4. ОС багатопроцесорних систем.

Серед різних класів обчислювальних систем останнім часом набула розповсюдження архітектура з багатьма потоками команд і багатьма потоками даних (multiple instruction multiple data – ***MIMD***). В таких системах кілька (чи багато) процесорів одночасно виконують різні послідовності команд з різними наборами даних. Всі процесори є універсальними в тому розумінні, що вони можуть виконувати всі команди, необхідні для обробки даних. В залежності від того, як процесори обмінюються даними, такі системи відносять до ***сильнозв'язаних*** або до ***вільно зв'язаних*** систем.

Вільно зв'язані системи мають ***розподілену пам'ять*** — кожен процесор працює із своєю пам'яттю, а обмін даними між окремими комп'ютерами відбувається або через спеціальні канали, або через мережеві пристрої. Такі обчислювальні системи називають ***кластерами*** або ***мультикомп'ютерами***. Кластерні системи найбільше пристосовані до нарощування їх потужності (властивість ***масштабування***), але для їх ефективного використання в ОС треба вносити певні доповнення. На сьогодні ці додаткові функції, як правило, реалізує ***проміжне програмне забезпечення***, що працює на кожному ***вузлі*** кластера разом з його ОС.

В функції проміжного програмного забезпечення входить:

- забезпечення єдиної точки входу в кластер — вся система виглядає для користувача як один комп'ютер;
- підтримка єдиної ієрархії файлів — всі файли і каталоги на різних вузлах виглядять змонтованими в одному кореневому каталозі;
- забезпечення єдиної точки управління — весь кластер можна контролювати із одного вузла;
- емуляція розподіленої спільно використовуваної пам'яті для роботи різних програм з одними і тими ж даними;
- єдина система управління завданнями — користувач не повинен явно вказувати вузли, на яких буде виконуватись його задача;
- єдиний інтерфейс користувача не залежний від типу робочої станції;
- єдиний простір введення-виведення — будь-який вузол може отримати доступ до будь-якого периферійного пристрою кластера;
- єдиний простір процесів — процес на одному вузлі може створювати процеси на інших вузлах та взаємодіяти з ними;
- підтримка контрольних точок для періодичного збереження стану процесів та результатів обчислень і відновлення роботи при збоях;
- забезпечення можливості ***міграції*** процесів для ***балансування навантаження*** на окремі вузли.

На відміну від кластерів, сильнозв'язані системи мають спільну пам'ять, доступ до якої розділяють між собою кілька процесорів. Розрізняють два варіанти таких систем.

В першому реалізується архітектура з ведучим і веденими процесорами (***master / slave architecture***) де ОС виконується тільки на одному виділеному процесорі, а решта виконують прикладні програми та, можливо, утиліти ОС. Пристосувати для роботи в такій системі звичайну однопроцесорну ОС відносно нескладно, але залишаються деякі проблеми. Так, збій в роботі ведучого процесора приводить до відмови всієї системи, а недостатня продуктивність ведучого може гальмувати всю роботу.

Тому на сьогодні більш розповсюджений варіант побудови ***симетричних багатопроцесорних систем*** (symmetric multiprocessor – ***SMP***). В таких системах ядро ОС може виконуватись на будь-якому процесорі і робота кожного процесора часто планується незалежно. Ядро, як правило, само складається з ряду процесів, які можуть виконуватись паралельно. Це може значно підвищити його продуктивність. Однак при розробці ОС з

підтримкою SMP необхідно вирішити ряд складних питань пов'язаних з плануванням виконання завдань, узгодженим управлінням пам'яттю, синхронізацією процесів, забезпеченням цілісності даних і таке інше. Хоча симетричні багатопроцесорні системи складніші ніж однопроцесорні і не можуть конкурувати з можливостями потужних кластерних систем, на сьогодні це найбільш розповсюджена архітектура робочих станцій і серверів. Тому сучасні ОС загального призначення сімейств Windows NT, UNIX, Linux мають вбудовану підтримку режиму SMP.

