# ЛЕКЦІЯ 5. БАГАТОЗАДАЧНІСТЬ, ВЗАЄМОДІЯ ПОТОКІВ

**План**

5.1. Багатозадачність та паралелізм
5.2. Принципи та проблеми взаємодії потоків
5.3. Механізми синхронізації: семафори, м'ютекси та умовні змінні
5.4. Взаємодія потоків у Linux
5.5. Взаємодія потоків у Windows XP

## 5.1. Багатозадачність та паралелізм

Як ми вже говорили, у більшості сучасних ОС може бути багато процесів, а в адресному просторі кожного процесу — багато потоків **(1:n)**. Ці ОС підтримують багатопотоковість, а процес у такій системі називають *багатопотоковим* процесом.

Як у багатопроцесорній ОС в адресному просторі процесу виконується один потік **(1:1)**, то говорять про **багатозадачність**.

**Багатопотоковість**

Використання декількох потоків в одному процесі (задачі) означає внесення в нього паралелізму (concurrency).

*Паралелізм* — це одночасне (з погляду прикладного програміста) виконання дій різними фрагментами коду застосування.

Така одночасність може бути реалізована на одному процесорі шляхом перемикання задач (випадок: псевдопаралелізму), а може ґрунтуватися на паралельному виконанні коду на декількох процесорах (випадок справжнього паралелізму).

У багатопроцесорних системах код окремих потоків може виконуватися на окремих процесорах.

Можна виділити такі основні види паралелізму:

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

## 5.2. Принципи та проблеми взаємодії потоків

Розглянемо спочатку принципи взаємодії потоків в одному процесі та проблеми, пов'язані з організацією такої візаємодії.

Потоки, які виконуються в рамках процесу паралельно, можуть бути *незалежними* або *взаємодіючі* між собою.

Потік є незалежним, якщо він *не впливає* на виконання інших потоків процесу, не зазнає впливу з їхнього боку, та не має з ними жодних спільних даних. Його виконання однозначно залежить від вхідних даних і називається *детермінованим*.

Усі інші потоки є такими, що взаємодіють. Ці потоки мають дані, спільні з іншими потоками (вони перебувають в адресному просторі їхнього процесу), їх виконання залежить не тільки від вхідних даних, але й від виконання інших потоків, тобто вони є *недетермінованими*.

Дані, які є загальними для кількох потоків, називають спільно використовуваними даними (shared data). Це — найважливіша концепція багатопотокового програмування. Усякий потік може в будь-який момент часу змінити такі дані. Механізми забезпечення коректного доступу до спільно використовуваних даних називають *механізмами синхронізації потоків*.

Працювати із незалежними потоками простіше ніж із тими, що взаємодіють. Програміст може не враховувати того, що одночасно з таким потоком виконуються інші, а також не звертати уваги на стан спільно використовуваних даних, з якими працює потік.

Проте обійтися без реалізації взаємодії потоків неможливо з кількох причин.

♦ Необхідно організовувати спільне використання інформації під час роботи з потоками. Наприклад, користувачі бази даних або веб-сервера можуть захотіти одночасно виконати запити на отримання однієї й тієї самої інформації, і система має забезпечити її паралельне отримання потоками, що обслуговують цих користувачів.

♦ Коректна реалізація такої взаємодії та використання відповідних алгоритмів можуть значно прискорити обчислювальний процес на багатопроцесорних системах. При цьому задачі розділяють на підзадачі, які виконують паралельно на різних процесорах, а потім їхні результати збирають разом для отримання остаточного розв'язання. Таку технологію називають *технологією паралельних обчислень.*

♦ У задачах, що вимагають паралельного виконання обчислень та операцій введення-виведення, потоки, що виконують введення-виведення, повинні мати можливість подавати сигнали іншим потокам із завершенням своїх операцій.

Необхідність організації паралельного виконання потоків, що взаємодіють, потребує наявності механізмів обміну даними між ними і забезпечення їхньої *синхронізації*.

