Національний університет "Києво-Могилянська академія"
Факультет інформатики
Кафедра мережних технологій

Школа системного адміністратора

А.Ю. Дорошенко, В.М. Кислоокий, О.Л. Синявський

Архітектура і операційні середовища комп'ютерних систем

Методичний посібник і конспект лекцій

Київ 2005

Зміст.

Період 2: Технічні засоби не так дорогі, як раніше, люди дорогі.
Період 3: Технічні засоби (hardware) дуже дешеві, люди дорогі.
Сучасна функціональність операційних систем.

2.3. Функції і структура операційних систем.
Управління процесами (Process Management).
Управління пам'яттю (Memory Management).
Управління файловою системою (File System Management).
Управління дисками (Disk Management).
Системні виклики (System Calls).

2.4. Особливості методів побудови операційних систем.
Структура мережної операційної системи.
Однорангові мережні ОС і ОС з виділеними серверами.
ОС для робочих груп і ОС для мереж масштабу підприємства.

2.5. Управління процесами.
Поняття процесу.
Створення/Завершення процесу.
Виконання процесів (Process Execution).
Модель процесу на два стани.
Планування використання процесора: round-robin – карусель.
Переходи процесів у моделі процесів на два стани.
Чекання на те, щоб щось сталось...
П'яти – станова модель процесу.
Переходи між станами у п'яти – становій моделі процесу
Інформація, що характеризує стан процесу.
Блок управління процесом (Process Control Block (PCB))
Перемикання контексту.
Процеси в ОС UNIX.
Створення процесу в UNIX.
Модель процесів в UNIX.
Розподілений простір процесів.

2.7. Планування навантаження центрального процесора (CPU Scheduling).
Типи планувальників.
Довготерміновий планувальник (планувальник робіт – job scheduler).
Середньомермінове планування
Короткотерміновий планувальник(CPU scheduler)
Алгоритми планування.

Цілі планування завантаження центрального процесора.

Порівняння перерівиванного (Preemptive) і неперерівиванного (Non-Preemptive) алгоритмів планування.

Алгоритм планування: Першим прийшов – першим обслуговується (First-Come-First-Served - FCFS).

Алгоритм планування Round-Robin (карусель).

Алгоритм планування «Найкоротша робота – першою» (Shortest-Job-First - SJF).

Планування по пріоритетам (Priority Scheduling).

Багаторівневі черги планування (Multilevel Queue Scheduling).

Багаторівневі черги планування з зворотнім зв'язком (Multilevel Feedback Queue Scheduling).

Планування процесора у UNIX з використанням Multilevel Feedback Queue Scheduling

Планування в сердовищі JAVA.

Планування для мультипроцесорів зі спільною пам'яттю.

Розподілене планування та міграція процесів (Distributed scheduling and Process Migration).

Мотивації для розподілу навантаження

Міграція процесів

Міграція процесів у гетерогенній системі

2.8. Нитки (потоки управління).

Поняття про нитки (threads).

Два погляди на процеси

Процеси та нитки управління

Чому доцільно використовувати нитки?

Які види програм можуть бути багатопоточними?

Програми, що важко реалізуються як багатопоточні.

Використання ниток.

Використання Ниток у Сервері.

Нитки Рівня користувача.

Нитки рівня ядра.

Потоки рівня користувача та рівня ядра.

2.9. Комунікації між процесами або потоками управління

Взаємодія процесів.

Проблема Виробник – Споживач.

Реалізація у системі Java.

Основна програма

Засоби синхронізації у системі Java.

Використання моделі Клієнт/Сервер для передачі повідомлень

Передача повідомлень з використанням передачі і прийому - Send & Receive.

Прямі та непрямі комунікації

Буферизація

Використання моделі Клієнт/Сервер для передачі повідомлень

2.10. Віддалений виклик процедур (RPC - Remote Procedure Call)

RPC – парадигма.
Чому передача повідомлень не Ідеал?
Модель Клієнт/Сервер з використанням віддаленого виклику процедур.
RPC Виклик (Більш Деталізований опис)
Реалізація RPC.
Передача параметрів.
Передача параметрів (продовження).
Генерація Заглушок.
Зв'язування.
Підтримка сервером інформації стану.

2.11. “RMI - Виклик віддаленого методу” в Java.

Реалізація об’єктів та “Виклик віддаленого методу” в Java.

Виклик віддаленого методу.

Створення віддаленого об’єкту.

Приклад RMI програми.
Створення RMI-серверу
Компіляція RemoteServer
Створення програми-клієнта

Запуск Registry та виконання коду

2.12. Час у комп’ютерних системах.

Вимірювання часу.
Чому нас хвилює “час” у комп’ютерній системі?
Вимірювання великих відрізків часу.
Фізичні стандарти часу.
Мережний протокол часу - NTP.
Синхронізація фізичних годинників у комп’ютерних системах.
Централізовані алгоритми
Алгоритм з осередненням.
Алгоритм з координатором.
Розподілені алгоритми
Глобальне усереднення:
Локалізоване усереднення:
Відшкодування збитків, заданих відхиленням годинника (time drift)

2.13. Логічні годинники.

Чи достатньо синхронізувати фізичні годинники?
Від фізичних до логічних годинників
Події та впорядкування подій
Відношення “сталося раніше”
Логічний годинник Лемпорта
Умови (conditions), що задовольняються за допомогою логічних годинників
Впровадження (implementation) логічних годинників
Приклад логічних годинників
Обмеження логічних годинників

2.14. Проблеми синхронізації.
Семафори.
Суть проблеми.
Термінологія синхронізації.
Реалізація Взаємного виключення (Mutual Exclusion).
Методи реалізації взаємного виключення:

Один з можливих алгоритмів.

Семафори - підпримка Взаємного виключення (Mutual Exclusion) операційною системою.
Деталі операцій з семафорами

Замки, умовні змінні та монітори.
Від семафорів до замків (Locks) та умовних змінних (Condition Variables)
Замки (Locks).
Замки проти умовних змінних.
Умовні змінні (Condition Variables).
Використання замків та умовних змінних.
Дві різновидності умовних змінних
Монітори (Monitors).

2.15. Розподілене виключне виконання (Distributed Mutual Exclusion)
Взаємне виключення у розподіленому середовищі
Взаємне виключення у розподіленому середовищі – загальні вимоги
Центральний фізичний годинник
Центральний координатор
Розподілений алгоритм Лемпорта [(1978)].

Децентралізований алгоритм на основі часових міток.

Розширення для К спільних ресурсів.

Алгоритм Token-Ring

Алгоритм широкомовний маркерний.

Алгоритм деревоподібний маркерний

Алгоритми вибору (Election Algorithms).

Алгоритм Гарсіа-Моліна (Garcia-Molina's)
Алгоритм кільця Чанга та Робертса (Chang, Roberts)

2.16. Тупикові ситуації (Deadlocks).
Виявлення тупикових ситуацій.
Тупикова ситуація (Deadlock).
Умови виникнення тупикової ситуації.
Граф розподілу ресурсів (Resource-Allocation Graph).
Інтерпретація RAG з одним екземпяром ресурсу.
Поведінка у тупиковій ситуації.
Виявлення тупикових ситуацій (один ресурс кожного типу).
Виявлення тупикових ситуацій (багато ресурсів кожного типу).
Алгоритм визначення тупикових ситуацій (багато ресурсів кожного типу).
Приклад визначення тупикових ситуацій (багато ресурсів кожного типу).
Приклад тупикової ситуації та її уникнення.
Взаємоблокування
Зупинка, відновлення та закриття потоку в Java 2

2.17. Тупиковий стан у розподіленій системі (Distributed Deadlock).
Централізоване виявлення тупика
Перший алгоритм.
Другий алгоритм.

Розподілене виявлення тупиків.

Ієрархічне виявлення тупиків.

Після виявлення тупика: відновлення

2.18. Віртуальна пам’ять.

Сторінкова організація пам’яті (Paging).

Концепції управління пам’яттю.

Підкачка (Paging) по запиту (Virtual Memory).

Запуск нового процесу

Виняток у зв’язку з відсутністю сторінки в оперативній пам’яті (Page Faults).

Заміщення сторінок.

Розміщення фреймів

Пробуксовка (Thrashing).

Робочі комплекти.

2.19. Сегментація та сторінкова організація в архітектурі Intel x86.

Сегментація.

Адресація сегментів.

Імплементація сегментів.

Приклад сегментації.

Управління сегментами.

Сторінкова організація (Paging).

Імплементація сторінкової організації.

Приклад сторінкової організації.

Управління сторінками і фреймами.

2.20. Розподілена загальнодоступна пам’ять (Distributed Shared Memory).

Класифікація MIMD Архітектур.

Мультикомпьютери (розподілена - distributed пам'ять)

Мильтипроцесорні системи з однорідним доступом до пам'яті

Симетрична мильтипроцесорна система (SMP) типу UMA

Мильтипроцесорні системи з неоднорідним доступом до пам'яті

Розподілена Загальнодоступна Пам'ять (DSM).

Стислий огляд

Базисна ідея (Kai Li, 1986)

Порівняння MIMD Систем з загальнодоступною пам'яттю

Моделі Несуперечливості

Строга несуперечливість (найдужча модель)

Несуперечливість послідовності

Всі процеси бачать усі записи пам'яті в тому ж самому порядку (але не обов'язково)

Причинна несуперечливість

Несуперечливість процесора

PRAM несуперечливість
Слабка несуперечливість
Несуперечливість звільнення
Порівняння моделей несуперечливості
Імплементація послідовної несуперечливості
у основаній на сторінковій організації DSM
Сторінки не копіюються, не переміщаються.
Сторінки не копіюються, але переміщаються
Сторінки копіюються та переміщаються
Сторінки копіюються, але не переміщаються

2.21. Ввод-вивід.
Керування вводом-виводом
Фізична організація пристроїв вводу-виводу
Програмне забезпечення вводу-виводу
Обробка переривань
Драйвери пристроїв
Незалежний від пристроїв шар операційної системи
Користувальницький шар програмного забезпечення
2.22. Структури файлових систем.
Абстракція файлової системи.
Цінність файлових систем.
Інтерфейс користувача з файловою системою.
Операції з файлами.
Загальні моделі доступу до файлів.
Операції з файлами (продовження).
Каталогізація і наіменування.
Проста система іменування.
User-based система іменування.
Multilevel система іменування.
Реалізація файлових систем
Апаратні засоби диску (Disk Hardware).
Структури даних для файлів.
Структури даних ОС для файлів.
Стуктури даних у UNIX для файлів.
Структури даних на диску для файлів.
Файлові системи у UNIX.

2.23. Розподілені файлові системи (Distributed File Systems).

Сервери розподіленої файлової системи – файл сервіс інтерфейс
Розподілена структура імен (Distributed Naming Structure)
Мережна файлова система Sun (Sun’s Network File System)
Перетворення віддалених файлових систем
2.24. Архітектура програмного забезпечення NFS
Протокол NFS
Кешування у NFS
Читання файлу
Записи файлів

Частина 3.
Системне та мережеве адміністрування.

3.1. Розвиток засобів системного та мережевого адміністрування.
Система моніторингу Інтернет сервісів.
Моніторинг сервісів Інтернет
Моделювання і моніторинг діяльності
Моніторинг критично важливих серверів
Керування на основі Service Level Agreement (SLA)
Масштабованість
Моніторинг від клієнта до магістралі мережі
3.2. Система керування вузлами мережі.
Керування вузлами мережі.
Можливості системи керування вузлами мережі.
Наочне представлення мережі
Керування мережами – від малих до великих
Обробка подій
Проактивне керування за допомогою генерації звітів і сховища даних
Прискорений інтелектуальний збір даних
Генерація звітів для ключових даних
Цілодобовий доступ до мережі з будь-якого місця
3.3. Система керування середовищем широкомовної передачі.
Моніторинг і керування трафіком широкомовної передачі.
Реалізація і керування середовищем широкомовної передачі
Функції системи керування середовищем широкомовної передачі.
Автоматичне виявлення зв'язків у топології маршрутизації широкомовної передачі
Інтерактивне відображення топології багатоадресної передачі
Вимірювання інтенсивності широкомовного трафіка

3.4. Система керування розподіленим обчислювальним середовищем.
Основні функції системи.
Використання комплексних концепцій керування

Зв'язок зі службою підтримки і звіти про якість IT-послуг
Додаткова безпека в системі.
Засоби керування і контролю якості роботи додатків.
Керування додатками

3.5. Megavision Web

Загальні характеристики.
Основні можливості
Загальні:
- Обробка подій у мережі
- Моніторинг продуктивності
- Інтерфейси
- Конфігурування пристроїв
- Захист інформації

Система обробки неполадок Event Horizon для MegaVision
Віддалений моніторинг і журнал подій
Моніторинг спрацьовування датчиків.
Інтегрований віддалений доступ
Віддалене керування
Віддалений моніторинг стану навколишнього середовища
Відправка оповіщень по електронній пошті

Передмова

Розробки сучасного програмного забезпечення характеризуються динамічним поступом у розвитку і використанні архітектурних засобів і принципів побудови операційних систем. Для системного і прикладного програмного забезпечення в сучасних умовах характерні такі риси, як використання концепцій паралельних і розподілених обчислень, мережних технологій та засобів Інтернет. Проте застосування цих та інших новацій вимагають грунтовних знань про базові архітектури апаратного і програмного забезпечення, а також про процеси операційних середовищ, що лежать в основі будь-яких комп'ютерних прикладних систем.

