Керування процесами і потоками

- Означення процесу та потоку
- Реалізація та використання моделі процесів і багатопотоковості
- Подання процесів і потоків в операційній системі
- Створення та завершення процесів і потоків
- Керування процесами та потоками в UNIX
- Керування процесами та потоками у Windows XP

Процеси і потоки в сучасних ОС

Під *процесом* розуміють абстракцію ОС, яка об'єднує все необхідне для виконання однієї програми в певний момент часу.

Програма - це деяка послідовність машинних команд, що зберігається на диску, в разі необхідності завантажується у пам'ять і виконується. Можна сказати, що під час виконання програму представляє процес.

Для успішного виконання програми потрібні певні ресурси. До них належать:

- ресурси, необхідні для послідовного виконання програмного коду (передусім процесорний час);
- ресурси, що дають можливість зберігати інформацію, яка забезпечує виконання програмного коду (регістри процесора, оперативна пам'ять тощо).

❖ Ці групи ресурсів визначають дві складові частини процесу:

❖ послідовність виконуваних команд процесора;

- набір адрес пам'яті *(адресний простір),* у якому розташовані ці команди і дані для них.

Потоком (потік керування, нитка, thread) називають набір послідовно виконуваних команд процесора, які використовують загальний адресний простір процесу. Оскільки в системі може одночасно бути багато потоків, завданням ОС є організація перемикання процесора між ними і планування їхнього виконання. У багатопроцесорних системах код окремих потоків може виконуватися на окремих процесорах.

Т.ч., *процесом* називають сукупність одного або декількох потоків і захищеного адресного простору, у якому ці потоки виконуються.

Захищеність адресного простору процесу є його найважливішою характеристикою.

Моделі процесів і потоків

Максимально можлива кількість процесів (захищених адресних просторів) і потоків, які в них виконуються, може варіюватися в різних системах.

- В однозадачних системах є тільки один адресний простір, у якому в кожен момент часу може виконуватися один потік.
- У деяких вбудованих системах теж є один адресний простір (один процес), але в ньому дозволене виконання багатьох потоків. У цьому разі можна організовувати паралельні обчислення, але захист даних застосувань не реалізовано.
- У системах, подібних до традиційних версій UNIX, допускається наявність багатьох процесів, але в рамках адресного простору процесу виконується тільки один потік. Це традиційна однопотокова *модель процесів.* Поняття потоку в даній моделі не застосовують, а використовують терміни «перемикання між процесами», «планування виконання процесів», «послідовність команд процесу» тощо (тут під процесом розуміють його єдиний потік).
- У більшості сучасних ОС (таких, як лінія Windows XP, сучасні версії UNIX) може бути багато процесів, а в адресному просторі кожного процесу - багато потоків. Ці системи підтримують багатопотоковість або реалізують *модель потоків.* Процес у такій системі називають *багатопотоковим процесом.*

Складові елементи процесів і потоків

До елементів процесу належать:

- ✓ захищений адресний простір;
- ✓ дані, спільні для всього процесу (ці дані можуть спільно використовувати всі його потоки);
- ✓ інформація про використання ресурсів (відкриті файли, мережні з'єднання тощо);
- ✓ інформація про потоки процесу. Потік містить такі елементи:
  - ✓ стан процесора (набір поточних даних із його регістрів), зокрема лічильник поточної інструкції процесора;
- ✓ стек потоку (ділянка пам'яті, де перебувають локальні змінні потоку й адреси повернення функцій, що викликані у його коді).

Багатопотоковість та її реалізація

Поняття паралелізму

Використання декількох потоків у застосуванні означає внесення в нього паралелізму (concurrency). *Паралелізм –* це одночасне (з погляду прикладного програміста) виконання дій різними фрагментами коду застосування.

Види паралелізму

Можна виділити такі основні види паралелізму:

- ❖паралелізм багатопроцесорних систем;
- ❖паралелізм операцій введення-виведення;
- ❖паралелізм взаємодії з користувачем;
- ❖паралелізм розподілених систем.

Способи реалізації моделі потоків

Потік користувача – це послідовність виконання команд в адресному просторі процесу. Ядро ОС не має інформації про такі потоки, вся робота з ними виконується в режимі користувача. Засоби підтримки потоків користувача надають спеціальні системні бібліотеки; вони доступні для прикладних програмістів у вигляді бібліотечних функцій. (*потоки POSIX*).

Потік ядра – це послідовність виконання команд в адресному просторі ядра. Потоками ядра управляє ОС, перемикання ними можливе тільки у привілейованому режимі. Є потоки ядра, які відповідають потокам користувача, і потоки, що не мають такої відповідності.