**5.3. Базові механізми синхронізації потоків**

*Механізмами синхронізації* є засоби операційної системи, які допомагають розв'язувати основне завдання синхронізації — забезпечувати координацію потоків, які працюють зі спільно використовуваними даними. Якщо такі засоби — це мінімальні блоки для побудови багатопотокових програм, їх називають *синхронізаційними примітивами.*

Синхронізаційні механізми поділяють на такі основні категорії:

♦ універсальні, низького рівня, які можна використовувати різними способами *(семафори);*

♦ прості, низького рівня, кожен з яких пристосований до розв'язання тільки однієї задачі *(м'ютекси* та *умовні змінні);*

♦ універсальні високого рівня, виражені через прості; до цієї групи належить концепція *монітора,* яка може бути виражена через м'ютекси та умовні змінні;

♦ високого рівня, пристосовані до розв'язання конкретної синхронізаційної задачі (блокування читання-записування і бар'єри).

**5.3.1. Семафори**

Семафори є найстарішими синхронізаційними примітивами з числа тих, які застосовуються на практиці.

*Семафор* — це спільно використовуваний невід'ємний цілочисловий лічильник, для якого задано початкове значення і визначено такі атомарні операції.

♦ *Зменшення семафора* (down): якщо значення семафора більше від нуля, його зменшують на одиницю, якщо ж значення дорівнює нулю, цей потік переходить у стан очікування доти, поки воно не стане більше від нуля (кажуть, що потік «очікує на семафорі» або «заблокований на семафорі»). Цю операцію називають також *очікуванням* — wait.

♦ *Збільшення семафора* (up): значення семафора збільшують на одиницю; коли при цьому є потоки, які очікують на семафорі, один із них виходить із очікування і виконує свою операцію down. Якщо на семафорі очікують кілька потоків, то внаслідок виконання операції up його значення залишається нульовим, але один із потоків продовжує виконання (у більшості реалізацій вибір цього потоку буде випадковим). Цю операцію також називають *сигналізацією* — post.

Фактично значення семафора визначає кількість потоків, що може пройти через цей семафор без блокування. Коли для семафора задане нульове початкове значення, то він блокуватиме всі потоки доти, поки якийсь потік його не «відкриє», виконавши операцію up. Операції up і down можуть бути виконані будь-якими потоками, що мають доступ до семафора.

**5.2.2. М'ютекси**

*М'ютексом* називають синхронізаційний примітив, що не допускає виконання деякого фрагмента коду більш як одним потоком. Фактично м'ютекс є реалізацією блокування на рівні ОС.

М'ютекс реалізує взаємне виключення. Його основне завдання — блокувати всі потоки, які намагаються отримати доступ до коду, коли цей код уже виконує деякий потік.

М'ютекс може перебувати у двох станах: вільному і зайнятому. Початковим станом є «вільний». Над м'ютексом можливі дві атомарні операції.

*1. Зайняти м'ютекс* (mutex_lock): якщо м'ютекс був вільний, він стає зайнятим, і потік продовжує своє виконання (входячи у критичну секцію); якщо м'ютекс був зайнятий, потік переходить у стан очікування (кажуть, що потік «очікує на м'ютексі», або «заблокований на м'ютексі»), виконання продовжує інший потік.

*2. Звільнити м'ютекс*: м'ютекс стає вільним; якщо на ньому очікують кілька потоків, з них вибирають один, він починає виконуватися, займає м'ютекс і входить у критичну секцію. У більшості реалізацій вибір потоку буде випадковим. Звільнити м'ютекс може тільки його власник.

**5.2.3. Умовні змінні та концепція монітора**

**Поняття умовної змінної**

*Умовною змінною* називають синхронізаційний примітив, який дає змогу організувати очікування виконання умови всередині критичної секції, заданої м'ютексом. Умовна змінна завжди пов'язана із конкретним м'ютексом і даними, захищеними цим м'ютексом. Для умовної змінної визначено такі операції.

♦ *Очікування* (wait). Додатковим вхідним параметром ця операція приймає м'ютекс, який повинен перебувати в закритому стані. Виклик wait відбувається в ситуації, коли не виконується деяка умова, потрібна потоку для продовження роботи. Внаслідок виконання wait потік припиняється, а м'ютекс відкривається. Так інші потоки отримують можливість увійти в критичну секцію і змінити там дані, які вона захищає, можливо, виконавши умову, потрібну потоку. На цьому операція wait не завершується — її завершить інший потік, викликавши операцію signal після того, як умову буде виконано.

