Харківський національний університет імені В.Н. Каразіна
Факультет комп'ютерних наук
Кафедра штучного інтелекту та програмного забезпечення

А. Горбань

Операційні системи.
Вступ до курсу.
Частина 1.

Навчальний посібник
Попередня редакція від 18.12.18.

Зміст

1. Вступ

Курс «Операційні системи» знайомить студентів з принципами проектування і реалізації операційних систем (далі ОС) і основами системного програмування в середовищі ОС сімейств UNIX, Linux, Windows.

Обов'язкові теми курсу включають огляд компонентів операційної системи, питання взаємних виключень і синхронізації, реалізацію процесів і потоків, алгоритми планування і диспетчеризації, управління пам'яттю і файлові системи.

В цьому посібнику в максимально стислій формі приведений теоретичний матеріал що зазвичай вивчається в першому семестрі двосеместрового курсу. На жаль, із-за обмеженого об'єму зовсім не розглянуті питання планування процесів, управління пам'яттю, введенням-виведенням даних, файловою системою та деякі інші важливі теми.

При виборі форми викладу головними завданнями були ознайомлення студентів з спеціальною термінологією та базовими поняттями для більш ефективної роботи з численними друкованими, електронними джерелами та технічною документацією. В якості рекомендованої літератури свідомо вказані лише два класичних підручники, які проте відносно легкодоступні.

Ряду ілюстративних прикладів фрагментів програм передує коментарій такого виду:

```
// Далі текст в стилі мови C а не на C.
```

Читач знайомий з мовою програмування C (C++, C#, Java) зможе без проблем прочитати і зрозуміти текст маючи на увазі: в таких лістингах вважається що всі параметри передаються по посиланню.

Ваші коментарі, зауваження і пропозиції будуть прийняті з вдячністю на a.gorban@karazin.ua.

2. Процес та його життєвий цикл.

Екземпляр програми, яка в даний момент виконується на комп'ютері, називають ***процесом***. Поняття процесу має кілька складових. По перше, це потік виконання команд який реалізує алгоритм програми. По друге — ***повторно використовувані ресурси***, що виділені процесу. Це процесорний час, оперативна та віртуальна пам'ять, канали вводу-виводу, відкриті файли та резервовані ресурси зовнішніх пристроїв. По третє — вміст регістрів процесора та значення внутрішніх змінних програми які разом визначають внутрішній стан процесу. Нарешті, це значення зовнішніх відносно процесу змінних, які ідентифікують процес та виділені йому ресурси і фіксують ***стан процесу*** з точки зору ОС. Ці змінні зазвичай є елементами оперативних структур даних ОС. ***Життєвий цикл*** кожного процесу можна відобразити в вигляді діаграми на якій позначені стани процесу та можливі переходи між ними. На рис. 1 приведена діаграма станів процесів в гіпотетичній ОС.

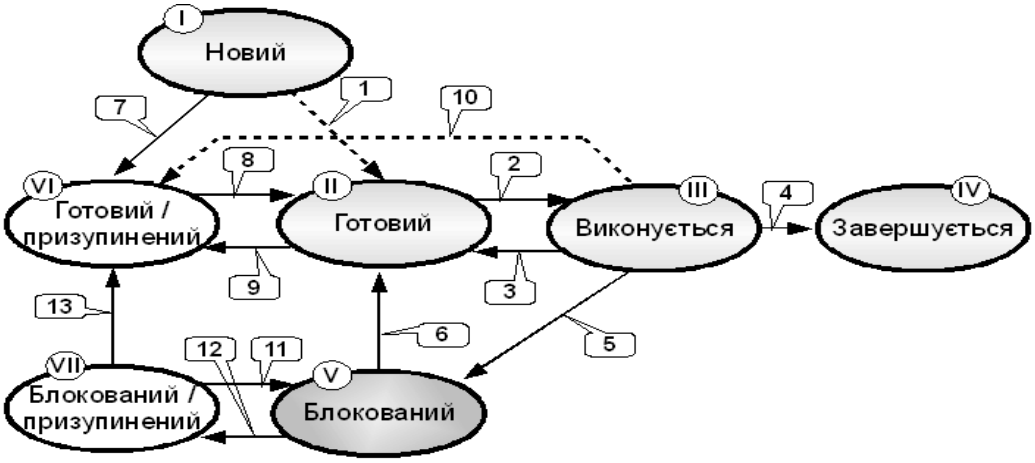

Рис. 1. Модель життєвого циклу процесу з сімома станами.

Розглянемо особливості кожного стану. "***Новий"*** процес (I) створюється ОС (часто за "дорученням" іншого процесу) для виконання певної програми. При цьому ОС створює та розміщує в пам'яті структури даних, що ідентифікують процес, зберігають інформацію про його стан та містять інформацію необхідну для управління процесом. Ці три типи інформаційних структур об'єднуються терміном ***управляючий блок процесу*** (process control block – PCB). Крім того ОС повинна розмістити в пам'яті код програми і дані та виділити пам'ять для стеків (як правило, окремі стеки для роботи в режимі ядра та в режимі користувача – див. далі) які використовуються для викликів процедур та збереження стану процесу. Разом PCB, пам'ять (точніше ***адресний простір*** — список адрес елементів пам'яті від деякої мінімальної до максимальної, до яких процес має доступ) програми, стеки і, можливо, сумісно використовуваний кількома процесами адресний простір складають ***контекст образу процесу***.

Якщо наявних в даний момент ресурсів в системі достатньо для виконання вищезгаданого то процес переходить в стан "***готовий***" до виконання (II). Коли ж ресурсів (в першу чергу пам'яті) недостатньо, процес ***призупиняється*** доки не з'явиться можливість їх виділення. При цьому ОС використовує механізм ***свопінгу*** (жарг. "підкачка") для переміщення образу процесу на диск, а при вивільненні ресурсів — зворотного переміщення в оперативну пам'ять і, таким чином, переводить процес із стану "***готовий / призупинений***" (VI) в "готовий" (переходи 8, 9).

Всі готові до виконання процеси знаходяться в упорядкованій черзі і час від часу переводяться ***диспетчером*** (компонентом ОС що реалізує почергове надання процесам процесорного часу) в стан "***виконується***" (III). При цьому процес отримує в своє розпорядження центральний процесор який виконує програму до завершення виділеного

кванту часу, або до запиту процесом обслуговування зі сторони ОС, результати якого не можуть бути отримані в межах поточного кванту часу. Наприклад, операція вводу даних в програмі на мові програмування високого рівня транслюється компілятором в виклик відповідного ***сервісу*** ОС (***системний виклик***) і наперед ніколи невідомо, коли після початку операції вводу дані будуть реально доступні в програмі. Тому диспетчер вивільняє процесор і переводить процес в стан "***блокований***" (V). В цьому стані процес буде перебувати до приходу ***асинхронного повідомлення*** (час приходу невизначений) про те, що очікувані дані вже доступні всередині ОС і операція вводу може бути завершена. Це повідомлення обробляється ОС яка і переводить процес в стан готового до виконання.

Якщо в блокованому стані знаходиться досить багато процесів і є процеси в стані "готовий / призупинений", то ОС може перерозподілити ресурси таким чином, щоб перевести деякі процеси із стану "готовий / призупинений" в стан "готовий" за рахунок вивантаження на диск частини блокованих процесів. Останні при цьому до моменту настання очікуваної події знаходитимуться в стані "***блокований / призупинений***" (VII).

Оскільки блоковані і блоковані / призупинені процеси очікують настання асинхронних подій — їх черги неупорядковані, тоді як черги готових та готових / призупинених упорядковані. Тип упорядкування для різних черг може бути різним і визначається ***стратегією планування*** (загальним планом діяльності, що охоплює тривалий період часу як спосіб досягнення складної мети, в даному випадку — максимальної продуктивності системи) і диспетчеризації процесів, яку реалізує дана ОС.