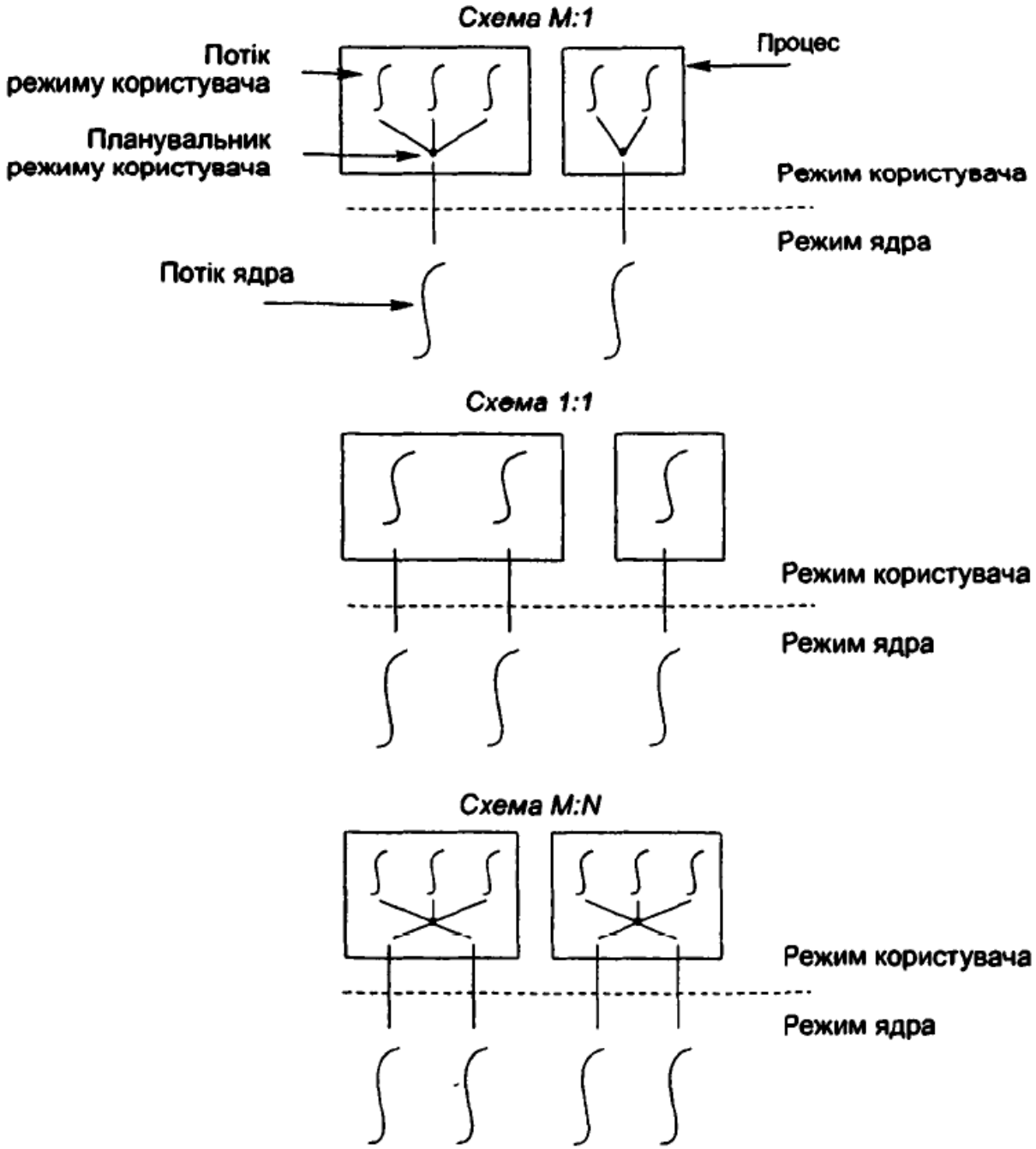
Схема M:1
Потік
режиму користувача
Планувальник
режиму користувача
Процес
Режим користувача
Режим ядра
Потік ядра
Схема 1:1
Режим користувача
Режим ядра
Схема M:N
Режим користувача
Режим ядра

Стани процесів і потоків

Для потоку дозволені такі стани:

- *створення* (new) — потік перебуває у процесі створення;
- *виконання* (running) — інструкції потоку виконує процесор (у конкретний момент часу на одному процесорі тільки один потік може бути в такому стані);
- *очікування* (waiting) — потік очікує деякої події (наприклад, завершення операції введення-виведення); такий стан називають також заблокованим, а потік - припиненим;
- *готовність* (ready) — потік очікує, що планувальник перемкне процесор на нього, при цьому він має всі необхідні йому ресурси, крім процесорного часу;
- *завершення* (terminated) - потік завершив виконання (якщо при цьому його ресурси не були вилучені з системи, він переходить у додатковий стан - *стан зомбі).*

Опис процесів і потоків

Для керування розподілом ресурсів ОС повинна підтримувати структури даних, які містять інформацію, що описує процеси, потоки і ресурси. До таких структур даних належать:

- таблиці розподілу ресурсів: таблиці пам'яті, таблиці введення-виведення, таблиці файлів тощо;
- таблиці процесів і таблиці потоків, де міститься інформація про процеси і потоки, присутні у системі в конкретний момент.

Керуючі блоки процесів і потоків

Інформацію про процеси і потоки в системі зберігають у спеціальних структурах даних, які називають керуючими блоками процесів і керуючими блоками потоків.

Керуючий блок потоку (Thread Control Block, TCB) відповідає активному потоку, тобто тому, який перебуває у стані готовності, очікування або виконання. Цей блок може містити таку інформацію:

- ідентифікаційні дані потоку (зазвичай його унікальний ідентифікатор);
- стан процесора потоку: користувацькі регістри процесора, лічильник інструкцій, покажчик на стек;
- інформацію для планування потоків.

Таблиця потоків - це зв'язний список або масив керуючих блоків потоку. Вона розташована в захищеній області пам'яті ОС.

Керуючий блок процесу (Process Control Block, PCB) відповідає процесу, що присутній у системі. Такий блок може містити:

- ідентифікаційні дані процесу (унікальний ідентифікатор, інформацію про інші процеси, пов'язані з даним);
- інформацію про потоки, які виконуються в адресному просторі процесу (наприклад, покажчики на їхні керуючі блоки);
- інформацію, на основі якої можна визначити права процесу на використання різних ресурсів (наприклад, ідентифікатор користувача, який створив процес);
- інформацію з розподілу адресного простору процесу;
- інформацію про ресурси введення-виведення та файли, які використовує процес.

Образи процесу і потоку

Сукупність інформації, що відображає процес у пам'яті, називають *образом процесу* (process image), а всю інформацію про потік - *образом потоку* (thread image). До образу процесу належать:

- керуючий блок процесу;
- програмний код користувача;
- дані користувача (глобальні дані програми, загальні для всіх потоків);
- інформація образів потоків процесу.

Програмний код користувача, дані користувача та інформація про потоки завантажуються в адресний простір процесу.

До образу потоку належать:

- керуючий блок потоку;
- стек ядра (стек потоку, який використовується під час виконання коду потоку в режимі ядра);
- стек користувача (стек потоку, доступний у користувацькому режимі).

Створення процесів

Базові принципи створення процесів

Процеси можуть створюватися ядром системи під час її ініціалізації.

Найчастіше процеси створюються під час виконання інших процесів. У цьому разі процес, який створює інший процес, називають *предком,* а створений ним процес — *нащадком*.