♦ *Сигналізація* (signal). Цю операцію потік (назвемо його $T_s$) має виконати після того, як увійде у критичну секцію і завершить роботу з даними (виконавши умову, яку очікував потік, що викликав операцію wait). Ця операція перевіряє, чи немає потоків, які очікують на умовній змінній, і якщо такі потоки є, переводить один із них ($T_w$) у стан готовності (цей потік буде поновлено, коли відповідний потік $Tg$ вийде із критичної секції). Внаслідок поновлення потік $T_w$ завершує виконання операції wait — блокує м'ютекс знову (поновлення і блокування теж відбуваються атомарно). Якщо немає жодного потоку, який очікує на умовній змінній, операція signal не робить нічого, і інформацію про її виконання в системі не зберігають.

♦ *Широкомовна сигналізація* (broadcast) відрізняється від звичайної тим, що переведення у стан готовності і, зрештою, поновлення виконують для всіх потоків, які очікують на цій умовній змінній, а не тільки для одного з них.

Отже, виконання операції" wait складається з таких етапів: відкриття м'ютекса, очікування (поки інший потік не виконає операцію signal або broadcast), закриття м'ютекса.

По суті, це перша *неатомарна* операція, визначена для синхронізаційного примітива, але така відсутність атомарності цілком контрольована (завжди відомо, де потік $T_w$ перейшов у стан очікування і що його з цього стану виведе).

Ідея монітора була вперше запропонована в 1974 році відомим ученим у галузі комп'ютерних наук Ч. А. Хоаром. Монітор часто розуміють як високорівневу конструкцію мови програмування (як приклад такої мови звичайно наводять Java), а саме як набір функцій або методів класу, всередині яких автоматично зберігається неявний загальний м'ютекс разом із операціями очікування і сигналізації. Насправді, як ми бачимо, концепція монітора може ґрунтуватися на базових примітивах — м'ютексах і умовних змінних — і не повинна бути обмежена якоюсь однією мовою.

Монітори Хоара відрізняються від тих, що були розглянуті тут (ці монітори ще називають *MESA-моніторами* за назвою мови, у якій вони вперше з'явилися). Головна відмінність полягає у реалізації сигналізації.

У моніторах Хоара після сигналізації потік $T_s$ негайно припиняють, і керування переходить до потоку $T_w$, який при цьому захоплює блокування. Коли потік $T_w$ вийде із критичної секції або знову виконає операцію очікування, потік $T_s$ буде поновлено.

У MESA-моніторах, як було видно, після сигналізації потік $T_s$ продовжує своє виконання, а потік $T_w$ просто переходить у стан готовності до виконання. Він зможе продовжити своє виконання, коли потік $T_s$ вийде з монітора (чекати цього доведеться недовго, тому що звичайно сигналізація відбувається наприкінці функції монітора).

Результатом є те, що для моніторів Хоара не обов'язково перевіряти умову очікування в циклі, досить умовного оператора (потік негайно отримує керування після виходу з очікування і не може статися так, що за цей час інший потік увійде в монітор і змінить умову). З іншого боку, ці монітори менш ефективні (потрібно витрачати час на те, щоб припиняти і поновлювати потік $T_s$); потрібно мати повну гарантію того, що між виконанням сигналізації та переданням керування потоку $T_w$ планувальник не передасть керування іншому потокові $T_x$, який увійде у функцію монітора. Забезпечення такої гарантії потребує втручання в алгоритм роботи планувальника ОС.

Ці недоліки призводять до того, що на практиці використовують переважно MESA-монітори.

### 5.2.4. Блокування читання-записування

М'ютекси є засобом, який захищає спільно використовувані дані від будь-якого одночасного доступу з боку кількох потоків — будь то читання чи зміна. Насправді нам не завжди потрібен такий однозначний захист, наприклад, для певного типу задач хотілося б розрізняти читання спільно використовуваних даних та їхню модифікацію (для того, щоб, скажімо, дозволяти читання кільком потокам одночасно, а модифікацію — тільки одному). Для розв'язання такої задачі використовують *блокування читання-записування* (read-write locks).