Результатом завершення виконання програми є перехід процесу із стану виконання в стан "***завершується***" (IV). Це не обов'язково означає видалення процесу із системи. Часто образ такого процесу містить інформацію необхідну іншим процесам. В цьому випадку частина структур РСВ деякий час зберігається і після завершення процесу. В ОС сімейства UNIX такий стан процесу називається ***зомбі*** (zombie).

І ще кілька загальних зауважень до діаграми на Рис. 1. На рисунку зображено всі можливі стани процесу і можливі переходи між ними. Одночасно в кожному стані може знаходитись багато процесів, за винятком стану "виконується" в якому може знаходитись не більше процесів ніж мається процесорів в обчислювальній системі. При цьому частину переходів (1, 2, 6 - 13) ініціює ядро ОС незалежно від тексту програми, що виконується. Перехід (3) як правило примусово виконує диспетчер при закінченні виділеного процесу кванту часу. Це так звана ***витісняюча багатозадачність*** (preemptive multitasking) що реалізована в більшості сучасних ОС загального призначення. При ***кооперативній*** моделі багатозадачності (cooperative multitasking) процес сам ініціює перехід (3), виконуючи системний виклик звільнення процесора. Перехід (4) відповідає виконанню системного виклику завершення процесу. Він може бути вказаний в програмі явно як виклик процедури або неявно по досягненню кінця програми. Можливий також варіант примусового завершення процесу іншим процесом який має відповідні права.

3. Архітектура ядра операційної системи.

Сучасна ОС загального призначення виконує багато різних функцій при тому, що рівні вимог до їх виконання різні. Так забезпечення скоординованого доступу програм до процесора, пам'яті, зовнішніх пристроїв, підтримка взаємодії між окремими процесами та сервісів файлової системи являються критично важливими для функціонування обчислювальної системи як з точки зору надійності, так і максимальної продуктивності. Тоді як, скажімо, реалізація інтерфейсу користувача як правило не є критичним фактором продуктивності обчислень але повинна забезпечити максимальну ефективність роботи оператора. Виходячи з цих міркувань розвинені ОС розробляють у вигляді комплексу програм та програмних модулів одні з яких (обслуговуючі програми – ***утиліти***) виконуються незалежно, а інші — під управлінням ***ядра*** ОС.

Ядро — центральна частина ОС, являє собою набір функцій, структур даних і окремих програмних модулів, які завантажуються в пам'ять комп'ютера при завантаженні ОС і забезпечують, як правило:

- управління введенням-виведенням інформації;
- управління оперативною пам'яттю;
- управління процесами;
- підтримку багатозадачності.

В залежності від ***архітектури*** (базової організації програмної системи, втіленої в її компонентах і відношеннях їх між собою та з оточенням) ядра конкретної ОС набір його функцій може розширюватись, або звужуватись порівняно із вказаним. Але в будь-якому випадку ядро є найбільш низьким рівнем абстракції для доступу програми до ресурсів системи. Можна сказати, що ядро є ***контролером*** ОС.

Перш ніж звертатись до класифікації архітектур ОС розглянемо питання режимів роботи процесора та їх відношення до виконання коду ядра. В сучасних ОС процедури ядра як правило працюють в ***привілейованому режимі*** процесора (інші назви — ***режим ядра***, ***режим супервізора***). В цьому режимі програми мають безпосередній доступ до всіх апаратних ресурсів таких як машинні команди спеціального призначення, порти зовнішніх пристроїв, реальні адреси елементів оперативної пам'яті та ін..

На відміну від привілейованого режиму ***захищений режим*** процесора, або ***режим користувача*** ізолює оперативну пам'ять, виділену процесу, від інших процесів. Крім того, доступ до зовнішніх пристроїв та виконання деяких машинних команд в режимі користувача можливий лише через виклики ядра ОС. Всі прикладні програми і утиліти ОС працюють саме в режимі користувача. Однак час від часу прикладний процес вимушений звертатися до сервісів ОС (наприклад для введення чи виведення даних). При цьому виконується послідовність команд системного виклику, процесор перемикається в привілейований режим і виконується код процедур ядра з подальшим зворотним перемиканням в режим користувача.

В сучасних комп’ютерах програми реалізують запити на обслуговування з боку ядра ОС через механізм ***програмних переривань*** (з точки зору програміста це просто машинні команди). Для зручності програмування ***програмний інтерфейс*** ОС реалізується у вигляді бібліотеки (або кількох) захищеного режиму які прийнято називати ***Application Program Interface*** (***API***).

Якщо процедури ядра і дані необхідні для обслуговування процесу, що викликав ядро, знаходяться в тому самому адресному просторі що і код самого процесу, то говорять що код ядра виконується в контексті прикладного процесу. Важливо що при цьому працює механізм захисту пам'яті і ділянки пам'яті, виділені для ядра, недосяжні для прикладних процедур навіть в межах одного процесу.

В ряді ОС процедури ядра виконуються в окремому адресному просторі (***автономне ядро***). В такій схемі поняття процесу застосовується тільки до прикладних процесів, а ядро

розглядається як окремий об'єкт що виконується в привілейованому режимі.

3.1. Типи архітектур ядра.

Поширена класифікація типів архітектур ядра виглядає так:

- монолітне ядро;
- модульне ядро;
- багаторівневе ядро;
- мікроядро;
- екзоядро;
- наноядро;
- гібридне ядро.

Монолітне ядро — найстаріший спосіб організації ОС який, в дещо зміненому виді, використовується і в наш час. В цій схемі всі компоненти ядра є складовими частинами однієї програми, використовують загальні структури даних і взаємодіють один з одним шляхом безпосереднього виклику процедур. Всі процедури працюють в привілейованому режимі процесора, як правило це автономне ядро. Набір функцій ОС, реалізованих в монолітному ядрі, найбільш широкий з поміж всіх типів архітектур. Реально реалізація всіх постійно необхідних під час роботи сервісів ОС зосереджена в ядрі. На жаль, монолітність ядер ускладнює їх модернізацію та налагодження, неправильна робота окремих процедур ядра несе великий ризик виходу з ладу всієї ОС. Великі розміри монолітних ядер вимагають багато місця для розміщення їх в оперативній пам'яті. Крім того при зміні складу обладнання комп'ютера необхідно заново перекомпілювати ядро.

Модульне ядро — сучасна модифікація архітектури монолітного ядра позбавлена двох останніх його недоліків. Модульні ядра, як правило, не вимагають повної перекомпіляції ядра при зміні складу апаратного забезпечення комп'ютера. Натомість модульні ядра надають той або інший механізм ***підвантаження*** окремих модулів ядра, що підтримують те чи інше апаратне забезпечення (наприклад, драйверів). Підвантаження модулів може бути як динамічним без перезавантаження ОС, так і статичним при перезавантаженні ОС після переконфігурації системи на завантаження тих або інших модулів. Всі модулі ядра працюють в адресному просторі ядра і можуть користуватися всіма функціями, що надаються ядром.

Модульні ядра вимагають для своєї роботи менше оперативної пам'яті завдяки можливості мати в пам'яті тільки необхідну в даний момент конфігурацію коду ядра. Вони зручніші для розробки, ніж традиційні монолітні ядра, оскільки від розробника не вимагається багаторазова повна перекомпіляція ядра при роботі над окремою його підсистемою. Виявлення і усунення помилок при тестуванні також полегшуються.

Більшість сучасних UNIX-подібних ОС, таких як Linux, FreeBSD, Solaris мають модульну (частково) структуру ядер і дозволяють під час роботи за потреби динамічно підвантажувати і вивантажувати модулі, що виконують частини функцій ядра.

