичною для всієї програми,
тому вона буде розділятися обома нитками виконання.*/
/* Нижче розміщений текст функції, який буде асоційовано
з 2-ю ниткою */
void *mythread(void *dummy)
/* Параметр dummy у нашій функції не використовується
і є присутнім тільки для сумісності типів даних. З
тієї ж причини функція повертає значення void *, хоча
це ніяк не використовується в програмі.*/
{
    pthread_t mythid; /*Для ідентифікатора нитки
                        виконання*/
    /* Зазначимо, що змінна mythid є динамічною
локальною змінною функції mythread(), тобто міститься
в стеку, і тому не розділяється нитками виконання. */
    /* Дізнаємось ідентифікатор нитки */
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n",
        mythid, a);
```

```
    return NULL;
}
/* Функція main() - вона ж асоційована функція головної
нитки */
int main()
{
    pthread_t thid, mythid;
    int result;
    /* Намагаємося створити нову нитку виконання,
асоційовану з функцією mythread(). Передаємо їй
як параметр значення NULL. При успішному виконанні в
змінну thid занесеться ідентифікатор нової нитки.
Якщо виникне помилка, то припинимо роботу. */
    result = pthread_create( &thid,
        (pthread_attr_t *)NULL, mythread, NULL);
    if(result != 0){
    printf ("Error on thread create,
        return value = %d\n", result);
    exit(-1);
    }
    printf("Thread created, thid = %d\n", thid);
    /* Дізнаємось ідентифікатор головної нитки */
    mythid = pthread_self();
    a = a+1;
    printf("Thread %d, Calculation result = %d\n",
        mythid, a);
    /* Очікуємо завершення породженої нитки, не
    цікавлячись, яке значення вона нам поверне. Якщо не
    виконати виклик цієї функції, то можлива ситуація,
    коли ми завершимо функцію main() до того, як
    виконається породжена нитка, що автоматично
    завершить її, непередбачено змінивши результати. */
    pthread_join(thid, (void **)NULL);
    return 0;
}
```

Лістинг 4.2. Програма 04-2.c для ілюстрації роботи двох ниток виконання.

Для збірки виконуваного файлу при роботі редактора зв'язків необхідно явно підключити бібліотеку функцій для роботи з нитками, яка не підключається автоматично. Це робиться за допомогою додавання до команди компіляції і редагування зв'язків параметра **-lpthread** – підключити бібліотеку **pthread**.

Зверніть увагу на відмінність результатів цієї програми від схожої програми, що ілюструвала створення нового процесу (тема 2). Програма, яка створювала новий процес, друкувала двічі однакові значення для змінної a,

тому що адресні простори різних процесів незалежні, і кожний процес додавав 1 до своєї власної змінної `a`. Розглянута програма друкує два різних значення, оскільки змінна `a` є розділюваною і кожний thread додає 1 до однієї і тієї ж змінної.

4.13. Необхідність синхронізації процесів і ниток виконання, які використовують загальну пам'ять

Усі розглянуті в цій темі приклади є не зовсім коректними. У більшості випадків вони працюють правильно. Однак, можливі ситуації, коли спільна діяльність цих процесів або ниток виконання приводить до невірних і несподіваних результатів. Це пов'язано з тим, що будь-які неатомарні операції, пов'язані зі зміною вмісту розділюваної пам'яті, є критичною секцією процесу або нитки виконання.

При одночасному існуванні двох процесів в операційній системі може виникнути наступна послідовність виконання операцій:

```
...
Процес 1: array[0] += 1;
Процес 2: array[1] += 1;
Процес 1: array[2] += 1;
Процес 1: printf(
   "Program 1 was spawn %d times,
   program 2 - %d times, total - %d times\n",
   array[0], array[1], array[2]);
...
```

Тоді друк буде видавати неправильні результати. Природно, що відтворити подібну послідовність дій практично нереально. Ми не зможемо підібрати необхідний час старту процесів і ступінь завантаженості обчислювальної системи. Але ми можемо змоделювати цю ситуацію, додавши в обидві програми досить тривалі порожні цикли перед оператором **`array[2] += 1;`**.

Отже, для написання коректно працюючих програм необхідно забезпечувати взаємовиключення при роботі з розділюваною пам'яттю і взаємну черговість доступу до неї. Це можна зробити за допомогою алгоритмів синхронізації, наприклад, алгоритму Петерсона або алгоритму булочної.

У наступній темі ми розглянемо семафори, які є засобом System V IPC, призначеним для синхронізації процесів.