Блокування читання-записування — це синхронізаційний примітив, для якого визначені два режими використання: відкриття для читання і відкриття для записування. При цьому повинні виконуватися такі умови:

- будь-яка кількість потоків може відкривати таке блокування для читання, коли немає жодного потоку, що відкрив його для записування;
- блокування може відкриватися для записування тільки за відсутності потоку, що відкрив його для читання або для записування Простіше кажучи, читати дані може будь-яка кількість потоків одночасно за умови, що ніхто ці дані не змінює; змінювати дані можна тільки тоді, коли їх ніхто не читає і не змінює.

Такі блокування корисні для даних, які зчитуються частіше, ніж модифікуються (наприклад, більшість СУБД реалізує блокування такого роду для забезпечення доступу до бази даних).

**Типи блокувань**

Розрізняють два типи блокувань читання-записування: з *кращим читанням* і *з кращим записом.* Відмінність між ними виявляється тоді, коли потік намагається відкрити таке блокування для читання за умови, що він робить це не першим і що є призупинені потоки, які очікують можливості відкрити це блокування для записування.

У разі кращого читання потік негайно відкриває блокування для читання і продовжує свою роботу незалежно від того, є потоки-записувачі, що очікують, чи ні. Потоки-записувачі продовжують своє очікування.

У разі кращого записування за наявності потоків-записувачів, що очікують, потік-читач припиняється й не буде поновлений доти, поки всі записувачі не виконають свої дії і не закриють блокування.

Зазначимо, що для обох типів потік-записувач не може відкрити блокування, поки його тримає відкритим хоча б один читач, — перевага надається тільки новим потокам-читачам, які намагаються відкривати додаткові блокування.

**Операції блокувань**

Розглянемо операції, допустимі для блокувань читання-записування.

♦ *Відкриття для читання* (rwl ock_rd1 ock). Якщо є потік, який відкрив блокування для записування, поточний потік припиняють. Якщо такий потік відсутній:

- у разі кращого читання блокування відкривають для читання і потік продовжує своє виконання;
- у разі кращого записування перевіряють, чи немає призупинених потоків, які очікують відкриття цього блокування для записування; якщо вони є - потік припиняють, якщо немає — блокування відкривають для читання і потік продовжує своє виконання. При цьому необхідно, щоб кілька потоків могли відкрити блокування для читання, тому в разі, коли блокування вже відкрите для читання, для його нового відкриття збільшують внутрішній лічильник потоків-читачів.

♦ *Відкриття для записування* (rwlock_wrlock). Якшо є потік, який відкрив блокування для читання або записування, поточний потік припиняють; коли жодного такого потоку немає, блокування відкривають для записування і потік продовжує своє виконання.

♦ *Закриття* (rwl ockunl оск). У разі наявності кількох потоків, які відкрили блокування для читання, воно залишається відкритим для читання, і внутрішній лічильник потоків-читачів зменшують на одиницю. Якщо блокування відкрите для читання тільки одним потоком (лічильник дорівнює одиниці) його знімають, якщо є потоки-записувачі, які очікують на цьому блокуванні, один із них поновлюють. Коли блокування відкрите для записування, його знімають, при цьому за наявності потоків-читачів, що очікують, всі вони поновлюються, а з потоків-записувачів, що очікують, поновлюють тільки один. Якщо очікують і читачі й записувачі, результат залежить від типу блокування (у разі кращого читання або якщо жодного записувача немає, поновлюють усіх читачів, а якщо читачі відсутні у разі кращого записування — поновлюють одного з записувачів.

Блокування читання-записування, як і м'ютекси, мають власника, тому не можна закрити блокування в потоці, який його не відкривав.

**5.4. Взаємодія потоків у Linux**

Бібліотеки підтримки потоків Linux (LinuxThreads і NPTL) надають програмам користувача набір синхронізаційних примітивів, визначених стандартом POSIX (м'ютекси, умовні змінні, семафори, блокування читання-записування, бар'єри). Крім того, у NPTL допускають використання цих примітивів у поєднанні з відображуваною або розподілюваною пам'яттю для реалізації міжпроцесової синхронізації.

Описані примітиви реалізовані на основі базових механізмів синхронізації, доступних через системні виклики (у ядрі версії 2.6 до них належать ф'ютекси). Крім того, код ядра може використовувати спеціальні механізми синхронізації, доступні тільки у ядрі.

Далі розглянемо базові механізми синхронізації ядра і застосувань користувача.