Метою навчального посібника "Архітектури і операційні середовища" є, з одного боку, висвітлення основних положень з побудови та засобів архітектури і операційних середовищ обчислювальних систем, що вже стали класичними, а з іншого – ознайомлення студентів з новими тенденціями у розвитку цієї дисципліни, а також набуття ними практичних навичок у розробці та застосуванні сучасного програмного забезпечення.

В основу посібника покладені матеріали курсів, прочитані авторами на бакалаврській та магістерській програмах факультету інформатики Національного університету "Києво-Могилянська Академія" у 1998-2003 роках.

В методичному плані курси спираються на знання, одержані студентами у циклі фундаментальних дисциплін з теорії алгоритмів, мов програмування, паралельних обчислювальних систем та мережних технологій і є складовою частиною нормативних дисциплін факультету інформатики.

Прохання надсилати відгуки, зауваження та побажання щодо матеріалів з цього курсу лекцій, що будуть враховані авторами у подальшій роботі, на кафедру мережних технологій факультету інформатики Національного університету "Києво-Могилянська академія".

Вступ

Історія розвитку комп'ютерної науки і індустрії налічує вже понад 50 років. Ідеї створення програмованих обчислювальних пристроїв, які діють за програмою, що зберігається в пам'яті, були відомі ще в середині XIX ст. завдяки роботам англійського дослідника Ч. Беббіджа. Але успішне втілення цих ідей стало можливим тільки у перших електронних обчислювальних машинах (ЕОМ), що з'явились після другої світової війни.

Перший в Україні комп'ютер (що став також першим не тільки у колишньому Радянському Союзі, але і у континентальній Європі) став до ладу у Києві у 1951 році. Загальні принципи. побудови електронних обчислювальних машин з програмою, що зберігається в пам'яті, були сформульовані американським математиком Дж. фон Нейманом. (1946 р.). Ці принципи, або як тепер звичайно визначають – *архітектура* – були принципами послідовних обчислень і передбачали: 1) послідовну організацію пам'яті; 2) процес послідовного виконання команд (операцій програм) у порядку їх розташування в пам'яті машини; 3) послідовну роботу пристроїв (арифметико-логічний пристрій, регістри, пристрої оперативної та довготривалої пам'яті), робота яких контролюється єдиним пристроєм керування. Але вже тоді розробникам обчислювальної техніки було зрозуміло, що наявність одного виконавчого пристрою і єдиного пристрою керування не є принциповим обмеженням. Більше того, оскільки ЕОМ складалася з пристроїв різної швидкості, це призводило до значних втрат потенційної продуктивності обчислювальної машини.

Методи паралельної обробки інформації виникали і удосконалювались у ході загального розвитку архітектур обчислювальних машин. У другому поколінні ЕОМ разом з операційними системами та системами програмування виникли методи мультипрограмування, тобто виконання декількох програм на одному і тому ж обладнанні. Проте комп'ютери залишались практично однопроцессорними. Машини третього покоління далі розвинули методи мультипрограмування. В архітектурі ЕОМ стали існувати канали – спеціалізовані процесори, що працювали паралельно з центральними процесорами і виконували виключно операції вводу/виводу, а також було введено у практику використання мультимашинних систем. Проте це ще не були в повному розумінні мультипроцесорні системи. Ідея паралелізму обчислень знаходила собі втілення поступово [1]. Тільки коли значно виросли потреби у високопродуктивних і високонадійних обчисленнях, виокремились такі компоненти системи програмного забезпечення ЕОМ як система програмування і операційна система) і визріла економічна доцільність побудови мультипроцесорних ЕОМ, створення параллельних обчислювальних систем (тобто комп'ютерних систем паралельної дії) стало реальною справою [2].

Сучасні комп'ютерні системи – це складні обчислювальні комплекси, що фактично є системами паралельної дії (наіть у найпростіших вріантах, наприклад, настільних), що забезпечує одночасне (паралельне) виконання операцій по одній або декількох програмах. Як правило, сучасний комп'ютер

підключений до мережі, локальної і глобальної. Тому традиційне поняття операційної системи для такого комп'ютера трансформується у більш складне утворення – *операційне середовище*, в якому з'являються, розвиваються і зникають обчислювальні процеси.

Таким чином, архітетктура і операційне середовище комп'ютерної системи тісно пов'язані між собою і є її основоположними рисами, що визначають принципи побудови і фунціювання всієї програмної надбудови включно з її прикладною частиною. Тому досконалі знання цих категорій для програміста, а надто системного, є передуумовою його успішної роботи у створенні, налагодженні та адмініструванні комп'ютерних систем будь-якої складності.

Автори цього навчального посібника поставили собі за мету дати студентові базові знання з архітетктури та операційних середовищ сучасних комп'ютерних систем і, разом з тим, стоворити передумови для самостійного вивчення ними нових розробок у цій царині, що динамічно розвиваються.

Частина 1.

Сучасні архітектури та засоби програмування мультипроцесорних систем

1.1. Особливості архітектури високопродуктивних мікропроцесорів

З 80-х років минулого століття основою архітектури комп'ютерних систем є мікропроцесори. За законом Мура, що підтверджується уже понад 30 років, кожні півтора року потужність мікропроцесорів подвоюється. А оскільки тактова частота мікропроцесорних чипів зростає значно повільніше, робимо висновок, що збільшення потужності відбувається також за рахунок удосконалення архітектури мікропроцесорів.

Така ж ситуація складається і із зростанням продуктивності комп'ютерних систем на на рівні макроархітектури. Якщо за останню декаду потужність мультипроцесорних систем зросла у 500 разів, то за той же час потужність мікропроцесорів ~ тільки у 15 разів [3]. Це говорить про те, що швидкість зростання потреб у високопродуктивних обчисленнях настільки велика, що вона може задовольнятися не стільки за рахунок покращення фізичних параметрів мікропроцесорних виробів, скільки завдяки радикальним змінам в архітектурі як мікропроцесорів, так і мультипроцесорних систем у цілому.

У чому ж полягають такі радикальні зміни, і які загальні тенденції їх виявлення? На ці питання ми дамо відповідь у даному розділі. Але спочатку про зростання продуктивності паралельних систем загалом. Для того, щоб в наступну декаду як і в попередні 10 років продуктивність паралельних систем могла виростина три порядки, тобто з 1 Тфлопс (10^{12} операцій у форматі з плаваючою точкою) до 1 Pflops (10^{15} флопс), сучасні дослідники вбачають три основні шляхи [3]:

- подальше підвищення тактової частоти мікропроцесорів у декілька разів;
- винайдення нових архітектурних рішень в системах і технологіях зв′язку між процесорами та систем вводу/виводу, що дадуть також підвищення продуктивності у декілька разів;
- підвищення їх продуктивності *на 2 порядки* за рахунок посилення архітектурних рішень для глибокого розпаралелювання обчислень на всіх рівнях комп'ютерної системи.

Перший з зазначених шляхів має обмеження технології CM0S (КМОН – комплекс метал-окисел-напівпровідник) і, за оцінками спеціалістів, має бути вичерпаний в найближчі 10-15 років. Обмежене значення (тільки в кілька разів) має і другий напрямок, що зорієнтований на удосконалення механізмів обміну та вводу-виводу.

Найбільші можливості пов′язані з розвитком третього напрямку, основний зміст якого крім власне розпаралелювання полягає у вирівнюванні і

балансуванні швидкодії процесорів та позапроцесорних пристроїв. Проблема полягає у тому, що продуктивність процесорів росте приблизно на 60-70% річно, а пристроїв оперативної пам′яті – тільки на 7%. Отже для подолання майже 10-кратного відставання пристроїв пам’яті від зростання продуктивності процесорів необхідні серйозні зрушення в архітетктурі мікропроцесорів. Основними напрямками подолання такої диспропорції є такі:

1) збільшення обсягів пам′яті на кристалі процесора та організація багаторівневої кеш-пам′яті;
2) збільшення пропускної здатності пам′яті за рахунок так званої інтелектуальної процесоро-пам′яті (PIM – processor in memori) до 256 біт 4096 за такт (до 50 Гбайт/сек);
3) перехід до мультипотокових та мультипроцесорних архітектур, тобто збільшення гранульованості обмінів між процесором та кристалами пам′яті;
4) інтеграція на кристалі процесора спеціалізованих пристроїв обробки, таких як мультимедійних, пристроїв керування періферійним обладнаннямп процесора, телекомунікаційних, мережних та ін., що раніше реалізувалися позакристальними пристроями.

Впровадження цих тенденцій покликане удосконалити домінуючу архітектуру RISC (Reduced Instruction Set Computer) мікропроцесорів, подальший ріст продуктивності яких є неадекватним ускладненню їх арітектури та швидкості зростання тактової частоти. На відміну від архітектури CISC (Complex Instruction Set Computer), типовим представником якої є процесори Intel від x86 до Pentium II, процесори RISC націлені на високу продуктивність, що досягається, зокрема, завдяки тому, що:

– однакова довжина і формат команд;

– операнди – тільки регістри, яких є багато на кристалі;

– операції виконують прості дії і виконуються переважно за 1 такт;

– широко використовуються ковейєрні механізми.

Відомими прикладами є мікропроцесори SPARC, MIPS, Alpha, Power PC, PA-RISC та інші, виробниками яких є відповідно Sun, MIPS, DEC, IBM, HP та інші.

Для вдосконалення архітектури RISC основні виробники вибирають два шляхи: еволюційний і революційний. Революційним шляхом ідуть Intel та HP, що запропонували архітектуру IA-64 та технологія EPIC (Explicitly Parallel Instruction Computing). Конкретні процесори, що виробляються за цією архітектурою (на 2002 рік): Merced/Itanium та McKeanly. Ключовими характеристиками архітектури IA-64 (Intel Arhitecture), що призначається для робочих станцій та серверів на основі 0,18 мікронної технології, є такі:

– велика кількість регістрів: 128(64)+128(80)+64(1);

– маштабованість архітектури до великої кількості фукціональних пристроїв (inherenty scalable instruction set);

–явний паралелізм EPIC на рівні команд (ICP), що визначається не процесором, а компілятором;

– паралельна обробка умовних гілок програм (Prediction) з відкиданням результату, що не є справжнім;

– завчасна загрузка даних з (повільної) основної пам′яті (speculative loading).

Що може запропонувати сучасний рівень мікроелектроніки для виробництва високопродуктивних мікпроцесорів, можна бачити на прикладі Intel Itanium-2 (McKeanly) [4]

- 900-1200 МГц; технологія 018 мк; 64-розр.; EPIC;
- 3-рівневий кеш: 32K(L1),256K(L2), 3M(L3);
- 221млн. транзисторів – найбільший процесор від Intel, 421 мм^2.(для порівняння, Pentium IV має 55 млн.);
- 128-розрядна внутрішня шина з тактовою частотою 400Мгц і швидкістю обміну 64 Гбайт/с;
- 8-стадійний конвеєр команд, 11 потоків (каналів);
- продуктивність 6 команд/такт;
- призначення – серверні архітектури симетричної мультиобробки SMP і кластерні архітетктури.

Еволюційний шлях пропонують розробники IBM та COMPAQ (DEC). Їх процесори такі, як Power4 та Alpha21364 мають такі головні архітектурні надбання від RISC:

- передбачення переходів;
- динамічне (out-of-order) призначення команд на пристрій виконання.

Разом з тим до них розробники IBM і COMPAQ додають нових рис у можливості паралельної обробки, а саме:

- інтеграція на кристалі мережного інтерфейсу високошвидкісних каналів мікропроцесорного зв′язку для мультипроцесорних утворень; (історично, вперше така інтеграція відбулась в трансп’ютерах [5]);
- введення можливостей мультипроцесорної (IBM) та мультипоточної (COMPAQ) обробки на кристалі одного мікропроцесора;
- вирішення завдань розпаралелювання як динамічним (апаратним), так і статичним шляхом – за допомогою компілятора;
- інтеграція на кристалі інтерфейсу вводу/виводу.

Це може давати значний ефект для підвищення ступеня розпаралелювання виконанння операцій. Наприклад, у процесорі Alpha 21364 одночасно може знаходитись в обробці для динамічного розпаралелювання до 80 команд (найбільша кількість для сучасних мікропроцесорів) [4]. Для цього мікропроцесорне ядро містить чотири АЛП і два пристрої з плаваючою точкою, файли регістрів, логіку динамічного розпаралелювання та черги команд для цілочисленних даних та даних з плаваючою комою.

Принципова відмінність Power4 від IBM – двопроцесорне мікропроцесорне ядро, і підтримка розпаралелювання на рівні потоків (Thread Level Parallelism).