Нові процеси можуть бути створені під час роботи застосування відповідно до його логіки.

Розрізняють два типи процесів з погляду їхньої взаємодії із користувачем – *інтерактивні процеси* та *фонові процеси*.

Керування адресним простором під час створення процесів

- Системні виклики fork() і exec()
- Запуск застосування одним системним викликом

Технологія копіювання під час запису (copy-on-write)

Особливості завершення процесів

Три варіанти завершення процесів.

Процес *коректно завершується* самостійно після виконання своєї Для цього код процесу має виконати системний виклик завершення процесу. Такий виклик у POSIX-системах називають exit(). Він може повернути процесу, що його викликав, або ОС код повернення, який дає змогу оцінити результат виконання процесу.

Процес *аварійно завершується* через помилку.

Процес *завершується іншим процесом* або ядром системи. Процес може припинити виконання іншого процесу за допомогою системного виклику, який у POSIX-системах називають kill().

Синхронне й асинхронне виконання процесів

Коли у системі з'являється новий процес, для старого процесу є два основних варіанти дій:

- продовжити виконання паралельно з новим процесом — такий режим роботи називають *асинхронним виконанням;*
- призупинити виконання доти, поки новий процес не буде завершений, - такий режим роботи називають *синхронним виконанням.* (У цьому разі використовують спеціальний системний виклик, який у POSIX-системах називають wait ().)

Створення і завершення потоків

Під час створення потоку система повинна виконати такі дії.

1. Створити структури даних, які відображають потік в ОС.
2. Виділити пам'ять під стек потоку.
3. Встановити лічильник команд процесора на початок коду, який буде виконуватися в потоці; цей код називають *процедурою* або *функцією потоку,* покажчик на таку процедуру передають у системний виклик створення потоку.

Локальні змінні функції потоку розташовані у стеку потоку і доступні лише його коду, глобальні змінні доступні всім потокам.

Як і процеси, потоки можуть виконуватися синхронно й асинхронно. Потік, створивши інший потік, може призупинити своє виконання до його завершення. Таке очікування називають *приєднанням потоків* (thread joining, очікує той, хто приєднує).

Керування процесами в UNIX і Linux

В UNIX-системах образ процесу містить такі компоненти:

- керуючий блок процесу;
- код програми, яку виконує процес;
- стек процесу, де зберігаються тимчасові дані (параметри процедур, повернені значення, локальні змінні тощо);
- глобальні дані, спільні для всього процесу.

Із кожним процесом у системі пов'язана ідентифікаційна інформація.

Ідентифікатор процесу (pid) є унікальним у межах усієї системи, і його використовують для доступу до цього процесу. Ідентифікатор процесу і nit завжди дорівнює одиниці.

Ідентифікатор процесу-предка (ppid) задають під час його створення. Будь-який процес має мати доступ до цього ідентифікатора. Так в UNIX-системах обов'язково підтримується зв'язок «предок-нащадок». Якщо предок процесу *P* завершує виконання, предком цього процесу автоматично стає init, тому ppid для *P* дорівнюватиме одиниці.

Із процесом також пов'язаний набір атрибутів безпеки.

- Реальні ідентифікатори користувача і групи процесу (uid, gid) відповідають користувачеві, що запустив програму, внаслідок чого в системі з'явився відповідний процес.
- Ефективні ідентифікатори користувача і групи процесу (euid, egid) використовують у спеціальному режимі виконання процесу — виконанні з правами власника.

Керуючий блок процесу в Linux відображається структурою даних **task_struct**. До найважливіших полів цієї структури належать поля, що містять таку інформацію:

- ідентифікаційні дані (зокрема pid — ідентифікатор процесу);
- стан процесу (виконання, очікування тощо);
- покажчики на структури предка і нащадків;
- час створення процесу та загальний час виконання (так звані таймери процесу); 4- стан процесора (вміст регістрів і лічильник інструкцій); 4 атрибути безпеки процесу (uid, gid, euid, egid).

Створення процесу

В UNIX-сумісних системах процеси створює системний виклик fork(). Його реалізація в Linux:

1. Виділяють пам'ять для нового керуючого блоку процесу (task_struct). Якщо пам'яті недостатньо, повертається помилка.
2. Усі значення зі структури даних предка копіюють у структуру даних нащадка. Після цього поля, значення яких мають відрізнятися від вихідних, будуть змінені.

   Якщо користувач перевищить заданий для нього ліміт кількості процесів або якщо кількість процесів у системі перевищить максимально можливе значення (яке залежить від обсягу доступної пам'яті), створення процесу припиняється і повертається помилка.
3. Для процесу генерується ідентифікатор (pid), при цьому використовують спеціальний алгоритм, що гарантує унікальність.
4. Для нащадка копіюють необхідні додаткові структури даних предка (таблицю

дескрипторів файлів, відомості про поточний каталог, таблицю оброблювачів сигналів тощо).

5. Формують адресний простір процесу.
6. Структуру даних процесу поміщають у список і хеш-таблицю процесів системи.
7. Процес переводять у стан готовності до виконання.

Опис fork() відповідно до POSIX

```
#include <unistd.h>
pid_t fork();      // тип pid_t є цілим
```

Звичайний код роботи з fork () має такий вигляд:

```
pid_t pid;
if ((pid = fork()) == -1) { /* помилка, аварійне завершення */ }
if (pid == 0) {
    // це нащадок
}
else {
    // це предок
    printf ("нащадок запущений з кодом %d\n", pid);
}
```

Завершення процесу

Для завершення процесу в UNIX-системах використовують системний виклик _exit().Під час його виконання відбуваються такі дії.