Багаторівневе ядро є ще одним напрямком розвитку архітектури монолітного ядра. Основною ідеєю є організація ОС як ієрархії рівнів. Рівні утворюються групами функцій операційної системи - файлова система, управління процесами і пристроями і т.п. Кожен рівень може взаємодіяти тільки з своїм безпосереднім сусідом - вище- або нижчележачим рівнем. Прикладні програми або модулі самої ОС передають запити на обробку вгору і вниз по цих рівнях.

Такий структурний підхід дозволив певним чином структурувати ядро і потенційно забезпечити високий рівень захисту системи.

Однак у системах з багаторівневою структурою важко замінити одну реалізацію рівня іншою через множинність і розмитість інтерфейсів між сусідніми рівнями.

Найбільш відомим практичним втіленням архітектури багаторівневого ядра є ОС

MULTICS, де ідея ієрархії рівнів системи розповсюджувалась і на прикладні програмні системи. Багаторівневий підхід також використовувався при реалізації ряду варіантів ОС UNIX. Пізніше на зміну йому прийшла модель клієнт-сервер і зв'язана з нею концепція мікроядра.

Мікроядро — це модель ядра з мінімальною функціональністю. Класичні мікроядра надають лише невеликий набір системних викликів, що реалізовують такі базові сервіси ОС:

- управління пам'яттю;
- управління процесами;
- засоби комунікації між процесами.

Решта всіх сервісів ОС, які в класичних монолітних ядрах надаються безпосередньо ядром, в мікроядерній архітектурі реалізуються як процеси в адресному просторі користувача і називаються ***сервісами***. Прикладами таких сервісів, що виносяться в простір користувача в мікроядерній архітектурі, є мережеві сервіси, підтримка файлової системи, драйвери пристроїв.

Така конструкція потенційно дозволяє поліпшити загальну швидкодію системи за рахунок того, що компактне мікроядро може розміщатися в кеші процесора. За рахунок високого ступеня модульності істотно спрощується додавання в ОС нових компонентів. У мікроядерній ОС можна, не перериваючи її роботи, завантажувати і вивантажувати нові драйвери, файлові системи і т.д. Оскільки сервіси ОС нічим принципово не відрізняються від програм користувача, то можна застосовувати звичайні засоби розробки та істотно спростити процес розробки і налагодження компонентів ядра. Мікроядерна архітектура також підвищує надійність системи, оскільки помилка на рівні непривілейованої програми менш небезпечна, ніж відмова на рівні режиму ядра.

В той же час мікроядерна архітектура ОС вносить додаткові накладні витрати. Вони пов'язані з тим, що окремі сервіси, працюючи в захищеному режимі процесора, можуть взаємодіяти між собою лише шляхом передачі повідомлень, а це негативно впливає на продуктивність. Для підвищення швидкодії мікроядерної ОС необхідна ретельна проробка розбиття системи на компоненти з метою мінімізації трафіку повідомлень між ними.

Приклади ОС, що базуються на архітектурі мікроядра — QNX, Window CE, AIX, Minix3, Mac OS X, Symbian OS.

Екзоядро — один з сучасних напрямків подальшого розвитку мікроядерної архітектури. Це ядро ОС, що надає лише функції для взаємодії між процесами і безпечного виділення і звільнення ресурсів. Надання прикладним програмам ***абстракцій*** для фізичних ресурсів не входить в обов'язки екзоядра. Ці функції виносяться в бібліотеку захищеного режиму — так звану libOS, яка може забезпечувати довільний набір абстракцій, сумісний з тією або іншою вже існуючою ОС, наприклад Linux або Windows.

В порівнянні з ОС на основі мікроядер, екзоядра забезпечують набагато більшу ефективність за рахунок відсутності перемикання процесів при кожному зверненні до апаратного устаткування.

Наноядро — архітектура ядра ОС, в рамках якої украй спрощене ядро виконує лише одне завдання — обробку апаратних переривань, що генеруються пристроями комп'ютера. Після обробки переривань від апаратури наноядро, у свою чергу, посилає інформацію про результати обробки вищерозміщеному програмному забезпеченню за допомогою того ж механізму переривань.

Найчастіше в сучасних обчислювальних системах наноядра використовуються для ***віртуалізації*** апаратного забезпечення з метою дозволити кільком різним ОС працювати одночасно і паралельно на одному і тому ж комп'ютері. Наноядра також використовуються для забезпечення переносимості ОС на різне апаратне забезпечення або для забезпечення можливості запуску ОС на новому, несумісному апаратному забезпеченні без її повного переписування і перекомпіляції.

Найбільш відомі приклади використання — сервер ***віртуальних машин***

Vmware ESX Server, наноядро для Mac OS Classic з процесором POWERPC яке емулювало для ОС апаратуру процесорів Motorola 680x0, наноядро Adeos, що працює як модуль ядра для Linux і дозволяє виконувати одночасно з Linux яку-небудь іншу ОС.

І нарешті ***гібридні ядра*** як практичний результат спроб поєднати переваги мікроядерної архітектури з ефективністю монолітного ядра. Частіш за все являють собою модифіковані мікроядра, що дозволяють для прискорення роботи запускати частину сервісів в просторі ядра як це робиться в ОС з монолітним ядром.

Найбільш показовим прикладом змішання елементів мікроядерної архітектури і елементів монолітного ядра є ОС сімейства Windows NT. Інші приклади — ОС Syllable, BEOS, NetWare, окремі реалізації BSD.

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 (або ж 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 ***довільно зв'язаних*** систем.

Довільно 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.

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 виконання;
- 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 знижує переносимість програм між різними ОС.

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 рєєнтерабельною, 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 (процесу);
- процедура не повинна повертати покажчики на об'єкти, спільні 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. 8.1.

Таблиця 8.1. Види взаємодії процесів.

Ступінь обізнаності	**Вид взаємодії**	**Вплив одного процесу на на інші**	**Потенційні проблеми**
Процес не обізнаний про наявність інших процесів.	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 безпосередньо 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.

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.

В обох випадках ми стикаємося з трьома проблемами:

- Необхідність ***взаємних виключень*** (mutual exclusion) - тобто виключення процесом, який отримав доступ до ресурсу, що розділяється, можливості одночасного доступу до цього ресурсу для решти процесів (також використовують термін ***взаємовиключення***). 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. Якщо процес не знає про можливість існування інших процесів, він цілком являє собою критичний розділ відносно всіх спільних ресурсів! 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 доступу до 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 взаємне блокування і голодування — його неприємними наслідками.

Коли ж процеси безпосередньо обізнані про наявність один одного, 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 являє собою обмін повідомленнями 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***. Взагалі витратними ресурсами вважають інформацію, якою обмінюються процеси.

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 або 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:

- Чисто програмна реалізація (програмний підхід).
- Використання машинних команд спеціального призначення.
- Надання певного рівня підтримки з боку мови програмування або ОС.

9. Засоби взаємних виключень та синхронізації процесів.

9.1. Програмний підхід.

Програмний підхід до реалізації взаємних виключень полягає в програмуванні критичних розділів таким чином, щоб вони відповідали викладеним вище вимогам. Подивимось наскільки це просто (чи складно). Напишемо перший варіант взаємного виключення скажімо так.