### 5.4.1. Механізми синхронізації ядра Linux

Ядро Linux є *реентерабельним.* Це означає, що одночасно в режимі ядра може виконуватися код кількох процесів. В однопроцесорних системах процесор у конкретний момент виконує код тільки одного процесу, інші перебувають у стані очікування. У багатопроцесорних системах код різних процесів може виконуватися паралельно. Для досягнення реентерабельності всередині ядра повинна бути реалізована така сама синхронізація, що і між потоками одного процесу. Для цього у ядрі передбачені механізми взаємного виключення, які забезпечують безпечний доступ до спільно використовуваних даних ядра.

Аналогом потоків у ядрі виступають *шляхи передачі керування ядра* (kernel control paths). Таким шляхом є послідовність інструкцій, виконуваних ядром для реалізації реакції на системний виклик або обробку переривання. Така послідовність звичайно зводиться до виконання кількох функцій ядра. Наприклад, для обробки системного виклику шлях передачі керування починають з виклику функції system_cal 1 () і завершують викликом ret_systefli_call (). Надалі іноді говоритимемо не про шляхи керування, а про *процеси* в *режимі ядра,* маючи на увазі шляхи передачі керування, що відповідають системним викликам, виконаним процесами.

**Витісняльність ядра**

До останнього часу ядро Linux належало до категорії *невитісняльних* (nonpre-emptive). Це означало, що процес, виконуваний в режимі ядра, не міг бути призупинений (витіснений іншим процесом), поки він сам не вирішить віддати керування. У разі переривання керування після виклику обробника мало бути повернуте в той самий процес.

У ядрі версії 2.6 ситуація змінилася. Це перше ядро, що є витісняльним (preemptive). Тепер процес, виконуваний в режимі ядра, може бути призупинений, коли минув квант часу або почав виконуватися процес із вищим пріоритетом. Після обробки переривання керування теж може бути передане іншому процесові. У результаті скоротився час відгуку системи. Тепер процеси, які проводять занадто багато часу в режимі ядра, не затримуватимуть виконання інших процесів. Природно, що у ядрі все одно залишаються місця, де його не можна витісняти, але тепер їх потрібно виділяти явно.

**Необхідність синхронізації у ядрі**

Спільно використовувані дані у ядрі можуть бути змінені:

•♦ у коді, викликаному асинхронно внаслідок переривання (до такого коду належать і сам обробник, і *код відкладеної реакції на переривання* (softirq), який пов'язують із обробником, але виконують трохи пізніше);

♦ у коді, виконуваному на іншому процесорі;

♦ у коді, що витиснув розглядуваний (у разі витісняльного ядра).

Для забезпечення коректної роботи потрібно завжди забезпечувати синхронізацію доступу до цих структур. Розглянемо механізми такої синхронізації.

**Заборона переривань**

Для однопроцесорних систем у ядрі можлива проста синхронізація через заборону переривань на час виконання критичних секцій. Даний підхід може бути ефективним тільки тоді, коли критична секція невелика.

**Атомарні операції**

Як елементарну альтернативу блокуванням ядро Linux пропонує набір *атомарних операцій.* До них належить набір найпростіших операцій (збільшення і зменшення на одиницю, побітові операції), що їх атомарне виконання гарантує ядро. Зазначимо, що саме на базі цих операцій реалізовані складніші примітиви синхронізації, такі як семафори ядра або блокування читання-записування, а також ф'ютекси, на яких ґрунтується реалізація синхронізаційних примітивів режиму користувача.

**Спін-блокування**

*Спін-блокування* (spinlocks) — це найпростіші можливі блокування ядра. Вони працюють аналогічно до традиційних м'ютексів, за винятком того, що коли процес у режимі ядра запросить спін-блокування, зайняте у цей час іншим процесом, він не призупиниться, а виконуватиме цикл активного очікування доти, поки блокування не звільниться. У результаті не затрачаються ресурси на призупинення процесу. З іншого боку, такий процес витрачатиме процесорний час, тому спін-блокування краще зберігати недовго. Спін-блокування не є рекурсивними, повторна спроба запровадити таке блокування призводить до взаємного блокування.