1. Стан процесу стає TASK_ZOMBIE.
2. Процес повідомляє предків і нащадків про те, що він завершився (за допомогою спеціальних сигналів).
3. Звільняються ресурси, виділені під час виклику fork().
4. Планувальника повідомляють про те, що контекст можна перемикати.

```
#include<unistd.h>
void _exit(int status); // status задає код повернення
#include<stdlib.h>
void exit(int status);  // status задає код повернення
exit (128);             // вихід з кодом повернення 128
```

Запуск програми

Запуск нових програм в UNIX відокремлений від створення процесів і реалізований за допомогою системних викликів сімейства `exec()`. На практиці звичайно реалізують один виклик (у Linux це – `execve()`).

```
#include <unistd.h>
int execve( const char *filename. // повне ім'я файлу
            const char *argv[].   // масив аргументів командного рядка
            const char *envp[]);  // масив змінних оточення
```

Приклад:

```
char *prog = "./child";
char *args[] = { "./child". "arg1". NULL };
char *env[] = { NULL };
if (execve (prog. args. env) == -1) {
    printf ("помилка під час виконання програми: %s\n". prog);
    exit (-1);
}
```

Реалізація технології' fork+exec:

```
if ((pid = fork()) < 0) exit (-1);
if (pid == 0) {  // нащадок завантажує замість себе іншу програму
    // ... завдання prog, args і env
    if (execve (prog, args, env) == -1) {
        printf("помилка під час запуску нащадка\n");
        exit (-1);
    }
}
else {
    // предок продовжує роботу (наприклад, очікує закінчення нащадка)
}
```

Очікування завершення процесу

Коли процес завершується, його керуючий блок не вилучається зі списку й хеша процесів негайно, а залишається там доти, поки інший процес (предок) не видалить його звідти. Якщо процес насправді в системі відсутній (він завершений), а є тільки його керуючий блок, то такий процес називають *процесом-зомбі* (zombie process).

Системний виклик waitpid()

Для вилучення із системи інформації про процес в Linux можна використати описаний вище системний виклик wait(), але частіше застосовують його більш універсальний варіант – waitpid(). Цей виклик перевіряє, чи є керуючий блок відповідного процесу в системі. Якщо він є, а процес не перебуває у стані зомбі (тобто ще виконується), то процес у разі виклику waitpid() переходить у стан очікування. Коли ж процес-нащадок завершується, предок виходить зі стану очікування і вилучає керуючий блок нащадка. Якщо предок не викличе waitpid() для нащадка, той може залишитися у стані зомбі надовго.

```
#include<sys/wait.h>
pid_t waitpid(pid_t pid,     // pid процесу, який очікуємо
              int *status,   // інформація про статус завершення нащадка
              int options); // задаватимемо як 0
```

Синхронне виконання процесів у прикладних програмах

```
pid_t pid;
if ((pid = fork()) == -1) exit(-1);
if (pid == 0) {
    // нащадок – виклик exec()
}
else {
    //  предок – чекати нащадка
    int status;
    waitpid (pid, &status, 0);
    // продовжувати виконання
}
```

Сигнали

За умов багатозадачності виникає необхідність сповіщати процеси про події, що відбуваються в системі або інших процесах. Найпростішим механізмом такого сповіщення, визначеним POSIX, є *сигнали.* Сигнали діють на кшталт програмних переривань: процес після отримання сигналу негайно реагує на нього викликом спеціальної функції - *оброблювача* цього сигналу (signal handler) або виконанням дії за замовчуванням для цього сигналу. Із кожним сигналом пов'язаний його номер, що є унікальним у системі. Жодної іншої інформації разом із сигналом передано бути не може.

Сигнали є найпростішим механізмом міжпроцесової взаємодії в UNIX-системах, але, оскільки з їхньою допомогою можна передавати обмежені дані, вони переважно використовуються для повідомлення про події.

Типи сигналів

Залежно від обставин виникнення сигнали поділяють на синхронні й асинхронні. *Синхронні сигнали* виникають під час виконання активного потоку процесу (зазвичай через помилку - доступ до невірної ділянки пам'яті, некоректну роботу із плаваючою крапкою, виконання неправильної інструкції процесора). Ці сигнали генерує ядро ОС і негайно відправляє їх процесу, потік якого викликав помилку.

Асинхронні сигнали процес може отримувати у будь-який момент виконання:

- програміст може надіслати асинхронний сигнал іншому процесу, використовуючи системний виклик, який у POSIX-системах називають kill(), параметрами цього виклику є номер сигналу та ідентифікатор процесу;
- причиною виникнення сигналу може бути також деяка зовнішня подія (натискання користувача на певні клавіші, завершення процесу-нащадка тощо).

Диспозиція сигналів

На надходження сигналу процес може реагувати одним із трьох способів (спосіб реакції процесу на сигнал називають *диспозицією* сигналу):

- викликати оброблювач сигналу;
- проігнорувати сигнал, який у цьому випадку «зникне» і не виконає жодної дії;
- використати диспозицію, передбачену за замовчуванням (така диспозиція задана для кожного сигналу, найчастіше це - завершення процесу).

Процес може задавати диспозицію для кожного сигналу окремо.

Приклади сигналів

Назвемо сигнали, що визначені POSIX і підтримуються в Linux (у дужках поруч з ім'ям сигналу наведено його номер).

До синхронних сигналів належить, наприклад, сигнал SIGSEGV (11), який генерує система під час записування в захищену ділянку пам'яті.

До асинхронних сигналів належать:

- SIGHUP (1) - розрив зв'язку (наприклад, вихід користувача із системи);
- SIGINT і SIGQUIT (2,3) — сигнали переривання програми від клавіатури (генеруються під час натискання користувачем відповідно Ctri+C і Ctrl+\);
- SIGKILL (9) — негайне припинення роботи програми (для такого сигналу не можна змінювати диспозицію);
- SIGUSR1 і SIGUSR2 (10,12) — сигнали користувача, які можуть використовувати прикладні програми;
- SIGTERM (15) - пропозиція програмі завершити її роботу (цей сигнал, на відміну від SIGKILL, може бути зігнорований).

Задання диспозиції сигналів

Для задання диспозиції сигналу використовують системний виклик sigaction().

```
#include <signal.h>
int sigaction(int signum,   // номер сигналу
   struct sigaction *action,   // нова диспозиція
   struct sigaction *old_action); // повернення попередньої
   диспозиції
```

Диспозицію описують за допомогою структури sigaction з такими полями:

- sa_handler — покажчик на функцію-оброблювач сигналу;
- sa_mask — маска сигналу, що задає, які сигнали будуть заблоковані всередині оброблювача;
- sa_flag - додаткові прапорці.