Лістинг 1.

```
// Далі текст в стилі мови C а не на C

boolean flag = false; /* змінна блокування */

  /* Процесс 1 */                          /* Процесс 2 */
  while(flag);  /*АКТИВНЕ ОЧІКУВАННЯ*/      while(flag);
  flag = true;                             flag = true;
  /* КРИТИЧНИЙ РОЗДІЛ */                   /* КРИТИЧНИЙ РОЗДІЛ */
  flag = false;                            flag = false;
  /* РЕШТА КОДУ */                         /* РЕШТА КОДУ */
```

Тут **flag** – глобальна змінна, яка. управляє доступом до критичного розділу для всіх процесів. Такі змінні називаються ***змінними блокування***. Недолік алгоритмів такого типу – наявність стану ***активного очікування***, тобто ситуації коли в очікуванні можливості входу в критичний розділ процес безцільно споживає процесорний час виконуючи постійні перевірки стану змінної блокування. Блокування, що використовує активне очікування, називається ***спін-блокуванням*** (spinlock).

На жаль, приведений приклад (ліст. 1.) непрацездатний. Причину знайти неважко — припустимо процес 1 виконав перевірку змінної **flag**, вийшов із циклу і був переведений диспетчером в стан готового до виконання не встигнувши виставити ознаку зайнятості критичного розділу. Далі продовжив виконуватись процес 2, який також виконує перевірку змінної блокування. Оскільки значення **flag**, все ще **false**, він входить в критичний розділ і теж переривається. Тепер перший процес, продовжуючи свою роботу, також знаходиться в критичному розділі! Проблема виявляється в тому, що виконання процесу може бути перерване між перевіркою змінної блокування та її установкою. Задача виявилась зовсім не простою. Першим задовільне програмне вирішення задачі взаємного виключення для двох процесів отримав Деккер (T. Dekker). Більш простій алгоритм був запропонований в 1981 р. Петерсоном (G.L. Peterson). Детальніше обидва алгоритми розглядаються в [2].

9.2. Апаратна підтримка.

В однопроцесорній системі для того, щоб гарантувати взаємне виключення досить захистити процес від переривання його роботи на час перебування в критичному розділі. При цьому структура програми виглядає так (ліст. 2.).

Лістинг 2.

```
      /* БЕЗПЕЧНИЙ КОД */;
/* Заборона апаратних переривань */;
      /* КРИТИЧНИЙ РОЗДІЛ */;
/* Дозвіл апаратних переривань */;
      /* БЕЗПЕЧНИЙ КОД */;
```

Недоліки такого підходу: неможливість ефективної роботи диспетчера (ми його просто відключили) і неможливість застосування такого підходу в багатопроцесорній архітектурі (подумайте, чому).

Значно кращі результати дає використання спеціальних машинних команд. Усунути

проблему з ліст. 1 можна, наприклад, об'єднанням перевірки стану змінної блокування і установки її значення в одній машинній операції. В цьому випадку забезпечується коректний вхід в критичний розділ будь-якого процесу, що виконується в одно- або багатопроцесорній системі, оскільки елемент пам'яті, що зберігає змінну блокування, на час машинного циклу виявляється захищеним від доступу до нього з боку решти процесорів.

Алгоритм операції перевірки і установки значення може бути наприклад такий як на ліст. 3.

Лістинг 3.

```
// Далі текст в стилі мови C а не на C
 boolean testset(int i){
      if(i==0){
            i=1;
            return false;
      }
      else
            return true;
 }
```

Все це виглядає як одна машинна команда (т. зв. ***атомарна операція***). У деталях реалізація команди **testset** на різних комп'ютерах може відрізнятися. Так, процесори сімейства Pentium мають команду **CMPXCHG reg/mem,reg1** (порівняти і замінити), що використовує три операнди: операнд-джерело в регістрі **reg1**, операнд призначення в регістрі або в пам'яті **reg/mem** і акумулятор (регістр **EAX**). Якщо значення в приймачі і в акумуляторі рівні, операнд призначення замінюється на джерело. Інакше початкове значення операнда призначення завантажується в акумулятор.

Крім операції перевірки і установки в деяких процесорах мається атомарна операція обміну значень двох слів оперативної пам’яті яку теж можна використати для реалізації взаємовиключення.

На ліст. 4 приведений приклад реалізації взаємних виключень через спін-блокування з використанням інструкції перевірки і установки як вона визначена вище:

Лістинг 4.

```
// Далі текст в стилі мови C а не на C
 const int n = /* кількість паралельних процесів */;
 int bvar;
 void P(int i){
      /* Безпечний код */;
      while(testset(bvar));    /* АКТИВНЕ ОЧІКУВАННЯ */
            /* Критичний розділ */;
      bvar = 0;
      /* Решта коду */;
 }
 void main(){
      bvar = 0;
      parbegin(P(1),P(2),...,P(n));  /* Стартуємо паралельні процеси */
 }
```

Переваги підходу з використанням апаратної підтримки:

- Застосовується до будь-якої кількості процесів як в одно-, так і в багатопроцесорних системах.
- Простий, а тому легко перевіряється.
- Може використовуватися для підтримки безлічі критичних розділів (для кожного з них виділяється власна змінна блокування).

Недоліки:

- Використовується перечікування зайнятості. Тобто в очікуванні входу в критичний розділ процес продовжує споживати процесорний час.
- Можливе голодування. Якщо процес покидає критичний розділ, а в черзі на вхід чекають кілька інших, то вибір "щасливчика" довільний.
- Можливе взаємне блокування.

Ще один неприємний момент пояснимо на такому прикладі. Нехай процес P1 виконує **`testset`** і входить в критичний розділ, а потім P1 переривається процесом P2 з вищим пріоритетом. Якщо P2 спробує звернутися до того ж ресурсу, йому буде відмовлено і він увійде в цикл очікування. Проте і P1 не може ефективно продовжити виконуватись, оскільки є активний процес з більш високим пріоритетом (т. зв. ***проблема інверсії пріоритетів***). Зауважте, взаємоблокування в даному випадку немає.

Спін-блокування як простий і ефективний механізм синхронізації потоків широко використовується в ядрах ОС. Проблема перечікування зайнятості там стоїть не так гостро з тієї причини, що спін-блокування використовують для доступу до ресурсів на короткий час, а отже і очікувати їх вивільнення недовго. Але залишається проблема інверсії пріоритетів. Розглянемо її вирішення на прикладі ОС сімейства Windows NT.

В ядрі Windows NT спін-блокування призначені в основному для захисту даних, доступ до яких виконується на підвищених рівнях ***IRQL*** (Interrupt ReQuest Level – ***рівні запитів переривань***). Для вирішення можливої проблеми інверсії пріоритетів потрібний механізм, що не дозволяє коду з деяким рівнем IRQL переривати код з нижчим рівнем IRQL в той момент, коли останній володіє спін-блокуванням. Таким механізмом є підвищення поточного рівня IRQL потоку у момент захоплення ним спін-блокування до деякого рівня, що асоціюється із даним спін-блокуванням, і відновлення колишнього IRQL потоку після його звільнення. Код, що працює на підвищеному рівні IRQL, не має права звертатися до ресурсу, захищеного спін-блокуванням у якого рівень IRQL нижчий, ніж у самого викликаючого коду.

9.3. Семафори.

Дейкстра (Dijkstra E.) запропонував механізм співпраці двох або більшого числа процесів за допомогою простих сигналів, таким чином, що у визначеному місці процес повинен припинити роботу, до тих пір поки не дочекається відповідного сигналу.

Для сигналізації використовуються спеціальні змінні, які називаються ***семафорами***. Для передачі сигналу через семафор **`s`** процес виконує примітив (процедуру) **`signal(s)`**, а для отримання сигналу – примітив **`wait(s)`**. В останньому випадку процес зупиняється (блокується) до отримання сигналу. Як правило, є також примітив **`init()`** для присвоєння семафору початкового значення. Семафор розглядають як ціле, над яким визначено 3 операції:

- Семафор можна ініціалізувати невід'ємним значенням.
- Операція **`wait`** зменшує значення семафору. Якщо значення стає від'ємним, процес, що викликав **`wait`** блокується.
- Операція **`signal`** збільшує значення семафору. Якщо в результаті значення семафору не додатне, то деблокується один із заблокованих раніше операцією **`wait`** процесів.