1. 2. Організація паралельних обчислювальних систем (архітектурні компоненти)

Пошук вирішення двох головних проблем ком′ютерної техніки – високої продуктивності та надійності – призвели до розробки паралельних обчислювальних систем (ПОС). З іншого боку, для появи комп’ютерних систем паралельної дії були і внутрішні причини, що логічно випливали з поступального розвитку концепцій програмування обчислювальних систем. Йдеться, перш за все, про появу операційних систем та розробку мультидоступу до обчислювальних послуг одночасно багатьох користувачів. В результаті тепер паралелізм обчислень є по суті всюдисущим і складає одну одну з фундаментальних засад побудови сучасних комп’ютерних систем від настільних систем до суперкомп’ютерів. Зазначимо також, що в літературі використовується і більш загальна назва ПОС – високопродуктивні обчислювальні системи (High Performance Computing Systems) [6].

Найбільш загальниими класами ПОС є мультипроцесорні системи та мультимашинні системи і мережі, а головними архітектурними компонентами для них є [6,7]:

1) процесори (або процесорні елементи) , їх типи і кількість;
2) організація основної пам′яті;
3) система зв′язку, керування та синхронізації паралельними процесорами;
4) організація системи вводу/виводу.

В залежності від типу процесорів архітектури паралельних систем поділяються на: векторно-конвейерні (від таких виробників, як Cray, NEC, Fujitsu); матричні (наприклад, ILIAC-IV [1,5]); асоціативні; систолічні; нейроподібні. Системи трьох останніх типів не є масовими, тому їх приклади не наводяться.

Основна пам′ять паралельної системи може бути *спільною* або *розподіленою*.

До важливих властивостей основної пам′яті належить також її обсяг, а також такі ознаки, як: 1) наявність кеша, тобто надшвидкої пам′яті для збереження тимчасових наборів даних, 2) буферів команд і даних, 3) векторної пам′яті – регістрових файлів для збереження секторів команд і даних; 4) розшарованості пам′яті – для забезпечення збільшення пропускної здатності пам′яті.

До спільної (shared) пам′яті процесори мають доступ рівноправний (жоден з процесорів не має переваг перед іншими) і рівнозначний (час доступу до всіх ділянок пам′яті однаковий). Будь-який процес, щоб звернутися до будь-якого слова пам'яті, може використовувати звичайні операції запису/ читання. Зв'язок між процесами завдяки розташуванню у спільній пам'яті простий. Синхронізація добре вивчена, використовуються класичні методи (семафори, і т.п.).

У комплексі апаратних засобів – шина (bus) стає критичним параметром при більше ніж 10-20 CPUs

Мильтипроцесорні системи з однорідним доступом до пам'яті називаються системами типу UMA = (Uniform Memory Access). Їх архітектура відображена на Рис 1.1.

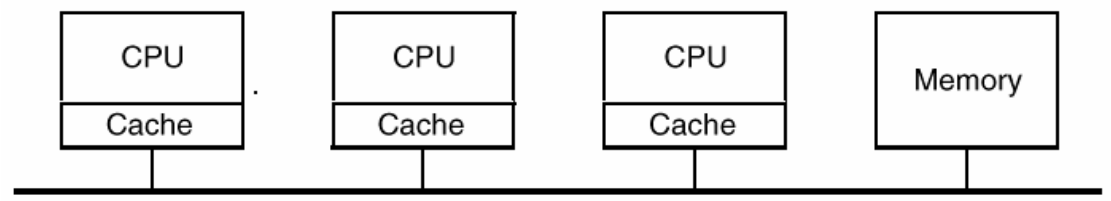


Рис1.1. Симетрична мильтипроцесорна система (SMP) типу UMA

Багато центральних процесорів (2-30), і одна загальнодоступна фізична пам'ять сполучені шиною. Кеші повинні забезпечувати несуперечливість.

При “ кеш-попаданні” читання (read hit) дані вибираються з локального кеша. При “ кеш-промаху” читання (read miss) дані вибираються з основної пам'яті та заносяться у локальний кеш. Одні і ті ж самі дані можуть бути у багатьох кешах.

При записі слова дані зберігаються в пам'яті та локальному кеші. Інші кеші виконують snooping («Snoop» - “совати ніс у чужі справи”): якщо вони мають це слово, вони оголошують вміст кеша невірним. Після того, як запис завершується, пам'ять є оновленою і слово є тільки в однім кеші.

При розподіленій пам′яті кожен з процесорів має свою, приватну, ділянку пам′яті, прямий доступ до якої (а значить найшвидший) має тільки він. Доступ до “чужої” пам′яті можливий тільки через спеціальний механізм зв’язку між процесорами і тому є значно повільнішим. Спільна пам′ять паралельних процесорів є безпосереднім узагальненням моделі послідовних обчислень і тому має перевагу простішої моделі програмування. Проте архітектури з спільною пам′яттю мають невелику масштабованість і тому завжди є “малопроцесорними”. Архітектури з розподіленою пам′яттю, навпаки, здатні до широкого масштабування, але мають складніші засоби програмування порівняно зі спільною пам′яттю. Для поєднання сильних сторін обох видів пам′яті в сучасних архітектурах використовують проміжний варіант – розділена (фізично) спільна (логічно) пам′ять (distributed shared memory), що приводить до архітектури NUMA (NonUniform Memory Access) – пам′яті з неоднорідним доступом (див. Рис 1.2.)

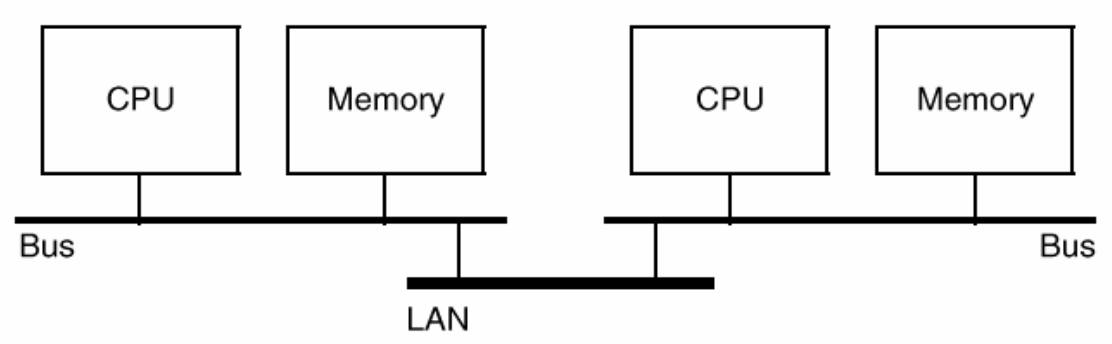


Рис 1.2. Мультипроцесорна система з неоднорідним доступом до пам'яті типу NUMA = (NonUniform Memory Access).

Багато центральних процесорів, кожний із власною (фізичною) пам'яттю. Вони спільно використовують єдину віртуальну пам’ять. Доступ до локальних розташувань у пам'яті набагато швидший (можливо у десятки разів), ніж доступ до віддалених розташувань у пам'яті. Ніяких апаратних засобів хешування, тому що має значення, в якій пам'яті збережені дані.

Посилання до віддаленої сторінки викликає апаратну подію - page fault - “відмова сторінки”, OS обробляє внутрішнє переривання і переміщає сторінку у локальну машину

Для ефективного функціювання паралельної системи важливу роль відіграють *засоби зв’язку* процесорів між собою, з пам′ятю та засоби синхронізації паралельних процесів. До засобів зв’язку належать:

1) спільна пам′ять – як канал зв′язку між процесорами;
2) спільна шина (*bus*), що розподіляється у часі процесорами, пристроями пам′яті, та пристроями вводу/виводу; можлива також мультишинна організація;
3) комутатор (*crossbar switch*) – пристрій для (перехресного) з′єднання процесорів та модулів пам′яті. Оскільки складність n^2 не дає змогу застосовувати цю схему для великої кількість процесорів n, то часто застосовують багаторівневі комунікаційні системи модулів (*multistage networks*) – деревоподібні системи зв′язку, складність яких є величиною порядку $O(n\log_k n)$, де k – кількість вихідних зв′язків модуля;
4) локально-зв′язувальні мережі (*nearest-neibour-mesh*) зв′язують між собою процесори, або процесорні елементи і застосовуються у паралельних системах з великою кількістю обчислювальних вузлів (процесорів); відомими прикладами локально-зв’язувальних мереж є топології локальних мереж (лінійка, кільце, зірка), гратка, тор, гіперкуб та інші.

Системи зв′язку паралельних обчислювальних систем характеризуються *латентністю* (запізненням проходження повідомлень через конфлікт у мережі та необхідність виконання протоколів) і *швидкістю передачі*, що залежить від пропускної здатності мережі. Для зменшнння

величини латентності використовується кеш-пам′ять. В останні роки для забезпечення високопродуктивних систем зв′язку у паралельних та розподілених системах, зокрема, кластерних, замість простих схем і топологій застосовуються комунікаційні технології, такі як Fast Ethernet, SCI, Memory Channel, Gigabit Ethernet, Myrinet та інші [8]. В наступній таблиці подані характеристики деяких з таких технологій.

Технологія Виробник	SCI Dolhin	Myrinet Myricom	CLAN Gigant	ServerNet Compaq	Fast Ethernet Intel, 3Com
Латентність (в мксек.)	5,6	17	30	13	170
Пікова пропускна здатність (Мб/с)	460	160	150	?	12,5
Пропускна здатність для MPI (Мб/с)	80	40	100	180	10

Для вирішення питань підвищення місткості та продуктивності систем неосновної пам′яті в паралельних системах застосовуються спеціальні пристрої та технодлогії розпаралелювання вводу/виводу. Одна з них технологія RAID (Redundent Array of Independent Disks) [9]. Основна ідея полягає у розподілі даних одного файла по декільком дискам та паралельному виконанні операцій обміну з цими дисками для підвищення швидкості. Крім того RAID-системи здатні підвищувати надійність системи зберігання даних за рахунок дублювання наборів одних і тих же даних на різних дисках.

Для складання оцінок продуктивності ПОС з 1988 р. існує міжнародна організація SPEC (Standard Performance Evalution Corporation) www.spec.org , що складає тестові пакети для оцінки продуктивності мікропроцесорів комп′ютерів та системною програмного забезпечення. Оцінки подаються у відносних одиницях SPECint та SPECfp, які є нормалізованими показниками по відношенню до Ultra SPARC II 300MHz.

1.3. Огляд архітектур сучасних високопродуктивних паралельних обчислювальних систем

Якого рівня продуктивність потрібна сьогодні на мультипроцесорних системах, скільки процесорів можуть налічувати у своєму складі мультипроцесорні системи і які галузі є основними споживачими

високопродуктивних обчислень, можна оцінити хоча б з наступних прикладів:

Комп′ютерне моделювання задач газо- і гідродинаміки характеризується великими обсягами обчислень. Наприклад, гідродинамічні процеси при видобутку нафти, за умови розгляду області моделювання протягом 10^3 кроків моделювання і обсягом 10^6 дискретних точок, в кожній з яких оцінюється значення 10 функцій, кожна з яких вимагає порядку 10^3 операцій, матимуть обчислювальну складність порядка $T=10^3 *10^6 *10^3 *10=10^{13}$ операцій з плаваючою точкою. З цього можна зробити висновок, що виконання цього завдання у "реальному часі" за 10 сек вимагатиме продуктивності паралельної системи порядка 1Тфлопс. Така продуктивність досяжна на сьогодні тільки для мультипроцесорних систем, що складаються з тисяч процесорів.

Інший приклад – менш чисельної за складом мультипроцерної системи. Для обслуговування клієнтів системи резервування авіаквитків Amadeus у США з 180 000 терміналів і 60 млн. запитів за добу за умови, щоб одна операція резервування квитка виконувалась у середньому кільканадцять секунд (оцінки 2000 року), знадобилось два сервери від Hewlett-Paccard по 12 процесорів у кожного [4].

На початку 90-х Національний науковий фонд (NSF) у США опублікував перелік "великих проблем" (Grand Challenges), що потребують надвисоких обчислювальних потужностей. Цей перелік зокрема включав:

- глобальний прогноз погоди та змін клімату;
- розшифрування геному людини;
- розрахунок турбулентності в моделюванні газо- і гідродинаміки;
- задачі динаміки автомобіля;
- моделювання океанських течій;
- візуальна динаміка потоків;
- моделювання надпровідності;
- квантова хромодинаміка;
- обробка зображень.

У середині 90-х з розвитком Internet до них додалися нові проблеми, пов′язані з мережними технологіями, а саме: цифрові бібліотеки, електронна комерція та інші. Просування в розв′язанні цих та інших "великих" задач були значними. Особливо вражаючим був результат комп′ютерного моделювання геному людини (2000 р.). Виявилось, зокрема, що людський організм включає не 120 тис. різних генів (ДНК, що містять закодовану інформацію про "програму" розвитку людського організму). як передбачали біологи, а значно менше – порядка 30 тис.

На сьогодні найбільш розповсюдженими архітектурами паралельних обчислювальних систем є:

1) векторно-конвейерні;
2) симетричної мультиобробки (SMP);
3) масивно паралельні;
4) кластерні;
5) метакомп′ютерні.

Ці класи архітектур у “чистому” вигляді зазвичай не використовуються і існують лише у різних комбінаціях між собою, до яких вдаються виробники мультипроцесорних систем.