Обмежимося обнуліннням sa_mask і sa_flag (не блокуючи жодного сигналу):

```
struct sigaction action = {0};
```

Поле sa_handler має бути задане як покажчик на оголошену раніше функцію, що має такий вигляд:

```
void user_handler (int signum) {
   // обробка сигналу }
```

Очікування сигналу

```
// задати оброблювачі за допомогою sigaction()
pause();   // чекати сигналу
```

Генерування сигналів

```
#include<signal.h>
int kill (pid_t pid.   // ідентифікатор процесу
          int signum); // номер сигналу
```

Приклад:

```
kill (atoi(argv[1]). SIGHUP);
```

Керування потоками в Linux

“Історичний” спосіб -- системний виклик clone(), який дає можливість створити новий процес на базі наявного. Особливості використання:

♦ Для нового процесу потрібно задати спеціальний набір прапорців, що визначають, як будуть розподілятися ресурси між предком і нащадком. До ресурсів, які можна розділяти, належать адресний простір, інформація про відкриті файли, оброблювачі сигналів. Зазначимо, що у традиційній реалізації clone() відсутнє спільне використання ідентифікатора процесу (pid), ідентифікатора предка та деяких інших важливих атрибутів.

♦ Для нового процесу потрібно задати новий стек (оскільки внаслідок виклику clone() два процеси можуть спільно використовувати загальну пам'ять, для них неприпустиме використання загального стека).

Підтримка clone() означає реалізацію багатопотоковості за схемою 1:1. При цьому первинними в системі є процеси, а не потоки (у Linux послідовності інструкцій процесора, з якими працює ядро, називають процесами, а не потоками ядра).

Традиційна реалізація багатопотоковості в Linux визначає, що потоки користувача відображаються на процеси в ядрі. Тому в цьому розділі недоцільно розглядати окремо структури даних потоку в Linux — він відображається такою ж самою структурою task_struct, як і процес.

Нова реалізація багатопотоковості в ядрі Linux — NPTL (Native POSIX Threads Library).

Програмний інтерфейс керування потоками POSIX

Створення потоків POSIX

Щоб додати новий потік у поточний процес, у POSIX використовують функцію pthread_create() із таким синтаксисом:

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

Розглянемо параметри цієї функції:

- th — покажчик на заздалегідь визначену структуру типу pthread_t, яка далі буде передана в інші функції роботи з потоками; далі називатимемо її *дескриптором потоку* (thread handle);
- attr — покажчик на структуру з атрибутами потоку (для використання атрибутів за замовчуванням потрібно передавати нульовий покажчик);
- thread_fun — покажчик на функцію потоку, що має описуватися як

  void *mythread_fun(void *value) { //виконання коду потоку }

- arg — дані, що передаються у функцію потоку (туди вони потраплять як параметр value).

Приклад створення потоку POSIX:

```
#include <pthread.h>
// функція потоку
void *thread_fun(void *num) {
    printf ("потік номер %d\n", (int)num);
}
// ...
pthread_t th;
// створюємо другий потік
pthread_create (&th, NULL, thread_fun, (void *)++thread_num);
// тут паралельно виконуються два потоки
```

Завершення потоків POSIX

Для завершення потоку достатньо дійти до кінця його функції потоку thread_fun() або викликати з неї pthread_exit():

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

Параметр retval задає код повернення. Приклад завершення потоку:

```
void *thread_fun(void *num) {
```

```
    printf ("потік номер %d\n),  (int)num);
    pthread_exit (0); }
```

Для коду початкового потоку програми є дві різні ситуації:

1- виконання оператора return або виконання всіх операторів до кінця main() *завершує весь процес,* при цьому всі потоки теж завершують своє виконання;

2 - виконання функції pthread_exit() усередині функції main() завершує тільки початковий потік, ця дія не впливає на інші потоки у програмі - вони продовжуватимуть виконання, поки самі не завершаться.

Виклик функції pthread_exit() використовують тільки для приєднуваних потоків, оскільки він не звільняє ресурси потоку — за це відповідає той, хто приєднає потік. Насправді, якщо такий потік не буде приєднаний негайно після завершення, його інформація залишиться в системі й може бути зчитана пізніше.

Приєднання потоків POSIX

Для очікування завершення виконання потоків використовують функцію pthread _join(). Вона має такий синтаксис:

```
int pthread_join(pthread_t th, void **status);
```

Ця функція блокує потік, де вона була викликана, доти, поки потік, заданий дескриптором th, не завершить своє виконання. Потік, на який очікують, має існувати в тому самому процесі й не бути від'єднаним. Якщо status не є нульовим покажчиком, після виклику він вказуватиме на дані, повернені потоком (аргумент функції pthread_exit()). Якщо потік уже завершився до моменту виклику функції pthread_join(), його інформація має бути зчитана

негайно.

Після приєднання потоку інформацію про нього повністю вилучають із системи, тому коли кілька потоків очікували на той самий потік, його приєднає тільки один із них, для решти буде повернено помилку (до того моменту потрібного потоку в системі вже не існуватиме). Таких ситуацій краще уникати.

Приклад приєднання потоку. Припустимо, що треба асинхронно запустити доволі тривалий запит до бази даних, виконати паралельно з ним деяку роботу (наприклад, відобразити індикатор виконання), після чого дочекатися результатів запиту. Пропоноване рішення зводиться до створення окремого потоку для виконання запиту (потоку-помічника, helper thread) і до очікування його завершення у базовому потоці. Приблизний програмний код:

```
struct result_data {
  // інформація з бази даних
};
void *query_fun(void *retval) {
  struct result_data db_result;
  // одержати дані з бази (search_db() тут не задаємо)
  db_result = search_db(...);
  // визначаємо результат
  *(struct result_data *)retval = db_result;
```

```
    pthread_ex1t(0);
  }
  int main () {
    pthread_t query_th;
    void *status;
    struct resu1t_data query_res;
    // почати виконувати запит паралельно з основним кодом main()
pthread_create(&query_th, NULL, query_fun, &query_res);
    // вчинити деякі дії паралельно з виконанням запиту
    // тут два потоки виконуються паралельно
    pthread_join(query_th, &status); // дочекатися результату запиту
    // тепер можна безпечно використати результат запиту query_res
    return 0;
}
```

Від'єднані потоки. Задання атрибутів потоків POSIX

Для того, щоб отримати від'єднаний потік, треба задати відповідні *атрибути потоку,* виконавши такі дії.