Немає ніяких інших способів отримання інформації про значення семафору або зміни його стану. Передбачається, що примітиви **`wait`** і **`signal`** атомарні (не можуть бути перервані в процесі виконання). В термінах об'єктно-орієнтованого програмування можна сказати, що механізм семафорів забезпечує приховання (інкапсуляцію) змінних блокування так, що доступ до них можливий тільки через виклики методів об'єкту «семафор».

Формальне визначення примітивів семафорів може бути наприклад таким:

Лістинг 5.

```
// Далі текст в стилі мови С а не на С
 struct semaphore{
      int count;           /* семафор (змінна блокування) */
      queueType queue;     /* черга процесів які очікують на семафор */
 }
 void wait(semaphore s){
      s.count--;
      if (s.count<0){
           /* Помістити процес в s.queue */;
           /* Заблокувати процес */;
      }
 }
 void signal(semaphore s){
      s.count++;
      if (s.count<=0){
           /* Видалити процес з s.queue */;
           /* Помістити процес в список активних */;
      }
 }
```

Такі ***семафори-лічильники*** зручно використовувати для розділення між процесами багатьох однорідних ресурсів. Для цього достатньо ініціалізувати змінну **count** значенням початкової кількості ресурсів в ***пулі***.

Для захисту єдиного ресурсу можна використовувати семафор простішого виду. ***Бінарний семафор*** може приймати тільки значення 0 (блокований) і 1 (неблокований):

Лістинг 6.

```
// Далі текст в стилі мови С а не на С
 struct binary_semaphore{
      enum{zero,one} value;      /* змінна блокування */
      queueType queue;           /* черга процесів */
 }
 void waitB(binary_semaphore s){
      if (s.value==one)
           s.value = zero;
 else {
           /* Помістити процес в s.queue */;
           /* Заблокувати процес */;
      }
 }
 void signalB(binary_semaphore s){
      /* функція is_empty() повертає значення >0 якщо черга порожня */
      if (is_empty(s.queue))
           s.value = one;
      else{
           /* Видалити процес з s.queue */;
           /* Помістити процес в список активних */;
      }
 }
```

Широко використовується версія бінарних семафорів що називаються ***м'ютексами*** (mutex, скорочення від mutual exclusion). Для них зазвичай використовують позначення примітивів **mutex_lock** і **mutex_unlock** замість **wait** і **signal** відповідно. Неблокованому стану м'ютекса відповідає значення 0, а всі інші – блокованому.

Для зберігання процесів і потоків, що очікують як на звичайні семафори, так і на

м'ютекси, використовується черга (для кожного семафора своя черга). Найкоректніше для черги використовувати принцип FIFO (т. зв. ***сильний семафор*** – strong semaphore). При цьому першим з черги вибирається процес, який був заблокований довше за інших. Якщо порядок вибору з черги не визначений – семафор називають ***слабким*** (weak semaphore). Зазвичай ОС використовують сильні семафори, оскільки вони гарантують відсутність голодування.

9.4. Семафори в мовах програмування і ОС.

Семафори — високорівневий механізм взаємних виключень, який може надаватися мовою програмування або ОС. Здається першою мовою програмування, що отримала семафори, став Algol-68. Примітиви, які ми назвали **wait** і **signal**, в Algol-68 отримали імена **down** і **up** відповідно.

Інша класична мова програмування, що підтримує синхронізацію процесів за допомогою семафорів, – Modula-2. Рекомендований автором мови Н. Віртом як стандартний модуль визначень Processes приведений в ліст. 7:

Лістинг 7.

```
 DEFINITION MODULE Processes;
   TYPE SIGNAL;

PROCEDURE StartProcess(P:PROC; n:CARDINAL);  (* Почати паралельний
     процес, що задається програмою P з робочою областю розміром n *)
  PROCEDURE SEND(VAR s:SIGNAL);    (* Відновлює один з процесів,
                            що чекають на s *)
  PROCEDURE WAIT(VAR s:SIGNAL);    (* Чекати поки не пришлють сигнал s *)
  PROCEDURE Awaited(VAR s:SIGNAL): BOOLEAN;      (* Якщо TRUE –
                            хоча б один процес чекає на s *)
  PROCEDURE Init(VAR s:SIGNAL);    (* Обов'язкова ініціалізація *)
END Processes.
```

Відзначимо наявність процедури обов'язкової ініціалізації семафора **Init** і неблокуючу процедуру перевірки черги процесів, що очікують на семафор **Awaited**.

Що стосується сучасних систем програмування на мовах C/C++, то вони, як правило, мають в своєму складі бібліотеки підтримки засобів синхронізації процесів які спираються на можливості, що надаються ОС. Так наприклад, бібліотека MFC фірми Microsoft містить класи **CSemaphore** і **CMutex**, що надають механізми семафорів і м'ютексів для багатопоточних застосувань.

Розгляд підтримки взаємних виключень засобами ОС почнемо з систем сімейства UNIX. Системні виклики для управління потоками в UNIX-подібних ОС стандартизовані в частині стандарту POSIX (P1003.1c). Стандартом не обумовлюється спосіб реалізації викликів, так що вони можуть бути як справжніми викликами сервісів ядра системи, так і викликами бібліотечних функцій, що працюють в просторі користувача.

Таблиця 9.1. Деякі виклики стандарту POSIX для роботи з семафорами

Виклик	Опис
pthread_mutex_init	Створює новий м'ютекс
pthread_mutex_destroy	Знищує м'ютекс
pthread_mutex_lock	Захоплення м'ютекса (див. **wait**)
pthread_mutex_unlock	Звільнення м'ютекса (див. **signal**)
sem_init	Ініціалізація семафору
sem_wait	Зменшує значення лічильника (див. **wait**)
sem_post	Збільшує значення лічильника (див. **signal**)
sem_getvalue	Читає поточне значення лічильника семафору

Відмітьте, що семафори в стандарті POSIX є динамічними об'єктами. Окрім мьютексов і семафорів-лічильників стандарт пропонує механізм ***змінних стану*** для довготривалої синхронізації. Докладніше про використання змінних стану дивись в [1].

Що стосується конкретних реалізацій UNIX систем, то всі вони надають багатий набір механізмів синхронізації процесів. Так наприклад, UNIX System V має семафори-лічильники що складаються з наступних елементів:

- Поточного значення семафора.
- Ідентифікатора останнього процесу, що працював з семафором.
- Кількості процесів, очікуючих, поки значення семафора не збільшиться.
- Кількості процесів, очікуючих, поки значення семафора не стане рівним 0.

Семафори при створенні об'єднуються в множини, з одного або кількох семафорів. Завдяки цьому є можливість одночасного виконання ядром ОС групових операцій з рядом семафорів що допомагає боротися з проблемою взаємоблокування.

ОС Linux в основному успадковувала механізми синхронізації UNIX System V, в тому числі і групові операції з семафорами (на ліст. 8 приведені оголошення групових операцій з файлу **<sys/sem.h>**):

Лістинг 8.

```
/* Structure used for argument to `semop' to describe operations. */
struct sembuf {
  unsigned short int sem_num;   /* semaphore number */
  short int sem_op;             /* semaphore operation */
  short int sem_flg;            /* operation flag */
};
/* Semaphore control operation.  */
extern int semctl (int __semid, int __semnum, int __cmd, ...);
/* Get semaphore.  */
extern int semget (key_t __key, int __nsems, int __semflg);
/* Operate on semaphore.  */
extern int semop (int __semid, struct sembuf *__sops, size_t __nsops);
```

Крім того, Linux підтримує (на рівні системної бібліотеки glibc) засоби синхронізації стандарту POSIX 1003.1b.

