ЛЕКЦІЯ 4. КЕРУВАННЯ ПРОЦЕСАМИ І ПОТОКАМИ

4.1. Створення і завершення процесів і потоків

Засоби створення і завершення процесів дають змогу динамічно змінювати в операційній системі набір застосувань, що виконуються. Засоби створення і завершення потоків є основою для створення багатопотокових програм.

Базові принципи створення і завершення процесів

Процеси можуть створюватися ядром системи під час її ініціалізації. Наприклад, в UNIX-сумісних системах таким процесом може бути процес ініціалізації системи `init`, у Windows – процеси підсистем середовища (Win32 або POSIX). Таке створення процесів, однак, є винятком, а не правилом.

Найчастіше процеси створюються під час виконання інших процесів. Нові процеси можуть бути створені під час роботи застосування відповідно до його логіки (компілятор може створювати процеси для кожного етапу компіляції, веб-сервер – для обробки прибулих запитів) або безпосередньо за запитом користувача (наприклад, з командного інтерпретатора, графічної оболонки або файлового менеджера).

Розрізняють два типи процесів з погляду їхньої взаємодії із користувачем.

- *Інтерактивні* процеси взаємодіють із користувачами безпосередньо, приймаючи від них дані, введені за допомогою клавіатури, миші тощо. Прикладом інтерактивного процесу може бути процес текстового редактора або інтегрованого середовища розробки.
- *Фонові* процеси із користувачем не взаємодіють безпосередньо. Зазвичай вони запускаються під час старту системи і чекають на запити від інших застосувань. Деякі з них (системні процеси) підтримують функціонування системи (реалізують фонове друкування, мережні засоби тощо), інші виконують спеціалізовані задачі (реалізують веб-сервери, сервери баз даних тощо). Фонові процеси також називають службами (services, у системах лінії Windows) або демонами (daemons, в UNIX).

Оскільки основним елементом процесу є захищений адресний простір, дуже важливо вирішити проблему його розподілу під час створення нового процесу. Розглянемо два різні підходи.

У *першому підході* адресний простір нового процесу (нащадка) створюють як точну копію адресного простору процесу, який створив новий поцес (предка). Така операція реалізована системним викликом, який у POSIX-системах називають `fork()`.

У цьому разі копіюється не тільки адресний простір, а й лічильник команд головного потоку процесу, тому після виклику `fork()` предок і нащадок виконуватимуть ту саму інструкцію. Розробник має визначити, у якому з двох процесів перебуває керування. Це можна зробити на підставі відмінностей між кодами повернення `fork()` для предка і нащадка.

У *другому підході*, коли створення нового процесу відбувається шляхом дублювання адресного простору предка, виникає потреба у спеціальних засобах для завантаження програмного коду в адресний простір процесу. Такі засоби реалізує системний виклик,

який у POSIX-системах називають `exec()`. Як параметр для виклику `exec()` треба вказувати весь шлях до виконуваного файла програми, який буде завантажено у пам'ять.

У системах із підтримкою `fork()` для того, щоб запускати на виконання програми, після виклику `fork()` потрібно негайно викликати `exec()` (це називають технологією `fork+exec`).

Підхід із використанням `fork()` і `exec()` є гнучкішим, бо він дає змогу в разі необхідності обмежитись якимось одним етапом запуску застосування. Сучасні ОС переважно реалізують деяку комбінацію першого та другого підходів.

Розглянемо ***три варіанти завершення процесів***.

Процес *завершується* ***коректно*** (самостійно) після виконання своєї задачі (для інтерактивних процесів це нерідко відбувається з ініціативи користувача, який, приміром, скористався відповідним пунктом меню). 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 *завершується* ***аварійно*** через помилку. Такий вихід може бути передбачений програмістом (під час обробки помилки приймається рішення про те, що далі продовжувати виконання програми неможливо), а може бути наслідком генерування переривання (ділення на нуль, доступу до захищеної області пам'яті тощо).