Векторно конвейерні системи

Яскравим представником цього класу архітетктур є мультипроцесорні системи фірми Cray. Під керівництвом засновника компанії С. Крея у 1976 р. було розроблено першу з серії машин Cray-1, з якою пов’язано походження терміну “суперкомп’ютер” Як не дивно, це була однопроцесорна машина, але досить складної побудови. ЇЇ спеціально сконструйований процесор мав тактову частоту 12.5 нс. і налічував 12 конвейерних функціональних пристроїв. Пристрій оперативної пам’яті мав час доступу 50 нс. і являв собою 10^6 64 розрядних слів, поділених на 16 банків. Cray-1 демонстрував середню продуктивність 130 Мфлопс при піковій потужності 160 Мфлопс, що було абсолютним рекордом на той час серед всіх ЕОМ, включно з мультипроцесорними. Принциповим моментом архітетктури Cray-1 були 8 векторних регістрів по 64 64-розрядних слів для зберігання чисел з плаваючою точкою (ЗПТ). Швидкість обміну з пристроєм оперативної пам’яті (ОЗП) складав 320 Мслів/сек.

У 80-ті роки фірма розширила цю архітектуру до мультипроцесорної з спільною пам′ятью в системах Cray X-MP та Cray Y-MP. Але головні архітектурні ідеї для високої продуктивності залишилися тими ж:

- розшарована память;
- швидкий процесор (9,5 нс. тактова частота);
- велика кількість регістрів скалерних;
- 12 функціональних пристроїв, які складають чотири групи – адресні, скалярні, векторні, обробки ЗПТ, і працюють паралельно.

У 90-і роки модель Cray-YMP C90 вже налічує 16 процесорів (з тактовою частотою 4,1 нс.), де паралелізм обчислень досягається на чотирьох рівнях. Перший рівень – конвейерезація виконань операцій. Всі операції виконання команд: звернення до основної пам′яті, обробка адрес, власне виконання команди – є конвейерними за рахунок буферних регістрів між ОЗП і функціональними пристроями.

На другому рівні паралелізм реалізується паралельними обчисленнями на різних функціональних пристроях. Це дозволяє виконувати розпаралелювання обчислень виразів на зразок: x=a*(в+c)/(d+e). Як очевидно з аналізу інформаційної залежності, підвирази a*(в+c) і (d+e) можуть обчислюватись незалежно, а значить, при наявності додаткового обладнання, і паралельно.

Третій рівень складає векторна обробка масивів. Саме на цьому півні забезечується найбільше прискорення обчислень – час виконання обробки може бути на порядок меншим, ніж для відповідних послідовних операцій. Виграш одержується за рахунок групування даних і операцій. Крім того, векторні операції, що використовують різні функціональні пристрої – теж виконуються паралельно. А ще до того ж викоритовується “зачеплення”

регістрів векторних операцій, тобто організація зв'язків між векторними операціями за входом і виходом всередині процесора на рівні регістрів, без виходу в оперативну пам'ять, причому з довільною глибиною такого зачеплення (внутрішній конвеєр). Наприклад, за такою схемою можуть реалізуватись операції читання, додавання, множення векторів, запис векторів під час виконання векторних і матричних обчислень..

Четвертий рівень – це паралелізм мультипроцесорної обробки.

Подальший розвиток архітектури Gray пішов по шляху систем MPP – масивно паралельних архітектур з розподіленою пам′яттю Cray, такі як T3D і Cray T3E. Сучасні машини Gray-T3D можуть налічувати від 32 до 2048 обчислювальних вузлів. Кожний вузол містить по два процесори Alpha 150MHz, кожен з 8 М слів ОЗП, та контроллера мережних передач. ОЗП вузла утворює "розподілену спільну" пам'ять (DSM, distributed shared memory), тобто фізично розподілену але логічно спільну пам′ять Звернення до "чужої" пам′яті в 6 разів повільніше. Швидкість обміну між вузлами 140 Мб/сек. Комунікаційним середовищем Gray T3D є тривимірний тор (кожний вузол має 6 сусідів). Пам′ять системи – розподілена. T3D має хост-машину (Cray C90), де виконується підготовка і компіляція програми. Швидкість зв′язку з хост-машиною 200 Мб/сек.

Для T3D використовуються ті ж рівні паралелізму, що і для C90. Конвейерний паралелізм забезпечується мікропроцесорами DEC Alpha-RISC. Використання серійного мікропроцесора Alpha значно здешевило розробку. При розпаралелювання задач кожній з них виділяється область вузлів, що утворює паралелепіпед. Основу програмного забезпечення складають транслятор з Фортрана CFT, ассемблер і операційна система, що забезпечує мультизадачність і синхронізацію паралельних процесів над спільною пам′яттю.

Використання векторно-конвейерного паралелізму потребує значних зусиль. Перші компілятори для Gray розпаралелювали тільки внутрішні цикли програм. Решту перетворень доводилося проводити вручну. При цьому такі можуть зустрічатися різні випадки. Циклічні оператори з несладною структурою залежності в тілі циклу, як наприклад,

```
DO 5 I = 1,n
      A(I) =  B(I) + C(I)
5 CONTINUE
```

векторизуються легко, бо в даному випадку має місце інформаційна незалежність ітерацій циклу. При векторизації циклу виграш у щвидкості виконання циклічного оператора одержується за рахунок: 1) виконання однотипної операції декодування в тілі циклу тільки один раз, а не щоразу для кожної ітерації; 2) завантаження порціями; 3) використання регістрів; 4) "зчеплення" регістрів для уникнення виходу проміжних результатів за межі процесора.

З іншого боку, існують набагато складніші випадки, як наприклад,

```
DO 6 I = 1,n
      A(I) = A(I-1) + S
6 CONTINUE,
```

в якому кожна з наступних ітерацій циклу не може початись доти, поки не завершена попередня. Тож такі цикли не векторизується.

Паралельні системи з архітектурою симетричної мультиобробки (SMP)

Як вже зазначалося вище, сучасна практика розробки архітектур паралельних систем налічує декілька основних класів таких архітектур: векторно-конвейерна; симетричної мультиобробки; масового паралелізму; кластерна та метакомп′ютерна.

Ахітектура симетричної мультиобробки SMP означає, перш за все, однорідність ресурсів паралельної системи (процесори, модулі пам′яті, засоби зв′язку), що має на меті спростити модель програмування і поліпшити розробку паралельних алгоритмів. Проте з іншого боку, підтримка принципу однорідності веде до обмеження маштабованості таких систем.

Для забезпечення однорідного доступу процесорів до спільної пам′яті використовуються жорсткі кабельні з′єднання (наприклад, Craylink – для суперкомпютерів фірми Cray), а також спільні шини та комутатори. Класичним прикладом шинної архітектури є мультипроцесорні системи Sequent Balance Series, що були популярними у 80-90 роках. Так, наприклад, Sequent Balance 2100 мав:

від 2 до 32 процесорів,

до 28 Мбайт оперативної пам′яті,

шину 26,7Мбайт/с (для порівняння: сучані шини мають пропускну здатність порядка 1GБ/с, такі як Sun Gigaplane або SGI Power Path),

операційну систему типу UNIX (DYNIX),

параллельний компілятор з Фортрана 77.

Це типовий приклад “малопроцесорних” мультипроцесорних систем, маштабованість яких обмежена можливостями спільної шини.

Залежність мультипроценсорного прискорення s від кількості процесорів p в таких системах є нелінійною. Діапазон кількості процесоів (1, p_0), на якому залежність $s(p)$ залишається (майже) лінійною, називається інтервалом масштабованості системи.

Сучасним прикладом архітектури симетричної мультиобробки є системи Onyx від Silicon Graphics з кількістю процесорів від 1 до 8 і шиною пропускної здатності 1,2 Гб/сек.

Як зберегти базову архітектуру SMP, разом з тим розширити можливості масштабування архітектур? Свій шлях запропонували фірми HP та Convex після їх злиття в архітектурі Exemplar, що належить до класу SPP-2000 (Scalable Parallel Processing). В основі їх підходу – дворівнева організація пам′яті з комутатором на нижньому рівні і торовидною мережею зв′язку на верхньому. Нижче на рис. 1.3. подано нижній (базовий) рівень цієї архітектури.

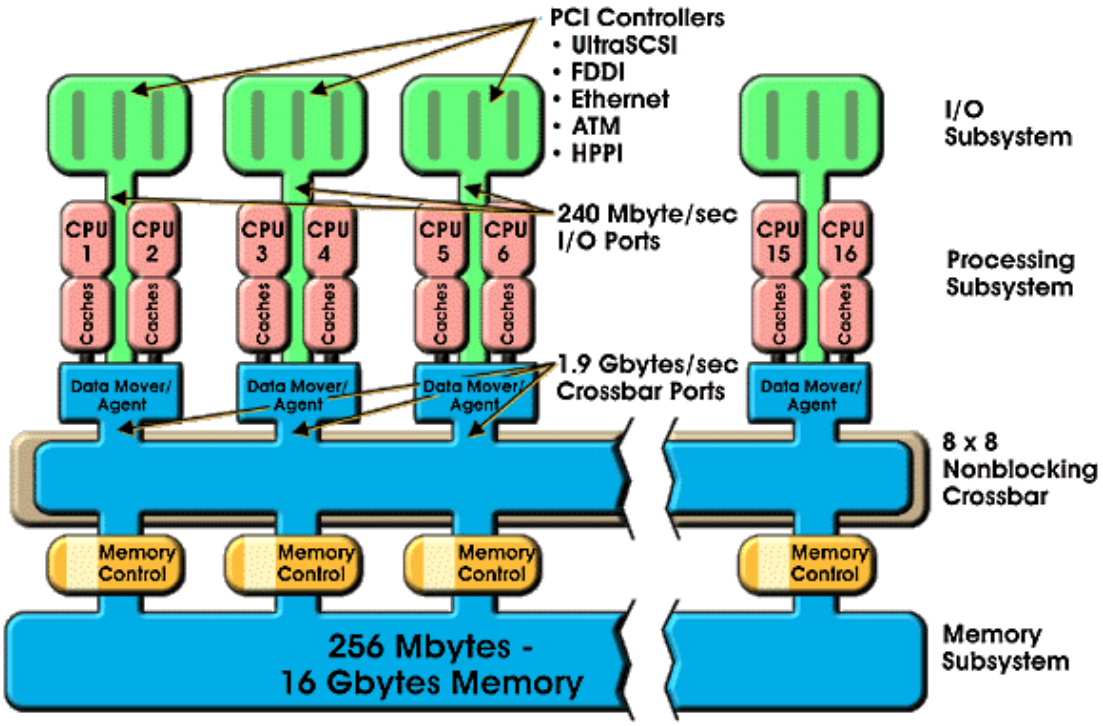


Рис. 1.3. Базовий рівень архітектури SPP-2000

Сучасна версія SPP-2000 Exemplar існує в двух варіантах S-клас та X-клас, що відповідає, відповідно, однокластерному та мультикластерному варіанту серверного вузла. SPP-2000 Exemplar являє собою архітектуру MIMD з кількістю процесорів від 4 до 64 зі спільною пам′яттю. Система класу S в кінці 90-х років мала такі показники:

– 64 розрядні процесори PA-RISC (4-16) з сумарною піковою потужністю до 11,5 Гфлопс,

– обсяг фізичної пам′яті: від 256 Мб до 16 Гб.

– полоса пропускання комутатора 960 Мб/сек *8*2=15,36 Гб/сек

– максисальна пропускна здатність каналів вводу/виводу 1,9 Гб/сек.

– розшарована пам′ять на: 8 контролерів * 4 модулів = 32 шарів.

– модуль керування даними Data Mover забезпечує властивість віртуально спільної пам′яті.

Іншими сучасними прикладами архітетктури симетричної мультиобробки є Seguent NUMA-Q, SGI Origin, SGI Onyx2 [9].

Для того, щоб програми користувачів могли сповна використовувати можливості, що мають місце в апаратному забезпеченні, потрібна відповідна операційна система, орієнтована на симетричну мультиобробку. Основні характеристики такої SMP-ОС такі:

1) – мультизадачність (multitasking).
2) – мультипоточність (multithreading).
3) – мультипроцесорність
4) – стандартизація.

Мультизадачність – здатність підтримувати одночасно декілька програм користувача. На відміну від простих ОС, де перемикання від задачі

до задачі керується самими задачами (наприклад, як у Windows 9x), в SMP-ОС реалізується система переривання. Найважливішою ознакою задачі є виділення для неї власного адресного простору у пам′яті, де існує процес виконання задач (кооперативна для Windous 9x, витісняюча – для Windous NT).

Потік, на відміну від процесу, власного адресного простору не має, і кілька потоків можуть співіснувати в рамках одного процесу (задачі), ділячи між собоюресурси цієї задачі. Потік є легким (light-weighted) процесом і має високу ефективність, що використовується в ОС для виконання швидкоплинних коротких завдань, наприклад, обслуговування з′єднання сервера з клієнтом.