У ОС сімейства Windows NT набір механізмів синхронізації, розрахованих на застосування при знижених рівнях IRQL, об'єднується поняттям ***диспетчерського об'єкту***. Загальною властивістю будь-якого диспетчерського об'єкту є те, що в кожен момент часу такий об'єкт знаходиться в одному з двох станів – сигнальному або несигнальному, а потік, що очікує захоплення об'єкту, блокується. У цьому плані єдиною відмінністю одного диспетчерського об'єкту від іншого є умова, по якій змінюється стан об'єкту (перехід в сигнальний або несигнальний стан). Диспетчерські об'єкти і правила зміни їх стану перераховані в табл. 9.2:

Таблиця 9.2. Диспетчерські об'єкти Windows NT

Тип об'єкту	Умова переходу в сигнальний стан	Результат для очікуючих потоків
М'ютекс	Звільнення м'ютекса	Звільняється один з очікуючих потоків
Семафор	Лічильник захоплень стає ненульовим	Звільняється деяка кількість очікуючих потоків
Подія синхронізації	Установка події в сигнальний стан	Звільняється один з очікуючих потоків
Подія сповіщення	Установка події в сигнальний стан	Звільняються всі очікуючі потоки
Таймер синхронізації	Наступив час або закінчився інтервал	Звільняється один з очікуючих потоків
Таймер сповіщення	Наступив час або закінчився інтервал	Звільняються всі очікуючі потоки
Процес	Завершився останній потік процесу	Звільняються всі очікуючі потоки
Потік	Завершився потік	Звільняються всі очікуючі потоки
Файл	Завершена операція введення-виведення	Звільняються всі очікуючі потоки

Особливістю Windows NT є те, що для очікування переходу в сигнальний стан будь-якого диспетчерського об'єкту використовується загальна функція очікування **WaitforSingleObject** (або **WaitForMultipleObjects** для кількох об'єктів).

Для м'ютексів Windows NT справедливе наступне:

- Захоплення м'ютекса відбувається в контексті конкретного потоку. Цей потік є власником м'ютекса і може захоплювати його рекурсивно. До речі, машинна команда CMPXCHG (див. вище) у разі неуспіху захоплення м'ютекса повертає ідентифікатор його поточного власника. Драйвер, що захопив, м'ютекс в контексті потоку, зобов'язаний звільнити його в тому ж контексті, інакше система буде зруйнована.
- Для м'ютексів передбачений наступний механізм виключення взаємних блокувань: при ініціалізації м'ютекса вказується його рівень (level). Якщо потоку потрібно захопити декілька м'ютексов одночасно, він повинен зробити це в порядку зростання їх рівня.

Семафори Windows NT ініціалізуються максимальним числом вільних ресурсів. При виклику функції очікування **WaitforSingleObject** або **WaitForMultipleObjects** лічильник семафору зменшується на 1 для кожного розблокованого потоку. Функція **ReleaseSemaphore** збільшує лічильник семафора на вказане параметром значення (звільняє вказане число ресурсів). При досягненні 0 семафор переходить в несигнальний стан. Захопити семафор може один потік, а звільнити – інший. Тому при розробці драйверів використовувати семафори необхідно з обережністю.

9.5. Задача виробника / споживача.

Окрім задачі взаємного виключення семафори використовуються і для розв'язання задач ***умовної синхронізації***. Розглянемо для прикладу дві канонічні задачі умовної синхронізації – задачу виробника-споживача та задачу читачів-письменників.

Однією з фундаментальних задач паралельних обчислень є ***задача виробника / споживача***. Важливість цієї задачі зокрема зумовлена тим, що ядру будь-якої ОС доводиться розвязувати її безліч разів. Задача формулюється так: є один або кілька виробників, що генерують дані (елементи) деякого типу (витратний ресурс) і поміщають їх в буфер (повторно використовуваний ресурс) і єдиний споживач, який вибирає з буфера елементи поодинці і використовує їх. Потрібно захистити програмну систему від перекриття операцій з буфером, тобто забезпечити одночасний доступ до буфера тільки для одного процесу.

Можна запропонувати багато вирішень з використанням тих або інших засобів синхронізації. Одне з них приведене в ліст. 9:

Лістинг 9.

```
#include <pthread.h>       // Оголошення POSIX-функцій для потоків
#include <semaphore.h>     // Оголошення POSIX-функцій для семафорів

const int SizeOfBuffer = 64;     // Такого розміру буде буфер
pthread_t PRODUCERID, CONSUMERID; // Ідентифікатори потоків
sem_t s;                         // Цей семафор захищає буфер
sem_t occupied;                  // Кількість елементів в буфері
sem_t available;                 // Розмір вільного місця в буфері

char Buffer[SizeOfBuffer];       // Ось він буфер для символів

void Produce(char* chIn);  // Процедура виробництва елементу chin

void Consume(char chOut);  // Процедура використання елементу chout
```

```
void* Producer(void* unused) {   // Це «оболонка безпеки» для виробника
      static int NextFree = 0;
      char chIn;

      while(true){
            Produce(&chIn);          // ВИРОБНИЦТВО ЕЛЕМЕНТУ
            sem_wait(&available);   // Чекаємо на вільне місце в буфері
            sem_wait(&s);    // і намагаємося отримати доступ до нього
            Buffer[NextFree]= chIn; // елемент в кільцевий буфер
            NextFree = (NextFree + 1) % SizeOfBuffer;
            sem_post(&s);            // Кінець критичного розділу
            sem_post(&occupied);    // Плюс один елемент в буфері
      }
 }

void* Consumer(void* unused) {   // Це «оболонка безпеки» для споживача
      static int NextChar = 0;
      char chOut;

      while(true){
            sem_wait(&occupied); // Чекаємо хоча б 1 елементу в буфері
            sem_wait(&s);      // і намагаємося отримати доступ до нього
            chOut = Buffer[NextChar]; // вибирати з буфера
            NextChar = (NextChar + 1) % SizeOfBuffer;
            sem_post(&s);          // Кінець критичного розділу
            sem_post(&available); // Плюс одне вільне місце в буфері
            Consume(chOut);        // ВИКОРИСТАННЯ ЕЛЕМЕНТУ
      }
 }

void main() {
      // ІНІЦІАЛІЗУЄМО СЕМАФОРИ:
      sem_init(&s, 0, 1);               // Буфер доступний
      sem_init(&occupied, 0, 0);      // Спочатку в буфері немає нічого,
      sem_init(&available, 0, SizeOfBuffer); // тобто от стільки місця
      // І СТАРТУЄМО ПОТОКИ:
      pthread_create(&PRODUCERID, NULL, &Producer, NULL);
      pthread_create(&CONSUMERID, NULL, &Consumer, NULL);
      // NULL – це «нульовий» покажчик
      // решта коду ...
 }
```

Тут для взаємних виключень при роботі з кільцевим буфером виробника і споживача використані семафори-лічильники в стандарті POSIX. Після деякого доопрацювання програму можна відкомпілювати і виконати наприклад в ОС Linux. Відмітьте, що з трьох семафорів тільки один використовується “за прямим призначенням” для взаємовиключення, а решта забезпечують необхідну синхронізацію роботи двох потоків.

9.6. Задача читачів / письменників.