1. Описати змінну для їхнього зберігання (атрибути потоку відображає структура даних pthread_attr_t):

```
pthread_attr_t attr;
```

2. Ініціалізувати цю змінну так:

```
pthread_attr_init(&attr);
```

3. Задати атрибут за допомогою виклику спеціальної функції (різним атрибутам відповідають різні функції).

Для задання атрибута, який визначає, чи буде цей потік від'єднаним чи ні, використовують таку функцію:

```
pthread_attr_setdetachstate(pthread_attr_t attr, int detachstate);
```

Другим параметром може бути одне зі значень: PTHREAD_CREATE_JOINABLE (приєднуваний потік); PTHREAD_CREATE_DETACHED (від'єднаний потік).

Приклад задання атрибута і його використання під час створення від'єднаного потоку:

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&th, &attr, ...);
```

Завершуються такі потоки виходом із функції потоку.

Керування процесами у Windows XP

Поняття процесу й потоку у Windows XP чітко розмежовані. Процеси в даній системі визначають «поле діяльності» для потоків, які виконуються в їхньому адресному просторі. Серед ресурсів, з якими процес може працювати прямо, відсутній процесор — він доступний тільки потокам цього процесу. Процес, проте, може задати початкові характеристики для своїх потоків і тим самим вплинути на їхнє виконання.

Складові елементи процесу

♦ Адресний простір процесу складається з набору адрес віртуальної пам'яті, які він може використати. Ці адреси можуть бути пов'язані з оперативною пам'яттю, а можуть — з відображеними у пам'ять ресурсами. Адресний простір процесу недоступний іншим процесам.

♦ Процес володіє системними ресурсами, такими як файли, мережні з'єднання, пристрої введення-виведення, об'єкти синхронізації тощо.

♦ Процес містить деяку стартову інформацію для потоків, які в ньому створюватимуться. Наприклад, це інформація про базовий пріоритет і прив'язання до процесора.

♦ Процес має містити хоча б один потік, який система скеровує на виконання. Без потоків у Windows XP наявність процесів неможлива.

Створення процесів

У Win32 API прийнято модель запуску застосування за допомогою одного виклику, який створює адресний простір процесу і завантажує в нього виконуваний файл. Окремо функціональність fork() і exec() у цьому API не реалізована.

Такий виклик реалізує функція CreateProcess().

Основні кроки створення нового процесу із використанням функції CreateProcess().

1. Відкривають виконуваний файл, що його ім'я задане як параметр. При цьому ОС визначає, до якої підсистеми середовища він належить. Коли це виконуваний файл Win32, то його використовують прямо, для інших підсистем відшукують необхідний файл підтримки (наприклад, процес підсистеми POSIX для POSIX-застосувань).

2. Створюють об'єкт-процес у виконавчій системі Windows XP. При цьому виконують такі дії:

   а) створюють та ініціалізують структури даних процесу (блоки EPROCESS, KPROCESS, PEB);

   б) створюють початковий адресний простір процесу;

   в) блок процесу поміщають у кінець списку активних процесів, які підтримує система.

3. Створюють початковий потік процесу. Послідовність дій під час створення потоку розглядатимемо під час вивчення підтримки потоків у Windows XP.

4. Після створення початкового потоку підсистемі Win32 повідомляють про новий процес і його початковий потік. Це повідомлення містить їхні дескриптори (handles) —

унікальні числові значення, що ідентифікують процес і потік для засобів режиму користувача. Підсистема Win32 виконує низку дій після отримання цього повідомлення (наприклад, задає пріоритет за замовчуванням) і поміщає дескриптори у свої власні таблиці процесів і потоків.

5. Після надсилання повідомлення розпочинають виконання початкового потоку (якщо він не був заданий із прапорцем відкладеного виконання).

6. Завершують ініціалізацію адресного простору процесу (наприклад, завантажують необхідні динамічні бібліотеки), після чого починають виконання завантаженого програмного коду.

Кожен процес може користуватися ресурсами через дескриптори відповідних об'єктів. Відкриті дескриптори об'єктів є індексами в *таблиці об'єктів* (object table), що зберігається в керуючому блоці процесу. Ця таблиця містить покажчики на всі об'єкти, дескриптори яких відкриті процесом.

Програмний інтерфейс керування процесами Win32 API

Деякі базові типи:

B00L — його використовують для зберігання логічного значення, насправді він є цілочисловим;

DWORD — 4-байтовий цілочисловий тип без знака, аналог unsigned int;

HANDLE — цілочисловий дескриптор об'єкта;

LPTSTR — покажчик на рядок, що складається із двобайтових або однобайтових символів;

LPCTSTR — покажчик на константний рядок, аналог const char * або const wchar_t *.

Створення процесів у Win32 API

Для створення нового процесу у Win32 використовують функцію CreateProcess().