Мультипроцесорність SMP-ОС зокрема означає, що будь-який процес чи потік може виконуватись на будь-якому процесорі. Це додає SMP системам гнучкості та ефективності, а також покращує їх стійкість до помилок та пошкоджень при виконанні паралельних програм. SMP – системи є сильнозв′язаними мультипроцесорними системами з рівними правами доступу процесорів до ресурсів системи. Зокрема це стосується і ядра ОС, що існує в одному примірнику в спільній пам′яті системи. SMP-ОС від розробників програмного забезпечення є стандартними, тобто такими, які можуть виконуватись на багатьох мультипроцесорних платформах. Прикладами таких ОС є Windous NT, SCO Open Server, UnixWare, NetWare SMP.

Застосування. Платформи SMP – найпоширеніша форма застосування паралельних систем у різних напрямках. Серед них – сервери застосувань, критичні до навантажень, що можуть задовольняти властивість масштабованості.

Архітектура масово-паралельних систем

Масово-паралельна обробка (Massively Parallel Processing – MPP) виникає в зв′язку з такими трьома тенденціями в паралельних обчисленнях:

1) досягнення надвисокої продуктивності;
2) надання мультипроцесорній системі властивості масштабованості – тобто лінійної залежності продуктивності від кількості процесорів; до того ж це надає їм гнучкості у виборі потрібної конфігурації та ціни;
3) здешевлення обчислень.

Неформально MPP вважають такою при кількості процесорів $p \geq 100$. Характерною ознакою системи основної пам′яті для MPP є її розподіленість (хоча б фізична) і різноманітність система комунікацій, що оптимізує кількість кроків передач між процесорами. Нижче розглядаються три приклади організації систем масового паралелізму, від Cray та IBM, що по-різному вирішують проблему досягнення високої продуктивності обчислень.

CRAY T3D.

Система з’явилась усередині 90-х років. CRAY T3D має від 32 до 2048 процесорів Alpha 21264, що зв′язані через комунікаційну мережу у вигляді тривимірного тора. Архітектурними елементами CRAY T3D є обчислювальні вузли, що кожен складається з 2-х процесорів, комунікаційна мережа та вузли вводу/виводу. Процесори 150Mtz, 64 розрядні RISC-архітектури, кожен має 8Мслів = 64 Мб локальної пам′яті і потужність 300 Мфлопс. Звертання до “чужої” пам′яті процесора є у 6 разів повільнішим від звертання до власної. До складу обчислювального вузла входять також контроллер асинхронного доступу до локальної пам′яті без переривань процесора, та мережний інтерфейс для формування посилок передач чер комунікаційну мережу.

Комунікаційна мережа являє собою три-вимірну гратку зі швидкістю обміну 140 Мб/сек, що має такі переваги невеликої кількості зв′язків при взаємодії різних процесорів (при 128 процесорах – 6 кроків, для 2048 – 12), а також можливість вибору іншого маршруту замість пошкодженого.

IBM SP2

На відміну від Cray T3D де реалізована архітектура розподіленої спільної пам′яті, в системі IBM SP2 (Scalable Processing) використаний інший підхід – архітектури розподіленої пам′яті з передачею повідомлень. Такий вибір диктувався гнучкістю архітектури, до якої прагнули розробники.

Система SP2 може мати від 2 до 512 процесорних вузлів, побудованих на базі процесора Power2 RS/6000 з локальною пам′яттю 64 Мб.
Треба відзначити використання серійного процесора RS/6000 для робочих станцій, що дало змогу використати для SP2 кілька тисяч застосувань без перепрограмування. Кожний процесор працює під управлінням власної копії операційної системи AIX.

В SP2 підтримуються два основних типи вузлів: обчислювальні та серверні. IBM розробила гнучку конфігурацію вузлів в залежності від потреб користувача і замовника. Найбільш уживаними є конфігурації P2SC Thin, P2SC Wide і SMP High. Останній варіант – для серверного вузла на 2, 4, 6, і 8 процесорів. Вузли Thin і Wide є однопроцесорними і відрізняються показниками обсягів пам′яті та продуктивності.
При тактовій частоті усього 99,7 МГц пікова продуктивність процесора Power2 RS/6000 складає 260 Мфлопс, а “широкі” вузли типу Wide можуть мати до 2 Гб оперативної пам′яті та 2,1 Гб/сек пропускної здатності.

Вузли SP2 зв′язані між собою швидкісним комутатором, щоб обмежити малопродуктивні інтерфейси операційної системи AIX. Комутатор побудований на принципах комутації пакетів, являє собою багатокаскадну, з обхідними шляхами комутаційну мережу, що забезпечує топологію зв′язку “кожний з кожним”. Наслідком такої конструкції є лінійне зростання пропускної здатності комутатора з ростом розмірів системи, а також постійна пропускна здатність каналу зв′язку між вузлами незалежно від їх розташування. Кожний кінець двостороннього каналу зв′язку забезпечує 40 Мб/сек. В результаті латентність зв′язку між підзадачами складає 40мкс (в

режимі UDP/IP ~ 300 мкс), а пропускна здатність в цих двох режимах, вдповідно, 35 Мб/сек і 10 Мб/сек.

Intel ASCI Red

Для досягнення найвищих показників паралельних систем для вирішення задач оборонної тематики міністерством енергетики США в 1995 р. була ініційована програма ASCI (Advanced Scientific Computing Intitretive) з 900 млн. фінансування на побудову масово-парарельних систем. Першою з них стала ASCI Intel Red Supercomputer, що в 1996 р. вперше перевищила рубіж середньої продуктивності 1 Терафлопс (1,06 Тфлопс). Система мала 9216 процесорів Intel Pentium Pro, організований у двопроцесорні вузли, що з′єднуються через три-вимірну сітку зв′язку 38*32*2. Система мала сумарну оперативну память 596 Гб, дві дискові дистели Raid по 1Гб; пікову потужність вузла 400 Мфлопс і 1,8 Тфлопс всієї системи.

Два процесори вузла мали спільну пам′ять і працювали при виконанні паралельних завдань таким чином, що другий процесор обробляв паралельні потоки першого процесора або був його комунікаційним співпроцесором, або зовсім не використовувався. Програмне забезпечення включало операційну систему UNIX, компілятор з паралельної версії Фортрана HPF та бібліотеку MPI. Час напрацювання на віднову одного вузла складав більше 50 годин, а час відновлення з нейтральної точки всієї системи близько 5хв. Система проектувалась так, щоб забезпечити безперебійну роботу понад чотири тижні, тому на ній можна було розв’язувати масштабні науково-технічні задачі з широкого кола застосувань.

Архітектура кластерних систем

Сучасні суперкомп′ютери розвивають потужність понад 1Тфлопс, проте це дуже дорогі обчислювальні системи. Ось деякі дані про співвідношення ціна/продуктивність для кількох виробників (www.osmag.ru, # 5-6 2000).

Виробник	Архітектура	Потужність (Гфлопс)	Співвідношення ціна/продуктивність ($тис./Гфлопс)
Compaq	84-вузловий кластер Beowulf	74	6,4
Hewleot-Paccard	HP 9000 server, class L	63	7,9
IBM	SP	38	6,3

Отже сучасний рівень для співвідношення ціна/продуктивність складає величину порядку $10 тис./Гфлопс. а потужний суперкомп′ютер з швидкодією 1 Тфлопс мусить коштувати десятки мілйонів доларів. Це унікальні установки, які доступні тільки небагатьом організаціям та

корпораціям. Проте, потреба у високопродуктивних засобах обчислень зростає. Вихід із ситуації полягає у зменшенняі ціни 1Гфлопс за рахунок архітектурних рішень в комунікаційних технологіях і, як наслідок, появі обчислювальних кластерів. На відміну від серверних кластерів, що обслуговують бази даних і Web-вузли і де головною метою є збереження надійності сервера, метою створення обчислювальних кластерів є одержання високої продуктивності за невисоку ціну. Таким чином кластерна технологія є продовженням тенденції MPP у напрямку здешевлення високопродуктивних паралельних обчислень.

Обчислювальний кластер – це об′єднання обчислювальних вузлів швидкісною мережею для розв′язання трудомістких задач.

Вузлами можуть бути однопрцесорні машини, або 2-6-прцесорні SMP-платформи, кожна з яких працює під управлінням власної копії мережної ОС (часто Linux). Історично першим кластерим процесором став Beowulf (1994р.) у центрі NASA в США. Це був 16-процесорниий кластер на процесорах Intel 486 PX4/100МГц з 16 Мб. пам′яті кожний, трьома мережними Ethernet-адаптерами та спеціальними драйверами, що розподіляють трафік між мережними картами.

Великого успіху досягли розробники проекту Avalon (1998р. Лос-Аламоська національна лабораторія). Це був Linux-кластер на 70 (потім 140) процесорів Alpha 21164 з тактовою частотою 533 МГц. В кожному вузлі – по 256 Мб пам′яті, жорсткий диск 3 Гб і мережний адаптер Fast Ethernet. Вартість системи складала 313 тис. доларів, а середня швидкодія 47,7 Гфлопс. Із цими показниками проект зайняв 114 місце в 12-ому списку TOP500 і одержав у 1998р. премію Г.Белла за показником ціна/продуктивність.

Одне з останніх досягнень (рекорд 2002 р.) у співвідношенні ціні/продуктивність одержано в проекті KLAT2 (Kentukky Linux Athlon Testbed2). Кластерна система мала 64 вузли з процесором Athlon/700 від AMD і по 128 Мб. пам′яті в кожному, розподілений комутатор Flat Nighbourhood Netwok (ENN), 32 портових комутаторів (маршрутизаторів), що завжди забезпечують зв′язки “кожний – з кожним”. Показник ціна/продуктивність склав усього $650/Гфлопс.

Невідємною частиною кластера є спеціальне програмне забезпечення, яке крім задач високої продуктивності та надійності виконує низку спеціальних функцій, характерних саме для кластерних систем:

– резервне копіювання інформації та відновлення після аварії;

– перезапуск застосувань аварійного вузла;

– паралельний доступ до спільної дискової пам′яті та інше.

Щодо відновлення після аварії та перезапуску застосувань, то кластери можуть мати декілька сценаріїв такого відновлення, а також засоби адміністрування таких сценаріїв, як для самого вузла, так і для кластера в цілому. Ці функції запускаються автоматично при аварії або адміністратором при виведенні вузла з конфігурації.

Спільна дискова пам′ять, як правило, організовується на масивах дисків RAID. Для доступу вузлів до спільної файлової системи використовується механізм розподіленого керування блокуванням (DLM – Distributed Lock Manager), який закриває доступ до пристрою чи файла для всіх вузлів, крім одного, який в даний момент модифікує дані. Зауважимо, що це слабкіша вимога, ніж при класичному взаємному виключення, оскільки допускає паралельне читання однієї одиниці даних з боку декількох процесів. В системах без DLM в кожний момент часу тільки один вузол має право до спільної дискової пам′яті. Це вирішується або апаратно (наприклад, засобами спільної шини), або програмою – засобами операційної системи.

Для підвищення продуктивності та готовності кластерних систем, а також як засіб, альтернативний спільній дисковій пам′яті, часто застосовується механізм віддзеркалення інформації – тобто копіювання даних одного вузла на дисковий накопичувач іншого. Крім обміну даними віддзеркаленої інформації для підтримання нормальної дієздатності кластера вузли обмінюються також деякою динамічною інформацією про зміни в конфігурації вузлів та даними про спільні накопичувачі дискової пам′яті.

Важливе значення для продуктивності та інших характеристик кластеру має комунікаційне середовище, що з′єднує кластерні вузли у єдине ціле. Вимоги до комунікаційного каналу залежать від ступеня інтегрованості вузлів та характеру паралельних прикладних програм, що виконуються на кластері. Наприклад, якщо вузли не ділять між собою дисковий простір і обмінюються тільки контрольними повідомленнями, то такий тип обміну не потребує значних ресурсів і може бути реалізований засобами звичайної Ethernet 10 Мбіт/сек. В інших випадках потрібні більш високопродуктивні комунікаційні технології.

Основними показниками комунікаційних технологій, як зазначалось вище, є два: 1) латентність, тобто час затримки, пов′язаний з неохідністю подолання шарів протоколу та апаратури до моменту виходу за межі вузла; 2) пропускна здатність. Прикладом такої технології є SCI (Scalable Coherent Interface), що застосовується в мультипроцесорних кластерних платформах [9]. Основа технології – швидкісні кільця складені з однонаправлених лінків з піковою пропускною здатністю 400 Мб/сек. Реальні показники продуктивності: латентність 5,6мкс, пропускна здатність 80 МБ/сек (на тестових задачах Linpack). Підтримуються ОС Linux, Windows NT і Solaris, кількість вузлів – до 100. Найбільша відома установка – 96 процесорів. Приклад технології SCI на 16 однопроцесорних вузлів – двовимірний TOP.

Метакомп’ютерні системи

Метакомп′ютерні системи, що розвиваються з початку 90-х років, за основну мету мають об′єднання різнорідних обчислювальних ресурсів та їх оптимальному використанні. До складу метакомп′ютерної паралельної системи можуть входити, наприклад, робочі чи графічні станції – для підготовки та візуалізації даних та результатів, серверні вузли – для забезпечення доступу до баз даних чи Internet, та цілі векторно-конвейерні чи

MPP-системи –для виконання паралельних обчислень, що потребують великої потужності. Отже, характерною особливістю метакомп′ютерних систем є їх спеціалізація, що може давати, в порівнянні з однорідними паралельними системами, значно більший вигрш у ефективності обчислень, а також колективне використання ресурсів, що можуть використовуватись у глобальному масштабі.