Використання спін-блокувань дає змогу вирішити базові проблеми організації взаємного виключення. Для складніших задач можна застосовувати *семафори ядра.*

**Семафори ядра**

Семафори ядра, на відміну від спін-блокувань, змушують процеси не переходити до активного очікування, а призупинятися, тому їх звичайно використовують тоді, коли очікування може тривати довго (якщо це не так, спін-блокування ефективніші). Семафор ядра за принципом дії не відрізняється від традиційного семафора, він реалізований як структура, що містить цілочисловий лічильник і покажчик на чергу очікування. Структури даних призупинених процесів додають у цю чергу.

**Блокування читання-записування**

Ядро Linux пропонує також блокування читання-записування, подібні до описаних у розділі 5.3.4. Основна їхня відмінність від традиційної реалізації — режими доступу для читання і для записування повністю розділені (відкриттю для читання відповідає закриття для читання, а відкриттю для записування - закриття для записування). Є також варіант семафора, що розрізняє доступ для читання і для записування.

### 5.4.2. Синхронізація процесів користувача у Linux. Ф'ютекси

Для того щоб можна було використовувати у застосуваннях різні примітиви синхронізації (м'ютекси, семафори, умовні змінні), на рівні ОС необхідно розробити деяку структуру даних для сигналізації (наприклад, яка відображає лічильник монітора) і забезпечити її спільне використання кількома потоками. Також треба забезпечити можливість переведення потоків у стан очікування, і поновлення одного або одночасно кількох потоків (негайно або через деякий проміжок часу). Були запропоновані різні способи розв'язання цієї проблеми.

♦ Підтримка спеціальних засобів синхронізації - *семафорів System V,* які спочатку розробляли для систем із реалізацією моделі процесів. Уся робота з ними заснована на спеціальних структурах даних ядра. Доступ до них здійснюють за допомогою системних викликів. Такий підхід є неефективним, оскільки кожний системний виклик спричиняє виконання кількох сотень машинних інструкцій і перемикань між режимами процесора.

♦ Використання для операції очікування аналогів системного виклику sleep() і реалізація операції поновлення потоків на основі сигналів. Цей підхід було прийнято у бібліотеці LinuxThreads. Застосування сигналів знову передбачало використання системних викликів, а виконання sleep() призводило до втрат часу на перемикання контексту.

На практиці найкращим є вирішення, яке використовує системні виклики тільки в разі гострої необхідності й обходиться без перемикання контексту. Вирішення, що відповідає цим вимогам, реалізує швидке блокування режиму користувача (fast user-level locking).

Розглянемо реалізацію такого блокування для двох випадків.

♦ Потік успішно займає вільне блокування. За звичайного навантаження така ситуація трапляється найчастіше, тому саме для цього випадку потрібно домагатися максимальної ефективності, прагнучи до того, щоб обійтися без системних викликів.

♦ Є потік, що вже зайняв це блокування. У цьому разі можна виконати системний виклик, щоб призупинити поточний потік. Така ситуація виникає рідше, і втрати продуктивності будуть менші.

Щоб потоки не зверталися до ядра в разі успішного зайняття блокування, вони мають спільно використовувати дані в пам'яті, доступній у режимі користувача. Для потоків одного процесу забезпечити спільне використання нескладно, оскільки в них загальний адресний простір. Для потоків різних процесів потрібно організовувати розподілювану або відображувану пам'ять, докладніше про це буде сказано в розділі 6. Ці спільно використовувані дані називають *блокуванням користувача* (user lock). Вони визначають, закрите чи відкрите блокування і чи є потоки, що очікують на ньому. Потоки атомарно змінюють ці дані у разі спроби заблокування, якщо при цьому виникає необхідність заблокувати або поновити потік, виконують системний виклик, коли такої необхідності немає — потік продовжує виконання в режимі користувача.

Реалізація такого блокування була інтегрована у ядро Linux версії 2.6. Вона дістала назву *ф'ютекс* (futex, від fast user-level mutex — швидкий м'ютекс користувача) [73].

Ф'ютекс — цілочисловий лічильник, що перебуває у спільній пам'яті, яку використовують потоки або процеси. Роботу з цим лічильником ведуть аналогічно до роботи із семафором. Для його збільшення або зменшення потоки мають виконувати атомарні інструкції процесора.