Іншою класичною задачею паралельних обчислень є задача ***читачів/письменників***. Вона формулюється таким чином. Є багаторазовий ресурс (наприклад, файл бази даних) і невизначена кількість процесів одні з яких тільки читають зміст файлу (читачі), а інші можуть як читати, так і вносити зміни (письменники). Кілька читачів можуть одночасно працювати з ресурсом, а для письменника потрібен монопольний доступ. При вирішенні цієї задачі можна віддати пріоритет на право

доступу до ресурсу читачам чи письменникам. В першому випадку новий процес-читач отримає доступ до ресурсу, з яким уже працює інший читач, навіть якщо в черзі є процес-письменник. При пріоритеті на запис допуск нових читачів до ресурсу призупиняється як тільки в черзі з'являється процес-письменник. Вибір типу пріоритету визначається оптимальною стратегією яку «сповідує» програмна система для досягнення максимальних показників продуктивності і т.п..

Лістинг 10.

```
// Розв'язання задачі читачів-письменників з пріоритетом по читанню
// Далі текст в стилі мови C а не на C

int RdrCount = 0;           // Лічильник читачів (спочатку нікого)
semaphore CounterSem = 1,   // Для захисту RdrCount
          WrExSem = 1;      // Для взємовиключення при запису

void Reader(int n){
   while( true ){
      wait( CounterSem );   // --- Початок КР по RdrCount
      RdrCount++;
      if( RdrCount == 1 )   // Якщо це перший читач ..
         wait( WrExSem );   // .. заборонити доступ письменникам
      signal( CounterSem ); // --- Кінець КР по RdrCount

      READERWORK();         // Тут читаємо із ресурсу
           // далі виходимо:
      wait( CounterSem );   // --- Початок КР по RdrCount
      RdrCount--;
      if( RdrCount == 0 )   // Якщо це останній читач ..
         signal( WrExSem ); // .. дозволити доступ до ресурсу
      signal( CounterSem ); // --- Кінець КР по RdrCount

     // решта коду ...
   }
}

void Writer(int m){
   while( true ){
     // Для роботи письменника забезпечимо «повне» взаємовиключення:
      wait( WrExSem );      // --- Початок КР

      WRITERWORK();         // Тут запис в ресурс

      signal( WrExSem );    // --- Кінець КР
   }

void main(){
   parbegin( Reader(1),..., Writer(1),... );
}
```

Задача читачів/письменників на практиці розв'язується настільки часто, що деякі ОС навіть мають призначені для неї спеціалізовані примітиви синхронізації.

9.7. Бар'єри.

Цей механізм призначений для синхронізації багатьох процесів або потоків, які час від часу повинні всі знаходитися в деякому цілком певному стані.

Типові завдання, що вимагають такого типу синхронізацію - моделювання динаміки пучків заряджених частинок, плазми або галактик. При цьому розраховується переміщення десятків тисяч або мільйонів частинок за невеликий відрізок часу, потім по їх нових координатах розраховуються сили, що діють між ними, і цей цикл багато разів повторюється.

Розрахунки переміщення частинок можна виконувати паралельно. Проте розрахунок діючих сил можна починати тільки після того, як була переміщена остання частинка.

Таку синхронізацію реалізують, розміщуючи в кінці кожної фази виконання програми бар'єр. Коли процес підходить до бар'єру, він блокується (викликаючи функцію ***barrier***) до тих пір, поки решта процесів не дійде до бар'єру. Після цього блокування знімається одночасно зі всіх процесів. Як правило, бар'єри реалізовані у складі програмного забезпечення кластерів. Так, стандарт програмного інтерфейсу бібліотеки ***MPI*** (message passing interface – інтерфейс передачі повідомлень) визначає функцію **`MPI_BARRIER`**, що має описані властивості.

9.8. Монітори.

Семафори надають потужний і гнучкий механізм реалізації безпечної взаємодії процесів. Проте гнучкість завжди має зворотну сторону – програми, що створюються з використанням таких засобів, схильні до помилок, причиною яких є їх неминуче підвищений рівень складності.

В якості ілюстрації можна знову звернутися до програми з лістингу 9. Можете перевірити, що випадкова зміна порядку викликів примітивів синхронізації здатна зробити програму непрацездатною, притому це не завжди очевидно. А це ж всього лиш коротенький учбовий приклад. Помилки, пов'язані з неправильним використанням семафорів, можуть приводити до непередбачуваних і невідтворних критичних станів обчислювальної системи.

Звідси очевидна необхідність, в першу чергу для цілей прикладного програмування, мати засіб синхронізації більш високого рівня, який міг би забезпечити підвищення надійності паралельних програм. Таким засобом стали ***монітори***, запропоновані Хоаром (Hoare) і Брінч Хансеном (Brinch Hansen) в 1974 р. Основна ідея полягає в тому, щоб «заховати» ресурс, що захищається, всередину об'єкту який забезпечує взаємні виключення.

Монітор (версія Хоара - Хансена) – програмний модуль, що складається з ініціалізуючої послідовності, однієї або кількох процедур і локальних даних. Основні його особливості:

- Локальні змінні монітора доступні тільки його процедурам; зовнішні процедури доступу до локальних даних монітора не мають.
- Процес входить в монітор шляхом виклику однієї з його процедур.
- У моніторі в певний момент часу може виконуватися тільки один процес; будь-який інший процес, що викликав монітор, буде призупинений в очікуванні доступності останнього.

Очевидно, що об'єктно-орієнтовані ОС або мови програмування можуть легко реалізувати монітор як об'єкт із специфічними характеристиками. Якщо дані в моніторі являють собою якийсь ресурс, то монітор забезпечує взаємне виключення при зверненні до цього ресурсу.

Для застосування в паралельних обчисленнях монітори повинні забезпечувати синхронізацію процесів. Нехай процес, знаходячись в моніторі, повинен бути призупинений до виконання деякої умови. При цьому потрібен механізм, який не тільки призупиняє процес, але і звільняє монітор, дозволяючи увійти до нього іншому процесу. Пізніше, по виконанню умови, і коли монітор стане доступним, призупинений процес зможе продовжити свою роботу.

Монітор підтримує синхронізацію за допомогою ***змінних умов***, розміщених і доступних тільки в моніторі. Працювати з цими змінними можуть дві функції:

- **`cwait(c)`** призупиняє виконання процесу по умові **`c`**. Монітор при цьому стає доступним для використання іншим процесом.
- **`csignal(c)`** відновлює виконання деякого процесу, призупиненого викликом **`cwait`** з тією ж умовою. Якщо є декілька таких процесів, вибирається один з них; якщо таких процесів немає, функція не робить нічого.

Як бачимо, операції **`cwait / csignal`** монітора відрізняються від відповідних операцій семафора. Якщо процес в моніторі передає сигнал, але при цьому немає жодного очікуючого його процесу, то сигнал просто втрачається. Це означає, що змінні умов, на відміну від семафорів, не є лічильниками.

Хоаровськоє визначення моніторів (які ще називають ***моніторами з сигналами***) вимагає, щоб у випадку, якщо черга очікування виконання умови не порожня, при виконанні яким-небудь процесом операції **`csignal`** для цієї умови був негайно запущений процес з вказаної черги. Тобто процес, що виконав **`csignal`** повинен або негайно вийти з монітора, або завершитися.

У такого підходу є два недоліки:

- Якщо процес, що виконав **`csignal`** не завершив своє перебування в моніторі, то потрібно два додаткових перемикання процесів: для призупинення даного процесу і для відновлення його роботи, коли монітор стане доступним.
- Планувальник процесів, пов'язаних з даним сигналом, повинен бути ідеально надійним. При виконанні **`csignal`** процес з відповідної черги повинен бути негайно активований до того, як до монітора увійде інший процес і умова активації при цьому може змінитися. В результаті можливе взаємне блокування всіх процесів, пов'язаних з даною умовою.