На сучасному етапі метакомп′ютерні системи розвиваються у напрямку динамічних мережних утворень (аж до глобальних) з змінною конфігурацією та асинхронною роботою компонент, націленої на виконання задач зі слабозв′язаними частинами, наприклад переборного чи пошукового типу. Метакомп′ютерна система забезпечує прозорий контрольований доступ користувачів до системи через Internet та прозоре підключення не використовуваних (idle) обчислювальних ресурсів до мета-комп′ютера.

Відомими прикладами систем є Globus і Legion [9]. Одним з перших прикладів задач, для розв'язування яких знадобились обчислювальні ресурси у глобальному масштабі стала RSA Challenges – розшифровка кодів секретності. Поточна версія задачі з розшифровки тексту, закодованого 64-бітним ключем має справу з 2^{64} комбінаціями. У проекті залучено понад 200 тис. учасників; досягнута продуктивність 75 млрд.ключів/ сек. В іншому проекті SETI@home (Search for Exraterrestial Intelligence) – пошук позаземних цивілізацій – зареєстровано близько 1 млн. учасників.

Серед прикладів подібних масштабних проектів – проект Оксфордського університету та фірми Intel (http:/www.intel.com/cure) зі створенню Інтернет-суперкомпютера IPPPP (Intel Philantropie Peer-to-Peer Program) для пошуку нових ліків від лейкемії то ракових захворювань. Суть проблеми полягає у переборі сотень мільйонів хімічних сполук і оцінці їх здатності протистояти захворюванню. Складність задачі оцінюється близько 24 млн. годин на одному комп'ютері. Розглядаються формули білків, блокування яких веде до створення препарату. Передбачається об′єднання мільйонів ПК в мережу, продуктивність якої оцінюється близько 50 Тфлопс.

1.4. Тенденції розвитку програмного забезпечення паралельних обчислювальних систем

Системи паралельних обчислень є багаторівневими системами. Традиційно виділяють три головні рівні: аппаратне забезпечення, системне програмне забезпечення (ПЗ) і прикладне ПЗ. В останній час між системним і прикладним ПЗ виділяють ще й проміжне ПЗ (middleware), а в його складі – системне проміжне ПЗ, що складається з ядра ОС та системних сервісів. Ядро реалізує функції нижнього рівня, що є базовими (в цілому – це управління ресурсами). Системні сервіси – це абстрактне бачення операційної системи з боку прикладної програми (абстрактна машина). Пара <менеджер ресурсів, абстрактна машина> власне і визначає те, що називається операційним середовищем. Докладніше про структуру і функції

операційних середовищ йдеться в інших розділах цієї книжки. Підкреслимо тільки, що головними напрямами розвитку програмного забезпечення паралельних обчислювальних систем є тенденції до розподіленості і децентралізації ПЗ.

Історично, системи паралельного ПЗ пройшли декілька етапів у своєму розвитку, які розрізняються спупенем зв′язаності (coupling) підсистем, а отже ступенем її централізації чи децентралізації. Розрізняють апаратну і програмну зв′язаність. В апаратному розумінні паралельна система є тіснозв′язаною (tightly-coupled), якщо час міжпроцесорного зв′язку близький до часу міжпроцесного зв′язку вседині одного процесора. В протилежному випадку – система є слабкозв′язаною (loosely-conpled). Наприклад, шинна архітектура порівняно з локальною мережою є тіснозв′язаною у апаратному відношенні. У програмному розумінні паралельна система називається тіснозв′язаною, якщо її програмне забезпечення централізоване і використовує для управління глобальну інформацію; в протилежному випадку система є слабкозв′язана у програмному розумінні.

Хронологічно першим у розвитку паралельного ПЗ був етап централізованих систем (ЦС), що були тісноз′язаними у апаратному і програмному розумінні. За хронологією другим виник етап паралельних операційних середовищ у мережах – мережних систем (МС), що характеризуються слабкозв′язаністю у обох розуміннях. На подальших етапах виникали розподілені системи (РС)– слабкозв′язані в апаратному і тісно зв′язані в програмному розумінні, і нарешті, кооперативно-автономні системи (КАС)– що характеризуються послабленням централізованості розподілених систем. Отже ЦС → РС → КАС → МС – логічна лінія розвитку паралельних систем у напрямку їх децентралізації.

Змістовне порівняння цих етапів подане в наступній таблиці:

Етап	Систе-ма	Характеристика	Основна мета і розвинута абстракція
1	ЦС	управління процесорами, пам′ятю,В/В, файлами	Менеджмент ресурсами, Абстрактна машина. (Віртуальність)
2	МС	відалений доступ, інформаційний обмін	Спільне використання ресурсів, (Інтероперабельність)
3	РС	глобальні: ▪ файлова система; ▪ простір імен; ▪ час; безпека; ▪ обчислювальна потужність.	Подання мультикомп′ютерної системи Як єдиної машині. (Прозорість)

4	КАС	відкриті та кооперативні розподілені застосування	Кооперативна робота (Автономність)

Розвинуту абстракцію подано як кінцеву мету для даного класу систем (в дужках).

Інтероперабільність – це здатність забезпечення інформаційного обміну між гетерогенними компонентами системи,– найперша умова для мережної системи.

Віртуальність – ідеал для ЦС як продовження апаратного її забезпечення. Означає можливість користувачеві бачити те, що він хотів би бачити (наприклад, віртуальну машину або віртуальну пам′ять).

Прозорість – схожа до віртуальності, проте означає можливість користувачеві не бачити тих відмінностей, що він не хотів би бачити. Обидві властивості є абстракціями паралельних систем, але кожна по різному подає багаторівневе уявлення про неї. Прозорість є ключовим поняттям розподілених систем, що забезпечує логічний опис системи незалежно від її фізичного рівня.

Автономність – мета кооперативної роботи, користувачів, що більше нагадує поведінку в людському суспільстві і притаманне відкритим системам. Часто така властивість суперечить прозорості, оскільки остання передбачає деяку централізацію контролю. В КАС користувач може створювати своє влосне бачення системи, що може не співпадати з “прозорим “, але єдиним баченням РС.

Загалом, сучасні системи програмного забезпечення і операційні середовища є інтегрованими, в яких перераховані властивості, всі можуть бути бажаними. Термін “розподілені системи” може вживається як узагальнення всіх видів систем, крім першого, що розглянуті вище.

Центалізовані системи використовується на тіснозв′язаних архітектурах, прикладами яких є симетрично-паралельні, векторно-конвейерні та частково кластерні архітектури. ЦС являє собою великий програмний комплекс, який звичайно структурований по вертикалі і горизонталі. Вертикальне структурування – це розбиття на програмні шари, що допускають взаємодію між сусідніми шарами по принципу структурного програмування, тобто один вхід–один вихід.

Побудову ЦС схематично можна подати у вигляді наступних шарів:

Застосуван-ня	…	текстовий редактор	Облік системних ресурсів
Підсистеми	середовище програмування		СУБД

Утіліти	Компіля-тор	інтерпритатор команд	бібліотеки
Системні сервіси	файлова система	менеджер пам′яті	планувальник
Ядро ОС	мультиплексор, обробник переривань, драйвери пристроїв, примітиви синхронізації		

У горизонтальному напрямку відбувається структурування шару на модулі, що кожний виконує певну сервісну функцію. В об′єктно-орієнтованих системах модулі реалізуються як об′єкти з визначеними на них операціями для виконання сервісів. При цьому всі ресурси, включно з файлами і процесорами, уніформно представлені як об′єкти даних, фізична репрезентація яких схована у відповідних структурах даних. Така уніформність спрощує керування доступом та захистом інформації, а також полегшує перенесення систем на інші апаратні платформи.

Ядро ОС є монолітним програмним утворенням, що реалізує низькорівневі функції управління. Принцип мінімального ядра зменшує до мінімуму машинно-залежний код. Універсальне мінімальне ядро називається мікроядром (microkernel). Архітектуру мікроядра утворюють пластформно-залежне мінімальне ядро та апаратно-незалежні виконавчі модулі (executives) з їх прикладними прграмними інтерфейсами. Значення мікроядра полягає у його універсальності, що дає необхідні і достатні умови для побудови будь-якої операційної системи чи підсистеми з мінімальними затратами. Прикладами відомих мікроядер є IBM Microkernel та Microsoft Windous NT. Мікроядро, на відміну від загального випадку ядра, структурується відповідно до особливостей цільових архітектур. Тому з′являється проміжний шар апаратної абстракції HAC (Hardware Abstraction Layer), що полегшує перенесення (мобільність) коду на різні платформи.

Мережні системи. Найперша мета мережної ОС – усуспільнення ресурсів (програм і даних) мережі. Але оскільки вона є слабкозв′язаною, то єдиним способом взаємодії в мережі є обмін через зовнішній канал зв′язку. Властивість інтероперабельності характеризує гнучкість такого обміну в гетерогенному середовищі, а прикладами механізмів, що підтримують інтероперабельність, стандартні комунікаційні протоколи та загальні інтерфейси до спільних баз даних або файлових систем.

В мережному середовиші обмін інформацією реалізується ієрархічно організованою сукупністю протоколів, з яких головними прошарками є: 1) на апаратному рівні – комунікаційна підсистема; 2) далі вище – транспортна служба; 3) прикладний рівень обміну процесів. Детальніше ця ієрархія описується 7-рівневою моделлю OSI - ISO.

Високорівневими засобами API в мережних системах є сокети або віддалені виклики процедур (RPC). Прикладами мережних функцій є:

– віддалена реєстрація (login), що дозволяє перетворити комп′ютер у термінал; прикладом такої функції є *telnet*;

– передача файлів (наприклад, *ftp*);

– обін повідомленнями (rnessaging) – на відміну від передачі файлів, не потребує встановлення зв′язку у реальному часі (наприклад, електронна пошта (*e-mail*);

– мережева навігація (netwok browsing) – наприклад, як у всесвітній павутині WWW;

– віддалене виконання сервісів, наприклад – MIME (Multipupose Internet Mail Extention) – активне поштова служба; що враховує тип документа; інший приклад – аплети Java.

Розподілені системи. Вони розширюють усуспільнення ресурсів у мережних ОС до розподілених обчислень. Ключова відмінність розподіленої ОС від мережної полягає у наявності у першої властивості прозорості, тобто незалежності від фізичних деталей і обставин реалізації системи. Проявляється у різних аспектах по-різному, тому розкривається пізніше (див. нижче). Розподілена система програмного забезпечення складається з трьох головних компонент: 1) координації розподілених процесів; 2) управління розподіленими ресурсами; 3) реалізація розподілених алгоритмів (протоколів, правил, узгоджень) у відсутності глобальних станів інформації.

Кооперативно-автономні системи. Якщо розподілені системи характеризуються декомпозицією сервісів, то КАС – їх інтеграцією, нагадуючи побудову людського суспільства. Тобто, КАС – це орієнтовані на сервіси програмні системи, де наявним є середній шар ПЗ, що відіграє роль своєрідної програмної шини, що поєднує клієнтів і постачальників серверів. Прикладами є ODP (Open Distributed Processing) і CORBA (Common Object Broker Architecture) архітектури [9].

Таким чином, зазначені дві тенденції у розвитку систем програмного забезпечення паралельних обчислювальних систем, а саме розподіленість і інтегрованість, суттєво доповнюють одна одну.

1.5. Принципи побудови та концепції сучасних систем паралельного програмного забезпечення

Розповсюдження мереж комп′ютерів і, з іншого боку, потреба широкого доступу до інформаційних ресурсів спонукають розвиток паралельних і розподілених систем. Загальними вимогами до побудови паралельних систем програмного забезпечення є :

– ефективність – в порівнянні з послідовними системами, паралельні системи мають непродуктивні витрати на обмін даними, синхронізацію та реалізацію протокола. Крім того, виникає ще проблема оптимізації використання апаратних та програмних компонент, тобто не повинно бути вузьких місць.

– гнучкість – це "налаштованість" системи на користувача, що проявляється у наявності зручного інтерфейсу і відсутності необов′язкових обмежень. З системної точки зору – це можливості модульності, масштабованості, переносимості та інтероперабельності (нефункціональні можливості).

– цілісність (consistency) – через відсутність глобального стану системи, реплікації даних, можливості відмови обладнання підтримка цілісності паралельних систем утруднена. Для користувача ця властивість означає передбачуваність поведінки системи та помилок.

– стійкість до відмов (robustness) – це здатність системи до знаходження цілісних станів (можливість відкатів), знаходження і локалізація аварій, здійснення змін у топології. Стійкість включає також поняття захисту інформації та безпеки користувача.

Принцип прозорості

Принцип прозорості (transperaneg) – означає намагання приховати локально-залежні деталі від користувача і створити образ підтримуваної "чистої" системи паралельних розподілених обчислень. Прозорість забезпечує цілісне логічне сприняття системи і є компромісом між простотою (абстракцією) і ефективністю (використанням знань деталей реалізації). В літературі намічують багато проявів принципу прозорості [9], головними з яких є наступні.