```
BOOL CreateProcess ( LPCTSTR app_name, LPCTSTR cmd_line,
    LPSECURITY_ATTRIBUTES psa_proc, LPSECURITY_ATTRIBUTES psa_thr,
    BOOL inherit_handles, DWORD flag_create,
    LPVOID environ, LPTSTR cur_dir,
    LPSTARTUPINFO startup_info. LPPROCESS_INFORMATION process_info );
```

де: app_name — весь шлях до виконуваного файла (NULL — ім'я виконуваного файла можна отримати з другого аргументу):

```
CreateProcess("C:/winnt/notepad.exe", ...);
```

cmd_line - повний командний рядок для запуску виконуваного файла, можливо, із параметрами (цей рядок виконується, якщо app_name дорівнює NULL, зазвичай так запускати процес зручніше):

```
CreateProcess(NULL. "C:/winnt/notepad.exe test.txt", ...);
```

psa_proc, psa_thr — атрибути безпеки для всього процесу і для головного потоку (NULL означає задання атрибутів безпеки за замовчуванням);

inherit_handles — керує спадкуванням нащадками дескрипторів об'єктів, які використовуються у процесі;

flag_create — маска прапорців, які керують створенням нового процесу (наприклад, прапорець CREATE_NEW_CONSOLE означає, що процес запускається в новому консольному вікні);

environ — покажчик на пам'ять із новими змінними оточення, які предок може задавати для нащадка (NULL — нащадок успадковує змінні оточення предка);

cur_dir — рядок із новим значенням поточного каталогу для нащадка (NULL -нащадок успадковує поточний каталог предка);

startup_info — покажчик на заздалегідь визначену структуру даних типу STARTUPINFO, на базі якої задають параметри для процесу-нащадка;

process_info — покажчик на заздалегідь визначену структуру даних PR0CESS_INFORMATION, яку заповнює ОС під час виклику CreateProcess().

Серед полів структури STARTUPINFO можна виділити:

– cb — розмір структури у байтах (це її перше за порядком поле). Звичайно перед заповненням всю структуру обнуляють, задаючи тільки cb:

```
STARTUPINFO si = { sizeof(si) };
```

– lpTitle - рядок заголовка вікна для нової консолі:

```
// ... si обнуляється
si.lpTitle - "Мій процес-нащадок";
```

Структура PR0CESS_I NF0RMATI0N містить чотири поля:

- hProcess — дескриптор створеного процесу;
- hThread - дескриптор його головного потоку;
- dwProcessId - ідентифікатор процесу (process id, pid);
- dwThreadld — ідентифікатор головного потоку (thread id, tid).

CreateProcess() повертає нуль, якщо під час запуску процесу сталася помилка.

Приклад виклику CreateProcess(), у якому вказані значення всіх необхідних параметрів:

```
// ... задається si
PROCESS_INFORMATION pi:
CreateProcess (NULL, "C:/winnt/notepad test.txt", NULL, NULL, TRUE,
CREATE_NEW_CONSOLE, NULL, “D:/”, &si, &pi);
printf ("pid=%d, tid=%d\n", pi.dwProcessId,pi.dwThreadld);
CloseHandle(pi.hThread);
CloseHandle(pi.hProcess);
```

Завершення процесів у Win32 API

Для завершення процесів використовують функцію ExitProcess():

VOID ExitProcess (UINT exitcode);

де exitcode - код повернення процесу. Наприклад

ExitProcess (100); // вихід з кодои 100

Для завершення іншого процесу використовують функцію TerminateProcess():

BOOL TerminateProcess (HANDLE hProcess. UINT exitcode);

Синхронне й асинхронне виконання процесів у Win32 API

Для того, щоб реалізувати синхронне виконання, після успішного виконання CreateProcess() процес-предок має викликати функцію очікування закінчення нащадка. Це WaitForSingleObject(), стандартна функція очікування зміни стану об'єкта Win32 API.

```
DWORD WaitForSingleObject (HANDLE ph, DWORD timeout):
```

Тут ph — дескриптор нащадка; timeout — максимальний час очікування в мілісекундах (INFINITE — необмежено). Повернене значення може бути WAIT_FAILED через помилку.

Приклад:

```
BOOL res;
if (res = CreateProcess(... , &pi)) {
CloseHandle(pi.hThread);
    if (WaitForSingleObject(pi.hProcess. INFINITE)!=WAITJAILED)
    GetExitCodeProcess(pi.hProcess, &exitcode);
  }
  CloseHandle(pi.hProcess);
}
```

Для асинхронного виконання достатньо відразу ж закрити обидва дескриптори і не викликати WaitForSingleObjectO:

```
if (res = CreateProcess(... , &pi)) {
  CloseHandle(pi.hThread);
CloseHandle(pi.hProcess); }
```

Керування потоками у Windows XP

Для того щоб виконувати код, у рамках процесу обов'язково необхідно створити потік. У системі Windows XP реалізована модель потоків «у чистому вигляді».
Багатопотоковість Windows XP базується на схемі 1:1. Процеси не плануються.

Складові елементи потоку

Потік у Windows XP складається з таких елементів:

- вмісту набору регістрів, який визначає стан процесора;
- двох стеків - один використовують для роботи в режимі користувача, інший — у режимі ядра; ці стеки розміщені в адресному просторі процесу, що створив цей потік;
- локальної пам'яті потоку (TLS);
- унікального ідентифікатора потоку (thread id, tid), який вибирають із того самого простору імен, що й ідентифікатори процесів.

Сукупність стану процесора, стеків і локальної пам'яті потоку становить *контекст потоку.* Кожний потік має власний контекст. Усі інші ресурси процесу (його адресний простір, відкриті файли тощо) спільно використовуються потоками.

Створення потоків

Основним засобом створення потоків у Windows XP є функція CreateThread() Win32 API. Етапи виконання цієї функції.

1. В адресному просторі процесу створюють стек режиму користувача для потоку.
2. Ініціалізують апаратний контекст потоку (у процесор завантажують дані, що визначають його стан). Цей крок залежить від архітектури процесора.

3. Створюють об'єкт-потік виконавчої системи у призупиненому стані, для чого в режимі ядра:

а) створюють та ініціалізують структури даних потоку (блоки ETHREAD, KTHREAD, TEB);
б) задають стартову адресу потоку (використовуючи передану як параметр адресу процедури потоку);

в) задають інформацію для підсистеми безпеки та ідентифікатор потоку;

г) виділяють місце під стек потоку ядра.

4. Підсистемі Win32 повідомляють про створення нового потоку.

5. Дескриптор та ідентифікатор потоку повертають у процес, що ініціював створення потоку (викликав CreateThread()).
6. Починають виконання потоку (виконують перехід за стартовою адресою).

Створення потоків у Win32 API

У Win32 API для створення потоку призначена функція CreateThread(), а для його завершення - EndThread().

На практиці пару CreateThread()/EndThread() є сенс використати лише тоді, коли з коду, що виконує потік, не викликаються функції стандартної бібліотеки мови С (такі, як printf() або strcmp()).

Річ у тому, що функції стандартної бібліотеки С у Win32 API не пристосовані до використання за умов багатопотоковості, і для того щоб підготувати потік до роботи за таких умов, необхідно під час його створення і завершення виконувати деякі додаткові дії. Ці дії враховані у спеціальних бібліотечних функціях роботи з потоками, описаних у заголовному файлі process.h. Це функція _beginthreadex() для створення потоку й _endthreadex() — для завершення потоку.

Розглянемо синтаксис функції _beginthreadex().