Лемпсон і Ределл (Lampson, Redell) розробили інше визначення монітора для мови Mesa (така ж структура монітора використана і в мові Modula3). Примітив **`csignal`** замінений примітивом **`cnotify(x)`**, який інтерпретується наступним чином. Якщо процес, що виконується в моніторі, викликає **`cnotify(x)`**, про це сповіщається черга умови **`x`**, але виконання процесу продовжується. Результат сповіщення полягає в тому, що процес на початку черги умови відновить свою роботу в найближчому майбутньому, коли монітор виявиться вільним. Проте, оскільки немає гарантії, що інший процес не увійде до монітора раніше, при відновленні роботи процес повинен перевірити, чи виконана умова.

Важливо, що для монітора Лемпсона-Ределла можна ввести граничний час очікування, пов'язаний з примітивом **`cnotify`**. Процес, який прочекав повідомлення на протязі граничного часу, поміщається в чергу активних, незалежно від того, чи було повідомлення про виконання умови. При активації процес перевіряє виконання умови і або продовжує роботу, або знову блокується. Це запобігає голодуванню у випадку, якщо інші процеси збоять перед сповіщенням про виконання умови.

Також в систему команд можна включити примітив **`cbroadcast`** (широкомовне повідомлення), який викликає активізацію всіх очікуючих процесів. Це зручно, коли повідомляючий процес нічого не знає про кількість очікуючих процесів або не в змозі визначити, який з них треба викликати. Тому монітори Лемпсона-Ределла ще мають називу ***монітори із сповіщенням і широкомовленням***.

Додатковою перевагою монітора Лемпсона-Ределла перед монітором Хоара, є менша схильність до помилок. Якщо прийшло помилкове сповіщення, процес все одно перевірить виконання умови.

Монітори є структурним компонентом мови програмування і відповідальність за організацію взаємного виключення лежить на компіляторі. Останній зазвичай використовує мьютекси і змінні умов. Деякі ОС надають підтримку реалізації моніторів саме через ці засоби (ОС Solaris). Типовим представником мов програмування, в якому монітори є основним засобом синхронізації потоків, є Java. Додавання в опис методу ключового слова

synchronized гарантує, що в разі якщо один потік почав виконання цього методу, жоден інший потік не зможе виконувати інший синхронізований метод цього об'єкту. Тобто, у мові Java у кожного об'єкту, що має синхронізовані методи, є пов'язаний з ним монітор.

Для ілюстрації знову приведемо розв'язок задачі виробника-споживача але цього разу з використанням моніторів з операціями **cwait / csignal**.

Лістинг 11.

```
// Далі текст в стилі мови C а не на C

monitor BoundedBuffer {          // Фактично це визначення класу
   char Buffer[N];               // Буфер на N елементів
   int NextIn, NextOut;          // Поточні позиції в буфері
   int Count;                    // Пот. кількість елементів в буфері
   int NotFull, NotEmpty;        // Синхронізація (змінні умов)

   void Append(char x) {         // Точка входу в монітор для виробника
      if (Count==N)              // Якщо буфер заповнений –
           cwait(NotFull);       // блокуємось (споживач може працювати)
      Buffer[NextIn] = x;
      NextIn = (NextIn+1)%N;
      Count++;
      csignal(NotEmpty);         // Поновлення роботи споживача
   }

   void Take(char x){            // Точка входу в монітор для споживача
      if (Count==0)              // Якщо буфер порожній –
           cwait(NotEmpty);      // блокуємось (виробник може працювати)
      x = Buffer(NextOut);
      NextOut = (NextIn+1)%N;
      Count--;
      csignal(NotFull);          // Поновлення роботи виробника
   }

   // Секція ініціалізації:
   NextIn = 0;                   // Спочатку буфер порожній
   NextOut = 0;
   Count = 0;
   NotFull = OK;
   NotEmpty = OK;
}     // end monitor BoundedBuffer

void Producer(){ // Реалізація процедури виробника
      char x;
      while(true){
           Produce(x);     // Процедура генерації елементу даних
           Append(x);      // Ввійти в монітор, добавити елемент і вийти
      }
}

void Consumer(){          // Реалізація процедури споживача
      char x;
      while(true){
           Take(x);        // Ввійти в монітор, взяти елемент і вийти
           Consume(x);     // Процедура використання елементу даних
      }
}

void main(){    parbegin(Producer,Consumer);      }
```

9.9. Передача повідомлень.

Загальним недоліком семафорів і моніторів є те, що вони розраховані на роботу в одно- або багатопроцесорній обчислювальній системі із загальною пам'яттю (сильнозв'язаній системі). Якщо система складається з кількох процесорів, кожний з яких має свою власну пам'ять, а зв'язок між ними здійснюється через канали введення-виведення (довільно зв'язана система), жоден з розглянутих вище механізмів реалізації взаємних виключень непрацездатний. В цьому випадку можлива тільки співпраця процесів з використанням зв'язку.

Співпраця процесів з використанням зв'язку реалізується через механізм передачі повідомлень. Системи передачі повідомлень можуть бути різного типу, але в найбільш загальному вигляді вони, як мінімум, повинні надавати програмам засоби (процедури) для передачі повідомлення, наприклад **`send(отримувач,повідомлення)`**, та його прийому **`receive(відправник,повідомлення)`**.

Тут параметри **`отримувач`** та **`відправник`** є адресами. Адресація може бути ***пряма*** (direct), коли безпосередньо вказується адресат, і ***непряма*** (indirect) що використовує черги – поштові скриньки і порти. Адреса відправника може бути вказана неявно – параметр **`відправник`** набуває значення після отримання повідомлення.

Хоча обмін повідомленнями використовується як загальний механізм обміну інформацією між процесами, зараз нас більше цікавить його використання для синхронізації роботи процесів.

Відправлення повідомлення може бути блокуючим або неблокуючим. В першому випадку процес-відправник блокується до тих пір, поки адресат не отримає повідомлення. В другому випадку процесу для продовження роботи достатньо передати повідомлення підсистемі ОС, яка забезпечує його транспортування.

Так само є дві можливості і у процесу, який викликає процедуру **`receive`**: при блокуючому отриманні він буде чекати приходу повідомлення, а при неблокуючому – продовжить роботу незалежно від того, отримає повідомлення, чи його не виявиться.

Отже можливі різні комбінації:

- Комбінація блокуючого відправлення і блокуючого отримання (***рандеву*** - rendezvous) забезпечує тісну синхронізацію процесів.
- Неблокуюче відправлення і блокуюче отримання дозволяє швидко відправити кілька повідомлень різним адресатам. Використовується найчастіше.
- Неблокуюче відправлення і неблокуюче отримання дозволяє уникнути блокування отримувача при втраті повідомлення.

Щоб краще зрозуміти принципи синхронізації процесів «поштою», розглянемо реалізацію взаємних виключень з використанням повідомлень і поштової скриньки:

Лістинг 12.

```
// Далі текст в стилі мови C а не на C
const int np = /* кількість процесів */;
void P(int n){
      message msg;
      while (1) {
            receive(mbox, msg);         // Повинен бути блокуючий прийом!
            /* критичний розділ */
            send(mbox, msg);            // Передача блокуюча або неблокуюча
            /* решта коду */
      }
}

void main() {
      create_mailbox(mbox);             // Створили поштову скриньку
      send(mbox, null);                 // Помістили пусте повідомлення
      parbegin(P(1),P(2),.P(np));       // Стартували процеси
}
```

Відмітьте, що в даному прикладі синхронізацію процесів забезпечував сам факт обміну повідомленнями, а із їх змістом можете обходитись на свій розсуд.

Рекомендована література.

1. Таненбаум С. Современные операционные системы, 2-е изд.: - СПб.: Питер, 2004. – 1040 с.
2. Столлингс В. Операционные системы, 4-е изд.: - М.: Издательский дом "Вильямс", 2002. - 848 с.