1.Прозорість доступу (acess transporency) означає здатність доступу до локальних і нелокальних об′єктів однаковим способом.

2.Прозорість імен – користувачам не відомо про місце знаходження об′єктів, які розрізняються по іменам. До цього додається також прозорість міграції, коли логічне ім′я об′єкта не змінюється при його переміщенні (незалежність від місця).

3.Прозорість одночасності (concurrency) – означає спільне використання об′єктів (без перекриття) (time-sharing).

4.Прозорість реплікації – цілісність множини копій файлів або даних. Тісно пов′язана з прозорістю одночасності, хоча виділяється окремо.

5.Прозорість паралельності (parallelism) – дозволяє паралельне виконання процесів без відома користувача про те, де, коли і як вони виконуються; паралелізм може мати місце навіть тоді, коли він не специфікується користувачем.

6. Прозорість продуктивності (performance) – передбачуваний рівень ефективності системи навіть при зміні структури системи чи розподілу навантаження. Окремим випадком є плавна деградація продуктивності при частковому виході з ладу обладнання, що називають прозорістю пошкодження (failure).

7.Прозорість розміру (size) – означає можливість нарощування модульності системи (вшир) без зміни уявлення користувача про систему. Аналогічною властивістю є прозорість вглиб (revision), коли зміна у вертикальному нарощуванні рівнів системи невидима для користувача.

Загалом, властивості доступу, імен і міграції є взаємозалежними і разом з властивістю прозорості розміру вшир і вглиб скеровані на забезпечення вимоги гнучкості. Прозорості паралелізму, одночасності і продуктивності скеровані на управління паралелізмом обробки на різних рівнях, відповідно, внутрішньому, міжмодульноиу та міжузловому в паралельній системі, мають на меті досягнення вимоги ефективності системи.

Прозорості реплікації та пошкодження сприяють системній цілісності, а розміру вглиб і вшир – забезпечують плавність при кількісних змінах в апаратурі і програмному забезпеченні (тобто стійкість). Цей список прозоростей не є вичерпним.

Наступна таблиця підсумовує зв'язок між різними видами вимог та концепцією прозорості.

Цілі	Ефективність	Гнучкість	Цілісність	Стійкість
Прозо-рість	Одночасність Паралелізм Продуктив-ність	Доступ імен Міграції розміру	Доступ Реплікації продуктив-ності	Пошкоджен-ня Реплікація розміру

Принцип сервісів (обслуговування)

Система паралельного програмного забезпечення є постачальником сервісів, організованих в ієрархію. Базовим рівнем є системні примітиви, що реалізуються ядром ОС на кожному вузлісистеми. На вищому рівні існують системні сервіси, що реалізуються на системному рівні програмного забезпечення. Іще вище за рівнем знаходяться прикладні сервіси.

До примітивних сервісів належать базові функції ядра ОС, а саме: функції зв′язку, синхронізації та мультиплексації процесів (процесорів). Так, для процесів з розподіленою пам′яттю зв′язок реалізується примітивами send і receive, що мають аргументами логічні канали зв′язку (імена). Зв′язок може бути синхронним або асинхронним. У першому випадку одночасно вирішується питання синхронізації, у другому потрібен сервер синхронізації. Сервер процесів керує створенням, видаленням та обслуговуванням процесів шляхом виділення їм ресурсів пам′яті та процесорного часу. Процесні сервери взаємодіють між собою через віддалену синхронізацію і зв′язок.

Системні сервери – це ті, що знаходяться поза ядром ОС. Перш за все, це сервери імен або каталогів (directories), що реалізують прозорість логічних імен фізичних об′єктів. Трансляція локальних імен об′єктів в шляхи доступу виконується сервером мережі. Функції масової передачі (broadcast, multicast) вимагають відповідних серверів в операційній системі. Ще один важливий системний сервіс – сервер часу, що використовується для синхронізації і планування операцій. Розрізняють фізичний і логічний час. Тайм-сервер фізичного часу обслуговує операції реального часу. Логічний тайм-сервер використовують для ведення порядку подій в розподіленій системі для гарантування коректності операцій.

Сервери імен та часу є прикладами інформаційних сервісів. Натомість існують і ресурсні сервери. Для файл-серверів, що можуть бути ієрархічно структурованими, можуть існувати підсервери каталогів та безпеки. Для серверів процесів важливими функціями є міграція і аутентифікація.

Прикладні сервіси є додатковими (value-added), що корисні в підтримці застосувань. Прикладом є групові сервери, що відповідають за створення, закінчення та проходження групових сервісів, таких як адресна книга, зв′язок, членство, політики доступу, привілеї членів і теке інше. Прикладами таких груп є групи мережних новин, телеконференції, Web.

Механізми на підтримку принципів прозорості та сервісів.

Моделі та схеми іменування. Об′єктами в паралельній комп′ютерній системі є процеси, файли, пам′ять, пристрої, процесори, мережі і т.д.. Об′єкти інуапсулюються в серверах і абстрактно представляються допустими на них операціями. Деталі реалізації сховані принципами прозорості. Ідентифікація серверів відбувається трьома шляхами: а) за іменем; б) за фізичною чи логічною адресою; в) за сервісом, що постачається сервером. Імена – унікальні і більш прозорі ніж адреси, зате адреси можуть нести в собі структурну інформацію про сервери. Прикладом логічної адреси є порти (входи). Один сервер може мати декілька портів і навпаки. Ідентифікація за сервісом – найбільш прозора .

Координація паралельних процесів. Проявляється у двох видах: конкурування при доступі до критичного ресурсу і кооперативна поведінка для узгодження при наявності спільної мети чи завдання. Розрізняють наступні види синхронізації:

а) барєрна синхронізація, коли вся множина процесів мусить прийти в певну точку синхронізації;

б) умовна координація – процес може продовжити обчислення по умові, що задовольняється іншими процесами; окрема проблема є тупики за ресурсами і комунікаційні;

в) взаємне виключення.

Міжпроцесний зв′язок. Для тіснозв′язаних паралельних систем, що мають спільну пам′ять такий зв′язок забезпечується через спільну пам′ять з необхідними засобами синхронізації. Для розподілених систем такими механізмами є обмін повідомленнями і віддалений виклик процедур (або клієнт-серверна взаємодія). Це – механізм парних зв′язків (unicast, або peer–to–peer). Крім того, існують і групові зв′язки: один–всім (broadcast) та multicast (всі –одному).

Розподілені ресурси. Відображення паралельних процесів на множину процесорів вимагає прозорих шляхів до планування паралельних обчислень та стратегії міграції процесів по процесорам. Мета полягає, як правило, у мінімізації часу виконання. Інший бік справи – прозоре зберігання та пересування даних в розподіленій файловій системі або розподіленій спільній пам′яті (distributed shared memory). Тут найважливіші проблеми – спільний доступ та реплікація даних.

Стійкість до відмов та безпека (fault tolerance and security). Масштабність та відкритість паралельних систем є причинами для спеціального розгляду питань стійкості і безпеки. Принцип прозорості тут проявляється у тому, що помилки (свідомо чи несвідомо) є прозорими для користувача (тобто невидимими). Для цього служать такі засоби, як дублювання (redundancy) та відкати (rollback), а також аутентифікація та авторизація.

1.6. Моделі та засоби прогамування паралельних процесів.

Процеси, поряд з файлами, пам′яттю, пристроями, процесорами і т. п. виступають основними об′єктами паралельних систем. Процес виступає як одиниця обчислень в ОС, що має власний адресний простір у основній пам′яті. Процеси можуть бути послідовними (що управляються однією програмою) і паралельними (concurent), що являють собою взаємодіючі послідовні процеси. Різновидом процесу є потоки (threads), що не мають власного адресного простору, але мають власний локальний стан. Потоки можуть породжуватись процесами або іншими потоками, і називаються легкими (light-weigted) процесами. Оскільки операції створення та знищення паралельних потоків є недорогими (за витратами часу і пам′яті), то типовими застосуваннями потоків є обслуговування запитів на серверній стороні (обробник переривань, файл-сервер) або очікування результатів множинних запитів на клієнтській множині. Реалізація потоків відбувається в середовищі процесу користувача або ядра ОС. Останнє стало тенденцією в зв′язку з появою мультипроцесорних платформ для мономашинних систем.

Процеси та потоки (всередині процесів) поєднуються між собою відношеннями синхронізації та обміну. Для відображення відносин синхронізації використовують орієнтовані графи G=(P,S) де P – процеси, а дуги (P_1 P_2) є S, що є направленими, позначають порядок слідування процесів. Неорієнтовані графи G=(P,E) відображають тільки зв′язок між процесами тобто наявність обмінів. Графові моделі є доволі абстрактними і придатні тільки для оцінки продуктивності, наприклад, максимального часу виконання процесів:

Більш інформативними є просторово-часові моделі для відображення синхронізаційних та комунікаційних відносин між процесами на рівні подій. Графові моделі можуть бути одержані безпосередньо з просторово-часових. Про техніку застосування графових моделей можно прочитати в [10,20].

Засоби програмування та мови програмування процесів визначають:

– специфікацію паралельності обчислень процесів;

– синхронізацію процесів;

– комунікації (обмін) процесів.

Часто ці функції поєднуються в одних мовних конструкціях. Загальні категорії таксономії синхронізаційних механізмів включають [9,13,15]:

- системи з спільною пам’яттю:

семафори;

критичні області;
монітори;
маршрутні вирази;
- системи обміну повідомленнями:
синхронні:
симетричні;
асиметричні
асинхронні;
симетричні;
асиметричні

Традиційним шляхом розвитку мов, що використовують згадані засоби, було розширення традиційних послідовних мов програмування. Проте виникали раніше і зараз виникають нові мови, в яких використано нові підходи до вирішення прблем синхронізації, комунікації і паралелізму. Прикладом таким мов у недалекому минулому є, зокрема, Оккам – мова програмування транспьютерів [12], що використала ідеї CSP; конструктори SFQ, PAR, ALT, що розглядають оператори як процеси. Синхронізація – на основі семантики рандеву [19].

Мова Linda [9] з′явилась як "ортогональна" до обчислень модель координації паралельних процесів. В основі її – спільний простір наборів даних (tuples) виду (<tag>, <значення>), де tag є символьним ім′ям списку <значення>, що складається з типізованих значень даних. Над наборами визначені базові операції:

– in(s) – блокуючий прийом з простору наборів TS (tuple space);

– out(s) – неблокуюча передача в просторі TS; де s – деякий набір або шаблон, з яким співставляються набори з TS для виконання операцій вставки та вилучення.

Результатом операції in(s) є вилучення одного з наборів з TS, що задовольняє шаблону s. Результат out(s) – вставка (додавання) нового набору до TS. Є варіант операції in(s) – rd(s), що не вилучає, а лише копіює значення набору з TS в локальну пам′ять процесу.

Отже, LINDA внесла асоціативний пошук – як новий елемент в механізмах синхронізації.

Мови та механізми синхронізації та комунікації, розглянуті досі, були призначені для тісно-зв′язаних систем. Але вони не є адекватними для слабо-зв′язаних систем, де поміж іншого потрібна підтримка надійності комунікацій, реплікації даних, інтероперабельності неоднорідних середовищ і мереж.

В нових умовах глобальних мереж мова Java запропонувала три фундаментальних механізми для виконання розподілених застосувань на мережах:

– стандартний інтерфейс для інтеграції програмних модулей; – це об′єктно орієнтована парадигма програмування, пакети як бібліотеки класів, що можуть бути спільними з прозорим розміщенням у мережі.

– можливість виконання будь-яких Java програмних модулів на будь-якій машині мережі, забезпечується концепцією віртуальної машини для кожної платформи, що інтерпретує байткод програми – проміжний код, що виробляється компілятором Java.

– інфраструктура для транспортування (HTTP) і координації (HTML) програмних модулів (аплетів), що паралельно можуть виконуватись в стандартному браузері.

1.7. Сучасні середовища паралельного програмування

Середовище програмування – це проміжний рівень, між системним та прикладним, що забезпечує останній необхідними сервісами виникнення, проходження та завершення застосувань в паралельній системі. Склад та функції цього проміжного шару програмного забезпечення все ще залежать від архітектури. У цьому розділі розглядаються основні концепції та засоби MPI [22]– середовища паралельного програмування для систем типу ЦС і РС.

MPI – сучасний стандарт інтерфейсу паралельних обчислень, орієнтованих на обмін повідомленнями. Роботи з його створення почалися у 1992р. і були викликані необхідністю уніфікації засобів підтримки паралельних обчислень, від різних виробників, що існували на той час. До розробки стандарту були залучені 60 спеціалістів з 40 організацій, державних і приватних (в основному США і Європи). Цілі стандарту визначали, з поміж іншого:

– незалежну від платформи семантику інтерфейсу;

– близькість до відомих на той час інших інтерфейсів, таких як PVM, NX, P4;

– зручність програмування для мови С і Фортрана;

– ефективна взаємодія процесів з прикриттям операцій обміну та обчислень;

– ефективна реалізація з урахування гетерогенного середовища.