5. Ядро з підтримкою багатопотоковості.

Поняття процесу з точки зору ОС має кілька складових. По-перше, це властивість володіння ресурсами. Для розміщення образу процесу йому виділяється пам'ять, процес може отримувати в своє розпорядження канали та пристрої введення-виведення даних, файли та ін.

По-друге, процес час від часу виконує код своїх процедур, має цілком визначений стан і пріоритет, відповідно якого ОС планує його виконання. Часто бажано мати можливість паралельно виконувати деякі процедури в рамках процесу координуючи їх роботу відповідно алгоритму програми.

Саме тому багато сучасних ОС розглядають процес тільки як власника ресурсів, а об'єкт виконання і диспетчеризації – ***потік*** (thread) ідентифікують як окрему сутність в складі процесу. Причому процес може мати більше одного потоку виконання. В такому разі кожен потік має свій ідентифікатор, стекову пам'ять, пріоритет, стан і є незалежним об'єктом диспетчеризації. Такі потоки називаються ***потоками рівня ядра*** (kernel-level threads - ***KLT***) Для реалізації багатопотокової моделі виконання програм насправді необхідно виконати дві умови:

- ядро ОС повинне підтримувати диспетчеризацію потоків та їх взаємну безпеку при виконанні потоками коду ядра;
- код прикладної програми повинен містити команди створення нових потоків для паралельного виконання процедур, забезпечення їх взаємної безпеки при роботі в режимі користувача, завершення окремих потоків.

Суть поняття безпеки потоків розглянемо далі. Відмітимо, що в порівнянні з реалізацією прикладної програми в вигляді кількох взаємодіючих процесів багатопотокова модель має ряд переваг:

- створення (як і завершення) нового потоку в межах процесу проходить набагато швидше, ніж створення (завершення) нового процесу;
- перемикання потоків в межах процесу виконується швидше, ніж перемикання процесів;
- ефективність обміну інформацією між потоками одного процесу підвищується за рахунок можливості прямого доступу до спільних ресурсів без посередництва ОС.

6. Багатопотоковість на рівні користувача.

На щастя розпаралелити програму можна і в тому випадку, якщо ОС не підтримує багатопотоковості. Для цього її треба розробити в системі програмування на зразок Modula-2, яка має відповідний бібліотечний модуль, або скористуватись спеціальною бібліотекою потоків для мови програмування загального призначення. Така бібліотека забезпечує підтримку ***багатопотоковості на рівні користувача*** (user-level threads – ***ULT***).

З точки зору ядра така програма виглядає як процес з одним потоком, але на рівні користувача (в захищеному режимі) програма може виконувати процедури бібліотеки потоків для створення і завершення нових потоків, обміну даними і повідомленнями між ними та планування їх виконання. В цьому ж процесі виконується і процедура “власного”

диспетчера потоків. Така нескладна схема має певні переваги над KLT:

- перемикання ULT потоків не потребує перемикання режимів процесора (користувача — ядра — користувача) що дозволяє знизити накладні витрати;
- з метою підвищення ефективності алгоритм планування виконання потоків можна підбирати в залежності від специфіки конкретної програми;
- багатопотоковість на рівні користувача можна реалізувати для будь-якої ОС аж до однозадачної включно, при цьому в ядро не треба вносити ніяких змін.

Разом з тим, схема ULT порівняно з KLT має і недоліки:

- якщо один з потоків виконує системний виклик і блокується, то блокуються і всі інші потоки процесу оскільки для ядра ОС це єдиний потік;
- потоки рівня користувача можуть виконуватись тільки на одному процесорі, отже така схема паралелізму не використовує переваг багатопроцесорних систем.

Деякі ОС підтримують комбінацію потоків обох типів. Так наприклад, в ОС Solaris кілька потоків рівня користувача відповідають такій же або меншій кількості потоків рівня ядра. Це потужний і гнучкий підхід, але він значно знижує переносимість програм.

7. Особливості паралельних обчислень.