Процес *завершується* ***іншим процесом*** або ядром системи. Наприклад, перед закінченням роботи ОС ядро припиняє виконання всіх процесів. Процес може припинити виконання іншого процесу за допомогою системного виклику, який у POSIX-системах називають `kill()`.

Після того як процес завершить свою роботу, пам'ять, відведена під його адресний простір, звільняється і може бути використана для інших потреб. Усі потоки цього процесу теж припиняють роботу. Якщо у даного процесу є нащадки, їхня робота переважно не припиняється слідом за роботою предка. Інтерактивні процеси звичайно завершуються у разі виходу користувача із системи.

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()`.)

Базові принципи створення і завершення потоків

Процедура ***створення потоків*** має свої особливості, пов'язані насамперед із тим, що потоки створюють у рамках вже існуючого адресного простору (конкретного процесу або, можливо, ядра системи). Існує кілька ситуацій, у яких може бути створений новий потік.

Якщо процес створюється за допомогою системного виклику `fork()`, після розподілу адресного простору автоматично створюється потік усередині цього процесу (найчастіше це – головний потік застосування).

Можна створювати потоки з коду користувача за допомогою відповідного системного виклику.

У багатьох ОС є спеціальні потоки, які створює ядро системи (код ядра теж може виконуватися в потоках).

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.

Під час ***завершення потоку*** його ресурси вивільняються (насамперед, стек); ця операція звичайно виконується швидше, ніж завершення процесу. Потік може бути завершений, коли керування дійде до кінця процедури потоку; є також спеціальні системні виклики, призначені для дострокового припинення виконання потоків.

4.2. Перемикання контексту й обробка переривань

Найважливішим завданням операційної системи під час керування процесами і потоками є організація ***перемикання контексту*** – передачі керування від одного потоку до іншого зі збереженням стану процесора.

Загальних принципів перемикання контексту дотримуються у більшості систем, але їхня реалізація обумовлена конкретною архітектурою. Звичайно потрібно виконати такі операції:

- зберегти стан процесора потоку в деякій ділянці пам'яті (області зберігання стану процесора потоку);
- визначити, який потік слід виконувати наступним;
- завантажити стан процесора цього потоку із його області зберігання;
- продовжити виконання коду нового потоку.

У процесі виконання ***потік може бути перерваний*** не лише для перемикання контексту на інший потік, але й у зв'язку із програмним або апаратним перериванням (перемикання контексту теж пов'язане із перериваннями, власне, із перериванням від таймера). Із кожним перериванням надходить додаткова інформація (наприклад, його номер). На підставі цієї інформації система визначає, де буде розміщена адреса процедури оброблювача переривання (список таких адрес зберігають у спеціальній ділянці пам'яті і називають *вектором переривань).*

Наведемо приклад послідовності дій під час обробки переривання:

- збереження стану процесора потоку;
- встановлення стека оброблювача переривання;
- початок виконання оброблювача переривання (коду операційної системи); для цього з вектора переривання завантажується нове значення лічильника команд;
- відновлення стану процесора потоку після закінчення виконання оброблювача і продовження виконання потоку.

4.3. Керування процесами у Windows

Поняття процесу й потоку у Windows 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.

Розглянемо базові 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 наявність процесів неможлива.

Для виконавчої системи Windows кожний процес зображається об'єктом-процесом виконавчої системи (executive process object); його також називають *керуючим блоком процесу* (executive process block, EPROCESS). Для ядра системи процес зображається об'єктом-процесом ядра (kernel process object), його також називають *блоком процесу ядра* (process kernel block, KPROCESS).

У режимі користувача доступним є *блок оточення процесу* (process environment block, PEB), що перебуває в адресному просторі цього процесу.

Розглянемо структури даних процесу докладніше. Зазначимо, що EPROCESS і KPROCESS, на відміну від PEB, доступні тільки із привілейованого режиму.