```
#include <process.h>
unsigned long _beginthreadex( void *security, unsigned stack_size,
  unsigned WINAPI (*thread_fun)(void *),
  void *argument., unsigned init_state, unsigned *tid );
```

security — атрибути безпеки цього потоку (NULL - атрибути безпеки за замовчуванням);

stack_size — розмір стека для потоку (зазвичай 0, у цьому разі розмір буде таким самим,

що й у потоку, який викликає _beginthreadex());

thread_fun — покажчик на функцію потоку;

argument — додаткові дані для передачі у функцію потоку;

init_state - початковий стан потоку під час створення (0 для потоку, що почне виконуватися негайно, CREATE_SUSPEND для припиненого);

tid - покажчик на змінну, в яку буде записано ідентифікатор потоку після виклику

(0, якщо цей ідентифікатор не потрібний).

Функція _beginthreadex() повертає дескриптор створеного потоку, який потрібно перетворити в тип HANDLE:

```
HANDLE th = (HANDLE)_beginthreadex( … );
```

Після отримання дескриптора, якщо він у цій функції більше не потрібний, його закривають за допомогою CloseHandle() аналогічно до дескриптора процесу:

```
CloseHandle(th);
```

Приклад задання функції потоку. Додаткові дані, які передаються під час виклику _beginthreadex() за допомогою параметра argument, доступні в цій функції через параметр типу void *.

```
unsigned int WINAPI thread_fun (void *num) {
    printf ("потік %d почав виконання\n", (int)num);
    // код функції потоку
printf ("потік %d завершив виконання\n", (int)num): }
```

Приклад виклику _beginthreadex() з усіма параметрами:

```
unsigned tid:
int number = 0:
HANDLE th = (HANDLE)_beginthreadex (
            NULL, 0, thread_fun, (void *)++number, 0, &tid);
```

Після створення потоку може виникнути потреба змінити його характеристики. Якщо це необхідно зробити у функції потоку, варто знати, як отримати доступ до його дескриптора. Для цього використовують функцію GetCurrentThread():

```
unsigned int WINAPI thread_fun (void *num) {
HANDLE curth = GetCurrentThread(); }
```

Завершення потоків у Win32 API

Функцію потоку можна завершити двома способами.

1. Виконати у ній звичайний оператор return (цей спосіб є найнадійнішим):

```
unsigned WINAPI thread_fun(void *num) {
return 0; }
```

2. Викликати функцію _endthreadex() з параметром, що дорівнює коду повернення:

```
unsigned WINAPI thread_fun (void *num) {
endthreadex(0): }
```

Приєднання потоків у Win32 API

Приєднання потоків у Win32 API, подібно до очікування завершення процесів, здійснюється за допомогою функції WaitForSingleObject(). Базовий синтаксис її використання з потоками такий:

```
HANDLE th = (HANDLE)_beglnthreadex (…);
if (WaitForSingleObject(th, INFINITE) != WAIT_FAILED) {
// потік завершений успішно
} CloseHandle(th);
```

Висновки

♦ Процеси і потоки є активними ресурсами обчислювальних систем, які реалізують виконання програмного коду. Потоком називають набір послідовно виконуваних команд процесора. Процес є сукупністю одного або декількох потоків і захищеного адресного простору, в якому вони виконуються. Потоки одного процесу можуть разом використовувати спільні дані, для потоків різних процесів без використання спеціальних засобів це неможливо. У традиційних системах кожний процес міг виконувати тільки один потік, виконання програмного коду пов'язували із процесами. Сучасні ОС підтримують концепцію багатопотоковості.

♦ Використання потоків у застосуванні означає внесення в нього паралелізму -можливості одночасного виконання дій різними фрагментами коду. Паралелізм у програмах відображає асинхронний характер навколишнього світу, його джерелами є виконання коду на декількох процесорах, операції введення-виведення, взаємодія з користувачем, запити застосувань-клієнтів. Багато-потоковість у застосуваннях дає змогу природно реалізувати цей паралелізм і домогтися високої ефективності. З іншого боку, використання багатопотоковості досить складне і вимагає високої кваліфікації розробника.

♦ Розрізняють потоки користувача, які виконуються в режимі користувача в адресному просторі процесу, і потоки ядра, з якими працює ядро ОС. Взаємовідношення між ними визначають схему реалізації моделі потоків. На практиці найчастіше використовують схему 1:1, коли кожному потоку користувача відповідає один потік ядра, і саме ядро відповідає за керування потоками користувача.

♦ Потік може перебувати в різних станах (виконання, очікування, готовності тощо). Принципи переходу з одного стану в інший залежать від принципів планування потоків і планування процесорного часу. Перехід процесу зі стану виконання у будь-який інший стан зводиться до перемикання контексту — передачі керування від одного потоку до іншого зі збереженням стану процесора.

♦ Кожному процесу і потоку в системі відповідає його керуючий блок - структура даних, що містить усю необхідну інформацію. Під час створення процесу (зазвичай за допомогою системного виклику fork O) створюють його керуючий блок, виділяють пам'ять і запускають основний потік. Створення потоку простіше і виконується швидше, оскільки не потрібно виділяти пам'ять під новий адресний простір.