Завдання для самостійної роботи

Самостійно опрацюйте матеріали теми 4 до початку практичного заняття, а також такі питання

1. Логічна організація механізму передачі інформації (встановлення зв'язку, інформаційна валентність процесів і засобів зв'язку,особливості передачі інформації за допомогою ліній зв'язку

(буферизація, потік вводу/виводу і повідомлень), надійність ліній зв'язку, завершення зв'язку).
2. Нитки виконання.
3. Interleaving, race condition і взаємовиключення.
4. Критична секція.
5. Програмні алгоритми організації взаємодії процесів (вимоги до алгоритмів, заборона переривань, строге чергування, прапорці готовності, алгоритм Петерсона, алгоритм булочної (bakery algorithm)).
6. Нитки виконання.

Завдання для лабораторної роботи

1. Які основні недоліки потокової моделі передачі даних?
2. Які є засоби організації взаємодії процесів в System V IPC?
3. Як здійснюється адресація в System V IPC? Як одержати значення ключа із двох компонентів?
4. Для чого використовуються дескриптори System V IPC?
5. Як відбувається створення області розділюваної пам'яті з певним ключем або доступ за ключем до вже існуючої області? Чи потрібно після одержання дескриптора включити область розділюваної пам'яті в адресний простір поточного процесу?
6. Як після закінчення використання розділюваної пам'яті процес може зменшити розмір свого адресного простору?
7. Наберіть програми **`04-1a.c`** і **`04-1b.c`**, відкомпілюйте їх і запустіть декілька разів. Проаналізуйте отримані результати.
8. Чи зберігається створена область розділюваної пам'яті в операційній системі навіть тоді, коли немає жодного процесу, що включає її у свій адресний простір? Чи існують способи вилучати невикористовувані ресурси System V IPC? Які?
9. Вилучіть створений вами сегмент розділюваної пам'яті з операційної системи, використовуючи команди **`ipcs`** і **`ipcrm`**.
10. Напишіть програму **`04-1c.c`** із використанням системного виклику **`shmctl()`** для звільнення спільних ресурсів, використаних у програмах **`04-1a.c`** і **`04-1b.c`**.
11. Поясніть поведінку сегментів розділюваної пам'яті при виконанні процесом системних викликів **`fork()`**, **`exec()`** і функції **`exit()`**.
12. Для закріплення отриманих знань напишіть дві програми, що здійснюють взаємодію через розділювану пам'ять. Перша програма повинна створювати сегмент розділюваної пам'яті і копіювати туди власний вихідний текст. Друга програма повинна отримувати звідти цей текст, друкувати його на екран і видаляти сегмент розділюваної пам'яті із системи.
13. Що таке **`нитка виконання`** (**`thread`**)? Чи підтримує ОС Linux нитки на рівні ядра повністю?

14. Як дізнатись ідентифікатор нитки виконання?
15. Як відбувається створення і завершення нитки виконання? Для чого призначені функції **`pthread_create()`**, **`pthread_exit()`**, **`pthread_join()`**?
16. Наберіть текст програми **`04-2.c`** для ілюстрації роботи двох ниток виконання, відкомпілюйте цю програму і запустіть на виконання. У чому відмінність результатів цієї програми від схожої програми, що ілюструвала створення нового процесу (програма з **`fork()`** з однаковою роботою батька і нащадка)?
17. Зверніть увагу на те, що для створення виконуваного файлу при роботі редактора зв'язків необхідно явно підключити бібліотеку функцій для роботи з нитками, яка не підключається автоматично. Це робиться за допомогою додавання до команди компіляції і редагування зв'язків параметра **`-lpthread`** – підключити бібліотеку **`pthread`**.
18. Модифікуйте попередню програму, додавши до неї третю нитку виконання.
19. Навіщо необхідна синхронізація процесів і ниток виконання, які використовують загальну пам'ять?
20. Модифікуйте програми **`04-1a.c`** і **`04-1b.c`** додавши в обидві програми досить тривалі за часом виконання порожні цикли перед оператором **`array[2] += 1;`**. Назвіть ці програми **`04-3a.c`** і **`04-3b.c`**, відкомпілюйте їх і запустіть кожну з них один раз для створення і ініціалізації розділюваної пам'яті. Потім запустіть іншу і, поки вона перебуває в циклі, запустіть, наприклад, з іншого віртуального термінала, знову першу програму. Ви одержите несподіваний результат: кількість запусків окремо не буде відповідати кількості запусків разом.
21. Модифікуйте програми **`04-3a.c`** і **`04-3b.c`** для коректної роботи за допомогою алгоритму Петерсона.