Керуючий блок процесу містить такі основні елементи:

- блок процесу ядра (KPROCESS);
- ідентифікаційну інформацію;
- інформацію про адресний простір процесу;
- інформацію про ресурси, доступні процесу, та обмеження на використання цих ресурсів;
- блок оточення процесу (PEB);
- інформацію для підсистеми безпеки. До ідентифікаційної інформації належать:
- ідентифікатор процесу (`pid`);
- ідентифікатор процесу, що створив цей процес (незважаючи на те, що Windows XP не підтримує відносини «предок-нащадок» автоматично, вони можуть бути задані програмним шляхом, тобто нащадок може сам призначити собі предка, задавши цей ідентифікатор);
- ім'я завантаженого програмного файла.

Блок процесу ядра містить усю інформацію, що належить до потоків цього процесу:

- покажчик на ланцюжок блоків потоків ядра, де кожний блок відповідає потоку;
- базову інформацію, необхідну ядру системи для планування потоків (ця інформація буде успадкована потоками, пов'язаними із цим процесом).

Блок оточення процесу містить інформацію про процес, яка призначена для доступу з режиму користувача:

- початкову адресу ділянки пам'яті, куди завантажився програмний файл;
- покажчик на динамічну ділянку пам'яті, доступну процесу.

Цю інформацію може використати завантажувач програм або процес підсистеми Win32.

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()`. Вона не є системним викликом ОС – це бібліотечна функція Win32 API, реалізована в усіх Win32-сумісних системах.

Функція `CreateProcess()` потребує задавання 10 параметрів, докладно їх буде розглянуто далі. Зазначимо, що системні виклики UNIX/POSIX потребують меншої кількості параметрів (як уже зазначалося, `fork()` не використовує жодного параметра, а `exec()` – використовує три параметри).

Наведемо 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. 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.
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.

Для ***створення нового процесу*** у Win32 використовують функцію `CreateProcess()`:

```
BOOL CreateProcess (app_name, cmd_line,
psa_proc, psa_thr,
inherit_handles, flag_create,
environ, cur_dir,
startup_info, 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 test.txt", ...);
```

`psa_proc`, `psa_thr` – атрибути безпеки для всього процесу і для головного потоку (особливості їхнього задавання розглянемо в наступних лекціях, а доти як значення всіх параметрів типу LPSECURITY_ATTRIBUTES задаватимемо NULL, що означає задавання атрибутів безпеки за замовчуванням);

`inherit_handles` – керує спадкуванням нащадками дескрипторів об'єктів, які використовуються у процесі (спадкування дескрипторів буде розглянуто пізніше);

`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 PROCESS_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 PROCESS_INFORMATION містить чотири поля:

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

Ідентифікатор `pid` унікально визначає процес на рівні ОС. ОС повторно використовує `pid` уже завершених процесів, тому небажано запам'ятовувати їхнє значення, якщо процес уже завершився або закінчився помилкою.

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

Наведемо приклад виклику `CreateProcess()`:

```
// ... 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.dwThreadId); CloseHandle(pi.hThread);
  CloseHandle(pi.hProcess);
```

Після створення процесу може виникнути необхідність змінити його характеристики з коду, який він виконує. Для цього треба отримати доступ до дескриптора поточного процесу за допомогою функції `GetCurrentProcess()`:

```
HANDLE curph = GetCurrentProcess();
```

Ідентифікатор поточного процесу можна отримати за допомогою функції `GetCurrentProcessId()`:

```
int pid = GetCurrentProcessId();
```

У разі ***завершення процесу*** відповідний об'єкт-процес стає кандидатом на вилучення із системи. При цьому диспетчер об'єктів викликає метод `delete` для об'єктів-процесів,

який закриває всі дескриптори в таблиці об'єктів цього процесу.