Повернемось до поняття ***потокової безпеки*** (thread-safety). Ця концепція програмування має відношення саме до багатопотокових програм. Код називають потоково-безпечним, якщо він функціонує коректно при виконанні в кількох потоках одночасно. Зокрема, він повинен забезпечувати коректний доступ кількох потоків до спільних даних. Існує кілька потенційних джерел порушень потокової безпеки:

- доступ до глобальних змінних або динамічної пам'яті;
- виділення/вивільнення ресурсів, таких як файли;
- неявний доступ через посилання і покажчики;
- побічний ефект функцій;

Є кілька способів досягнення потокової безпеки коду і один із найважливіших — забезпечення ***реєнтерабельності*** (reenterability). Програма в цілому або її окрема процедура називається реєнтерабельною, якщо вона розроблена таким чином, що одна і та ж копія інструкцій програми в пам'яті може бути спільно використана кількома потоками або процесами. При цьому другий процес може викликати реєнтерабельний код до того, як з ним завершить роботу перший процес і це не повинно привести до помилки. Забезпечення реєнтерабельності є ключовим моментом при програмуванні багатозадачних систем, зокрема, ОС.

Для забезпечення реєнтерабельності необхідне виконання кількох умов:

- ніяка частина коду, що викликається, не повинна модифікуватися;
- процедура, що викликається, не повинна зберігати інформацію між викликами;
- якщо процедура змінює які-небудь дані, то вони повинні бути унікальними для кожного користувача;
- процедура не повинна повертати покажчики на об'єкти, загальні для різних користувачів.

У загальному випадку, для забезпечення реєнтерабельності необхідно, щоб викликаюча процедура кожного разу передавала процедурі, що викликається, всі необхідні для її роботи дані. Практичним засобом вирішення цього завдання є вимога до кожного процесу (потоку) зберігати свою локальну копію змінних.

На жаль, реальні багатопотокові прикладні програми часто не є реєнтерабельними з тієї простої причини, що вони спеціально проектуються так, щоб кілька потоків працювали над

обробкою спільних даних (глобальних ресурсів).

Важливо що для будь-яких двох (або більшого числа) потоків, що працюють з одними і тими ж даними, неможливо наперед вказати їх відносні швидкості виконання. Отже порядок виконання команд є недетермінованим, а значить результат їх роботи залежить від випадкових чинників. Таке явище називають ***гонками*** (race condition). Із-за гонок спільні ресурси завжди знаходяться в небезпеці. Крім того, стає дуже важко виявити програмну помилку, оскільки результат роботи програми перестає бути детермінованим і відтворним.

8. Механізми захисту ресурсів.

Отже не важко прийти до висновку – глобальні змінні, що розділяються (тобто використовуються спільно кількома процесами), як і інші глобальні ресурси, потребують захисту. Єдиний спосіб зробити це – управляти кодом, що здійснює доступ до цих ресурсів.

Для подальшого викладу істотно, що способи взаємодії процесів відносно ресурсів, доступ до яких вони розділяють між собою, можна класифікувати по ступеню обізнаності один про одного [2] як показано в Табл. ?.1.

Таблиця ?.1. Види взаємодії процесів.

Ступінь обізнаності	**Вид взаємодії**	**Вплив одного процесу на на іншій**	**Потенційні проблеми**
Процеси не обізнані про наявність один одного.	Спостерігається конкуренція за володінням ресурсом.	Результат роботи одного процесу не залежить від дій інших. Можливий вплив роботи одного процесу на час роботи іншого.	Необхідність взаємних виключень. Взаємне блокування (для повторно використовуваних ресурсів). Голодування.
Процеси побічно обізнані про наявність один одного.	Співпраця з використанням розділення ресурсів.	Результат роботи одного процесу може залежати від інформації, отриманої від інших процесів (через ресурси, що розділяються). Можливий вплив роботи одного процесу на час роботи іншого.	Необхідність взаємних виключень. Взаємне блокування (для повторно використовуваних ресурсів). Голодування. Зв'язок даних.
Процеси безпосереднь о обізнані про наявність один одного (знають імена).	Співпраця з використанням зв'язку.	Результат роботи одного процесу може залежати від інформації, отриманої від інших процесів. Можливий вплив роботи одного процесу на час роботи іншого.	Взаємне блокування (для витратних ресурсів). Голодування.