Зазначимо, що лічильник ф'ютекса може розміщатися у спільній пам'яті де завгодно. Потоки можуть перетворити будь-яке місце пам'яті у ф'ютекс без попередньої підготовки (поки не виникне

суперництво за ф'ютекс, потоки лише збільшують і зменшують цілочислове значення у спільній пам'яті). Ф'ютексів можна створювати безліч.

Розглянемо ситуації, коли використання ф'ютекса потребує виконання системного виклику.

У разі спроби заблокуватися на ф'ютексі (коли потрібно його зменшити, а він дорівнює нулю) виконують системний виклик для організації очікування відповідно до операції futex_wait (серед інших параметрів йому потрібно передати адресу пам'яті, де перебуває лічильник, і, в окремих випадках, максимальний час очікування). У коді цього виклику адресу пам'яті додають у ядрі до хеш-таблиці й створюють чергу очікування, куди поміщають відповідний керуючий блок потоку. Значення ф'ютекса покладають рівним -1.

Коли ми збільшуємо ф'ютекс, може з'ясуватися, що вихідним значенням буде -1. У цьому разі виконують системний виклик для поновлення потоків відповідно до операції futex_wake (серед параметрів йому, крім адреси пам'яті, треба передати кількість потоків, які потрібно поновити). У результаті потоки переводять із черги очікування в чергу готових процесів.

Легко побачити, що ф'ютекси автоматично роблять можливою синхронізацію потоків різних процесів через розподілювану пам'ять.

Інтерфейс ф'ютексів не призначений для безпосереднього використання у прикладних програмах. Замість цього на основі ф'ютексів бібліотеки підтримки потоків можуть бути реалізовані примітиви синхронізації більш високого рівня (саме це й зроблено у NPTL).

## 5.5. Взаємодія потоків у Windows XP

### 5.5.1. Механізми синхронізації потоків ОС

Механізми синхронізації потоків у Windows XP реалізовані на трьох рівнях: синхронізація в ядрі (на рівні потоків ядра), виконавчій системі та в режимі користувача у підсистемі Win32.

**Синхронізація в ядрі**

Основними механізмами синхронізації ядра Windows XP є спін-блокування. Перед тим як почати роботу зі спільно використовуваними даними, потік ядра має отримати таке блокування, можливо, шляхом активного очікування. Зазначимо, що для однопроцесорних систем у разі одержання спін-блокування замість активного очікування тимчасово маскують переривання від тих джерел, котрі можуть потенційно мати доступ до спільно використовуваних даних.

**Синхронізація у виконавчій системі**

Код виконавчої системи теж може користуватися спін-блокуваннями, але вони пов'язані з активним очікуванням і їхнє застосування обмежується лише організацією взаємного виключення впродовж короткого часу. Складніші задачі синхронізації розв'язують за допомогою *диспетчерських об'єктів* (dispatcher objects) і *ресурсів виконавчої системи* (executive resources).

Диспетчерські об'єкти — це об'єкти виконавчої системи, що можуть бути використані разом із сервісами очікування менеджера об'єктів.

Ресурси виконавчої системи можуть бути використані тільки в коді ВС, коду користувача вони недоступні. Вони не є об'єктами виконавчої системи, а реалізовані як структури даних зі спеціальними операціями, які можна виконувати над ними. Такий ресурс допускає різні режими доступу — взаємне виключення (як м'ютекс) і доступ за принципом блокування читання-записування.

**Об'єкти синхронізації режиму користувача**

На основі диспетчерських об'єктів підсистема Win32 реалізує набір об'єктів синхронізації режиму користувача. До них належать, зокрема, м'ютекси, семафори і події.

**Очікування на диспетчерських об'єктах**

Потік може синхронізуватися з диспетчерським об'єктом через виконання очікування на його дескрипторі. Для цього менеджер об'єктів надає спеціальні сервіси очікування: очікування одного об'єкта і очікування кількох об'єктів (на базі цих сервісів у Win32 API реалізовані функції WaitForSingleObjectO і WaitForMultipie-Objects O).

Звертаючись до такого сервісу, потрібно задати дескриптор (або масив дескрипторів) об'єкта, на якому треба очікувати. Після звертання до сервісу очікування потік змінює диспетчерський стан на очікування, після чого ядро вилучає його керуючий блок із черги готових потоків і додає цей блок до черги очікування, пов'язаної з диспетчерським об'єктом.