Для завершення процесів використовують функцію `ExitProcess()`:

```
VOID ExitProcess (UINT exitcode);
```

де: `exitcode` – код повернення процесу. Наприклад

```
ExitProcess (100);   // вихід з кодом 100
```

Для ***завершення іншого процесу*** використовують функцію `TerminateProcess()`:

```
BOOL TerminateProcess (HANDLE hProcess, UINT exitcode);
```

У процесі, який запустив інший процес, можна отримати код завершення цього процесу за допомогою функції

```
GetExitCodeProcess (HANDLE hProcess, LPDWORD pexit_code);
```

Тут `pexit_code` – покажчик на змінну, в яку заносять код завершення.

```
int exitcode;
GetExitCodeProcess(pi.hProcess, Sexitcode);
printf ("Код повернення =%d\n", exitcode);
```

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 мілі-секундах (INFINITE – необмежене).

4.4. Керування потоками у Windows

Для того щоб виконувати код, у рамках процесу обов'язково необхідно створити потік. У системі Windows реалізована модель потоків «у чистому вигляді». Процеси і потоки є різними сутностями в системі, що перебувають у чітко визначеному взаємозв'язку один з одним; для роботи з ними використовують різні системні виклики. У Windows ніколи не використовували модель процесів, подібну до традиційної моделі UNIX.

Багатопотоковість Windows базується на схемі 1:1. Кожному потоку користувача відповідає сутність у ядрі, при цьому ядро відповідає за планування потоків. Процеси не плануються.

Потік у Windows 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.

Розрізняють два види потоків: потоки користувача і потоки ядра. Для виконавчої системи Windows кожен потік відображається *керуючим блоком потоку* (executive thread block, ETHREAD). Для ядра системи потік відображається *блоком потоку ядра* (thread

kernel block, KTHREAD).

У режимі користувача доступним є *блок оточення потоку* (thread environment block, TEB), який перебуває в адресному просторі процесу, що створив потік.

Неважко помітити, що кожній структурі даних потоку відповідає структура даних процесу (блоки `EPROCESS, KPROCESS` і `PEB`).

Керуючий блок потоку містить базову інформацію про потік, зокрема:

- блок потоку ядра;
- ідентифікатор процесу, до якого належить потік, і покажчик на керуючий блок цього процесу (EPROCESS);
- стартову адресу потоку, з якої почнеться виконання його коду;
- інформацію для підсистеми безпеки.

Блок потоку ядра, у свою чергу, містить інформацію, необхідну ядру для організації планування і синхронізації потоків, зокрема:

- покажчик на стек ядра;
- інформацію для планувальника;
- інформацію, необхідну для синхронізації цього потоку;
- покажчик на блок оточення потоку.

Блок оточення потоку містить інформацію про потік, доступну для застосувань режиму користувача. До неї належать:

- ідентифікатор потоку;
- покажчик на стек режиму користувача;
- покажчик на блок оточення процесу, до якого належить потік;
- покажчик на локальну пам'ять потоку.

У Win32 API для ***створення потоку*** призначена функція `CreateThread()`, а для його ***завершення*** – `EndThread()`. Назвемо етапи виконання функції `CreateThread()`:

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).

На практиці, однак, 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()`). В інших випадках використовується функція `_beginthreadex()` ***для створення потоку*** й `_endthreadex()` – ***для завершення потоку***, описаних у заголовному файлі `process.h`.

Розглянемо синтаксис функції _beginthreadex(). Відразу ж наголосимо, що той самий набір параметрів (відмінний лише за типами) передають і у функцію CreateThread().

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

де: 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);
```

Розглянемо ***приклад задавання 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);
}
```

Ось приклад виклику _beginthreadex() з усіма параметрами:

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

Щоб змінити характеристики потоку, використовують функцію GetCurrentThread():

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

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);
}
```

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

```
HANDLE th = (HANDLE) _beginthreadex (...);
If (WaitForSingleObject(th, INFINITE) != WAITJAILED) {
// потік завершений успішно
}
CloseHandle(th);
.
```