Якщо процес не знає про можливість існування інших процесів, він вважає всі наявні ресурси своєю власністю. Такі процеси конкурують за монопольний доступ до ресурсів. Єдиним арбітром в цій конкурентній боротьбі є ОС.

У разі конкуренції процесів ми стикаємося з трьома проблемами:

- Необхідність ***взаємних виключень*** (mutual exclusion) - тобто виключення процесом, який дістав доступ до ресурсу, що розділяється, можливості одночасного доступу до цього ресурсу для решти процесів. При цьому такий ресурс називають ***критичним ресурсом***, а частину програми, яка його використовує, називають ***критичним***

розділом (секцією) (critical section). Украй важливо, щоб в критичному розділі у будь-який момент могла знаходитися тільки одна програма. Здійснення взаємних виключень створює дві додаткові проблеми:

- ***Взаємне блокування*** (deadlock). Наприклад, два процеси P1 і P2 для свого виконання мають потребу в двох одних і тих же ресурсах R1 і R2 одночасно. При цьому кожний з них утримує по одному ресурсу (наприклад так: P1 – R1, P2 – R2) і безуспішно намагається захопити інший.
- ***Голодування*** (starvation). Нехай три процеси P1, P2, P3 періодично потребують одного і того ж ресурсу. Теоретично можлива ситуація при якій доступ до ресурсу отримуватимуть P1 і P2 в порядку черговості, а P3 ніколи не отримає доступу до нього, хоча ніякого взаємного блокування немає.

Треба розуміти, що взаємні виключення в даному випадку є базовим механізмом захисту даних, а взаємні блокування і голодування — його неприємними наслідками.

Якщо процес припускає наявність в системі інших процесів (процеси побічно обізнані про наявність один одного), він може демонструвати тактику співпраці, приймаючи заходи до підтримки цілісності спільно використовуваних ресурсів.

Коли ж процеси безпосередньо обізнані про наявність один одного (це також зазвичай справедливо для потоків одного процесу), вони здатні спілкуватися з використанням "імен" процесів і з самого початку створені для спільної роботи з використанням зв'язку. Зв'язок забезпечує можливість синхронізації або координації дій процесів. Зазвичай можна вважати, що зв'язок складається з посилки повідомлень певного типу. Примітиви для відправлення і отримання повідомлень надаються мовою програмування або ядром ОС. При цьому можна обійтись без взаємних виключень, проте проблеми взаємних блокувань і голодування залишаються. Так, наприклад, два процеси можуть заблокуватся взаємним очікуванням повідомлень один від одного. В ролі ресурсів в даному випадку виступають ті ж повідомлення — це ***витратний ресурс***.

Отже в більшості випадків без взаємних виключень не обійтися. Втім, будь-яка можливість підтримки взаємних виключень повинна відповідати таким вимогам:

- Взаємні виключення повинні здійснюватися в примусовому порядку.
- Процес, що завершує (перериває) роботу поза критичним розділом, не повинен впливати на інші процеси.
- Не повинна виникати ситуація нескінченного очікування входу в критичний розділ (виключення взаємних блокувань і голодування).
- Коли в критичному розділі немає жодного процесу, будь-який процес, що запросив можливість входу в нього, повинен негайно її отримати.
- Не робиться ніяких припущень щодо кількості процесів або їх відносних швидкостей виконання.
- Процес залишається в критичному розділі тільки протягом обмеженого часу.

Розроблені і використовуються ряд підходів до реалізації цих вимог:

- Передача відповідальності за відповідність вимог самому процесу (програмний підхід).
- Використання машинних команд спеціального призначення.
- Надання певного рівня підтримки з боку мови програмування або ОС.