Диспетчерський об'єкт може перебувати в одному з двох станів: *сигналізованому* (signaled) і *несигналізованому* (nonsignaled). Потік поновлює своє виконання, коли об'єкт, на якому він очікує, переходить у сигналізований стан (відбувається його *сигналізація).* При цьому виконавча підсистема перевіряє чергу очікування цього об'єкта і поновлює виконання одного або кількох потоків з неї (виконуючи перепланування і, можливо, тимчасово підвингуючи їхній пріоритет).

Способи сигналізації та реакція на неї розрізняються для різних об'єктів. Сигналізація зазвичай відбувається явно (коли інший потік викликає спеціальний *метод сигналізації),* однак для деяких об'єктів (процесів або потоків) сигналізація може бути і неявною. Наприклад, об'єкт-потік перебуває в несигналізованому стані впродовж усього свого життєвого циклу, а в разі завершення переходить у сигналізований стан внаслідок чого можуть бути поновлені потоки, які очікують (або приєднали) його. З іншого боку, об'єкт-процес переходить у сигналізований стан, коли завершується останній його потік.

**Структури даних синхронізації**

Для відстеження того, який потік очікує на який об'єкт, використовують дві структури даних: *диспетчерський заголовок об'єкта* і *блок очікування потоку.*

*Диспетчерський заголовок* пов'язаний із диспетчерським об'єктом. Він містить інформацію про тип об'єкта; сигнальний стан; покажчик на список блоків очікування потоків, які очікують на цьому об'єкті.

*Блок очікування* відображає той потік, що очікує на диспетчерському об'єкті. Він містить покажчик на цей диспетчерський об'єкт; покажчик на блок очікувального потоку; покажчик на наступний блок очікування того самого потоку (якщо потік очікує кілька об'єктів); покажчик на наступний блок очікування, пов'язаний з тим самим об'єктом (якщо цей об'єкт очікують кілька потоків); тип очікування (одного або кількох об'єктів); місце дескриптора цього об'єкта в масиві дескрипторів, переданому сервісу очікування у разі очікування кількох об'єктів.

Отже, виконавча система і ядро у будь-який момент можуть отримати інформацію про всі потоки, що очікують на диспетчерському об'єкті, а також інформацію про всі диспетчерські об'єкти, яких очікує потік.

**Висновки**

1. Потоки одного процесу, що взаємодіють, мають доступ до спільно використовуваних даних, розміщених в адресному просторі цього процесу. Будь-який потік здатний у будь-який момент часу змінити ці дані й спричинити стан змагання, коли результат залежить від послідовності виконання потоків.

2. Для забезпечення коректного доступу до спільно використовуваних даних застосовують механізми синхронізації потоків; основним з них є забезпечення взаємного виключення, коли в конкретний момент часу доступ до таких даних може мати тільки один потік. Для організації взаємного виключення використовують *блокування*.

3. Для розв'язання задач синхронізації можна використовувати різні синхронізаційні примітиви. До найпростіших примітивів належать м'ютекси, семафори, умовні змінні, блокування читання-записування і бар'єри.

4. Механізмом більш високого рівня є концепція монітора, що поєднує м'ютекси та умовні змінні, а також задає деякі правила їхньої взаємодії для захисту спільно використовуваних даних. Використання моніторів є найпослідовнішим підходом до синхронізації потоків.

**Контрольні запитання і завдання**

1. Яка різниця між взаємодією потоків і процесів в ОС?
2. Основні переваги і недоліки багатопотоковості
3. Який потік є детермінованим?
4. Коли виникає необхідність організації взаємодії потоків одного процесу?
5. Основні механізми синхронізації потоків.
6. Назвіть операції семафору.
7. М'ютекси. Їх використання для синхронізації потоків. Операції.
8. Умовні змінні та їхні операції.
9. Блокування читання-записування. Типи блокувань. Операції.
10. Взаємодія потоків в Linux. Ф'ютекси.
11. Механізми синхронізації потоків у Windows XP

**Література до лекції 5.**

1. Шеховцов В. А. Операційні системи. Підручник для ВНЗ. – К.: ВНV, 2008. –576 с.
2. Таненбаум Э. Операционные системы– СПб: Питер. – 2002 г. – 1040 с.