Стандарт визначає:

– засоби обмінів “точка – точка”;

– групові операції;

– групи процесів та контексти обмінів;

– топології процесів;

– зв′язок з С та Фортраном;

– управління середовищем та запити профілювання.

Стандартом не визначаються (у першій версії):

– явні операції з спільною пам′яттю;

– засоби налагодження (debugging);

– явна підтримка потоків (threads);

– підтримка управління задачами;

– функції вводу/виводу.

По суті MPI – це бібліотека функцій, доступних для виклику з прикладної програми. Модель MPI – програми являє собою набір довільного, але фіксованого числа процесів, що взаємодіють між собою через виклики функцій бібліотеки MPI. Аргументами функцій можуть бути змінні трьох категорій: IN (використовуються, але не змінюються), OUT (можуть змінюватись викликом), INOUT – змішані: і використовуються, і змінюються. Кількість процесів визначається при ініціалізації. Процеси розрізняють за їх номерами в інтервалі цілих (0..groupsize – 1).

Взаємодії процесів можуть бути двох видів: парні і групові. Серед об′єктів взаємодії розрізняються непрозорі об′єкти (opaque) (наприклад, буфери), що знаходяться в системній пам′яті; і специфікатори (hendles), через які здійснюється доступ до непрозорих об′єктів. Наперед заданими специфікаторами є комунікатори, що забезпечують середовище (контекст) групових обмінів. Вони є обов′язковими аргументами функцій MPI і визначають групу процесів. Найбільш загальний комунікатор MPI_COMM_WORLD визначає універсальну групу всіх процесів.

Наступний приклад ілюструє програму MPI, в якій процеси визначають розмір групи і свій власниий номер у групі і виводять їх в стандартний потік вводу/виводу (на екрані):

```
main (int argc, char ** argv){
        Int me, size
MPI_INIT (& arge, &argv);
MPI_COMM_RANK (MPI_COMM_WORLD, &me);
MPI_COMM_SIZE (MPI_COMM_WORD, &size);
printf (“Process %d size %d⁄n”, me, size);
MPI_FINALIZE;
```

Парні взаємодії здійснюються через повідомлення, що складаються з двох частин: даних і конверта. Дані – це трійка (<кількість>,<ім′я>,<тип>), а конверт – чевірка (<відправник>.<одержувач>,<тег>,<комунікатор>).
Допускається, щоб відправник і одержувач співпадали. Взаємодії бувають, в залежності від режиму виконання:

– неблокуючі (асинхронні), при яких виклик функції може завершитись до завершеня самої операції обміну;

– блокуючі (синхронні) – в протилежному випадку;

– локальні, коли операція обміну не вимагає взаємодії з іншим процесором;

– колективні, коли всі процеси групи повинні виконувати виклики MPI – функцій для здійснення взаємодій. Ці режими не є взаємовиключеними.

До блокуючих операцій відносять операції MPI_SEND і MPI_RECV. Специфікація операції передачі має вид:

int MPI_SEND (void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_COMM comm).

Тут всі параметри – вхідні (IN): buf – початкова адреса розташування буфера. Блокуючий характер цієї операції полягає у тому, що операція не завершується до моменту, поки дані – змістовна частина повідомлення – не буде збережена поза процесом, що їх передає. Однак стандартом MPI передбачено можливість вибору режиму протоколу, що контролює обмін повідомленнями. Звичайним (стандартним) режимом є нелокальний, існують також додаткові режими готовності (MPI_RSEND) і синхронний (MPI_SSEND), що уточнюють нелокальний стандартний режим, а також режим буферизації (MPI_BSEND), що взагалі є локальним, коли виконання передачі ніяк не пов′язується з виконанням операції прийому. Режим готовності означає, що передача починається тільки після того, як відбувся виклик відповідної операції прийому. Режим синхронний не пов′язує початок передачі з викликом прийому, але пов′язує завершення процесу тільки після початку виконання прийому. Специфікація операції прийому має вид:

int MPI_RECV (void * buf, int count, MPI_Detatype datatype,
int source, int tag, MPI_COMM comm, MPI_STATUS * status

де status є вихідним параметром, що визначає довжину отриманого повідомлення.

Неблокуючі операції в MPI означають обмін, що розщеплено на дві фази: ініціалізації і завершення. Так, функція передачі має специфікацію:

int MPI_ISEND (void * buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_COMM comm, MPI_REQUEST * request).

(Буква I назва – від слова immediate). Як видно функція має додатковий параметр request, який поєднує між собою ініціювання і завершальну частину обміну даними.

Завершальна частина передачі здійснюється одною з двох функці: MPI_WAIT або MPI_TEST.

Int MPI_WAIT (MPI_REQUEST * request, MPI_STATUS * status)
Int MPI_TEST (MPI_REQUEST * request, int * flag, MPI_STATUS status).

Параметр request зберігає інформацію про завершення/незавершення обміну, а між парою викликів функції ініціалізації та завершення можуть знаходитися інші операції, інформаційно незалежні від цієї операції обміну.

MPI_ISEND означає, що виклик може завершуватись до моменту копіювання даних у буфер. Можливі варіанти протоколу R, S і B, як і для блокуючого варіанту. Різниця між MPI_WAIT і MPI_TEST – у тому, що перша є нелокальною; вона пов′язується з завершенням операції, заданої параметрами request. А друга – локальна, але результат ststus доступний тільки, поки flag = 1.

Список функцій групової взаємодії, зокрема, включає:

1. Бар′єрна синхронізація int MPI_BARRIER (MPI_COMM comm).
2. Передача “один – всім”(broadcast) MPI_BCAST.
3. Збір даних від всіх процесів в одному MPI_GATHER.
4. Розсилання n посилок всім n членамгрупи з одного (кореневого процесу): MPI_SCATTER.

5. Збір даних всіма процесами MPI_ALLGATHER.
6. Збір і розсилання від всіх процесів MPI_ALLTOALL
7. Операції редукції з асоціативно-комутативною операцією (наприклад, Σa_i) MPI_REDUCE.
8. Комбінована операція редукції і розсилання MPI_ALLREDUCE.
9. Сканування – префіксна операція редукції над, даними, розподіленими між членами групи MPI_SCAN.

1.8. Системи пам′яті та файлові системи сучасних комп'ютерних систем.

Головними видами взаємодій в паралельних системах є: 1) спільна пам′ять; 2) обмін повідомленнями; 3) віддалений виклик процедур (RPC), що включає передачу параметрів. Спільна пам′ять є простим, близьким за парадигмою до послідовного програмування засобом, використання якого проте обмежене за розміром і масштабованістю. Тому застосовується тільки в тіснозв′язаних системах, а саме, векторно-конвейерних та симметричної обробки.

Розподілена пам′ять та обмін повідомленнями не мають цих недоліків, проте важче програмуються, тому в сучасних системах, як правило, використовується комбінований підхід DSM розподіленої спільної пам′яті, що моделює спільний адресний простір (логічний) над (фізично) розподіленими модулями пам′яті. Оскільки параметри доступу до локальних і віддалених об′єктів у DSM – неоднаковий, то такі архітектури називаються NUMA (NonUniform Memory Aceess). Для забезпечення продуктивності стають необхідними міграція і реплікація об′єктів даних до локальної пам′яті процесора.

Існує два основних види NUMA архітектур: з глобальною спільною пам′яттю ГСП (multiprocessor cashe) і розподіленою спільною пам′яттю (власне DSM). Відображення сторінок DSM за фізичними адресами здійснюється ядром операційної системи.]

Ключова відмінність – це наявність ГСП у фізичній реалізації, і віртуальність спільної пам′яті – в DSM. Це накладає і інші архітектурні (шина для ГСП і мережа зв′язку – для DSM) відмінності. Час доступу до спільної пам′яті в DSM – набагато більший. Одиницями пам′яті в обох випадках є блоки (сторінки), що відображаються в кеші. ГСП мають перевагу у швидкості доступу, DSM – у прозорості доступу. Причому, DSM є більш прозора архітектура ніж механізм обміну повідомленнями, оскільки такі деталі, як перетворення типів, протокол, і т. ін. залишаються поза увагою прикладного програміста. Крім того, DSM забезпечує більшу масштабованість.

Головними проблемами для паралельних систем пам′яті є проблеми когерентності (цілісності) і несуперечливості (coherence & consistency), що виникають через можливості паралельного доступу та реплікації даних. З точки зору програміста когерентність означає, що результатом операції читання є значення, записане найостаннішою операцією запису. Але оскільки "найостанніший" в системах з відсутнім глобальним часом – поняття невизначене, то властивість когерентності ослабляється до вимоги всіх копій спільної змінної мети одне значення після виконання всіх операцій запису через деякий час. Несуперечливість – це когерентність (цілісність) у синхронізаційній точці. Ослабляючи вимоги неуперечливості, можна збільшувати можливості паралелізму операцій, а отже підвищення продуктивності.

В сучасних архітектурах спільними можуть вважатись не просто дані, а об′єкти. Прикладами об′єктних DSM моделей (реалізацій) є Orca [12] і, певною мірою, Linda. Більше того, сучасною тенденцією є розробка гібридних архітектур для DSM, де ця модель підтримується програмно-апаратно в масштабах локальних мереж з великою пропускною здатністю мережі.

Файл – об′єкт операційної системи для довго-тривалого зберігання даних (persistent object), а файлова система, як компонента ОС, відповідає за іменування, створення, видалення, модифікацію і захист файлів.

Розподілені файлові системи – це реалізації централізованих файлових систем на фізично розподілених пристроях пам′яті, розмежування яких є прозорим для користувача. Для користувача файл виглядає як трійка: <ім′я файла><атрибути файлу><дані>.

1) ім′я – символьне ім′я, яке за допомогою сервісу каталогів файлової системи відображається на фізичну адресу файла.
2) атрибути містять інформацію про володіння, тип, розмір, час створення і авторизацію доступу до файла.
3) дані файла можуть бути організовані як плоска структура потоку байтів (послідовний доступ), як послідовність блоків (прямий доступ) або ієрархічну структуру індексованих записів (індексний доступ).

Чотири головних компоненти файлової системи:

1) сервіс каталогів (directory service): іменування, створення і видалення файлів.
2) сервіс авторизації (authorization): управління доступом.
3) сервіс файлів (file service): операці read/write та установка атрибутів; управління копіями та паралельним доступом (concur)/
4) системний сервіс (system): управління пристроями, кешем та блоками даних.

Центральними для файлових (розподілених) систем є те, як за умов розподіленості файлів і множини користувачів, з одного боку, забезпечити прозорість одночасності, а з іншого – через кешування, реплікацію і

концепцію спільності файлів (sharing) забезпечити продуктивність (прозорість паралельності), готовність, масштабованість і стійкість до відмов.

Для файлових систем, де спільність файлів є суттєвою, прозорість паралельності забезпечується прийняттям протоколів управління: 2PL (двофазним), з відміткою часу (timestemp) і оптимістичним. Найвідоміший – двофазний, що на першому етапі здійснює закриття блоків з подальшим виконанням операцій читання, модифікації та запису, а на другому – звільнення блоків. причому після звільнення транзакція вже не виконує операцій закриття.

Еквівалентом цілісності (несуперечливості) при транзакційному підході до визначення паралельності доступу є поняття лінеаризації (serializability), тобто такого порядку виконання планів транзакції, який був би еквівалентний одному з послідовних планів транзакцій.

Відомими прикладами розподілених файлових систем є XDFS (Xerox distributed file system) та NFS (Network File System) від Sun [13]. XDFS підтримує мультиверсійність файлів; служба каталогів реалізована як сервіс файла-сервера. Управління паралелізмом виконується на основі двофазного протоколу. Особливість NFS – в можливості монтування користувачеві своєї власної локальної файлової системи, причому навіть для бездискових робочих станцій (ця можливість реалізується як сервіс на віддаленому сервері). NFS не виконує операцій відкриття і закриття файлів. Тому при "падінні" сервера інформація про файли не втрачається (безстановий сервер).

1.9. Паралельні алгоритми та засоби оцінки їх продуктиввності та ефективності.

Паралельні обчислювальні системи – дуже складні, щоб існували прості рецепти проектування паралельних алгоритмів. Метою проектування є досягнення високої продуктивності, але також масштабованості, відмовостійкості та інших характеристик, що впливають на продуктивність (кількість процесорів p, час паралельного виконання T_p, коефіцієнт прискорення s_p, коефіцієнт ефективності e_p, і ціна паралельного алгоритму c_p) [7,10,24]. Основними етапами проетування паралельного алгоритму є такі:

1) декомпозиція – вивчення можливостей розпаралелювання і розбиття обчислень на підзадачі незалежно від наявної кількості процесорів.
2) визначення схем обміну даними між задачами та координації їх взаємодій.
3) оцінка продуктивності при поєднанні декомпозиції та схем обміну і, при необхідності, укрупнення задач для покращення продуктивності.
4) відображення множини задач на конкретну архітектуру мультипроцесорної системи і організація ефективного використання процесорів.

В загальному випадку процес ітераційний [1-4]*.

Розглянемо процес проектування паралельної програми на прикладі задачі множення матриць [7].

Початковою специфікацією є вдіоме співідношення $C_{ij} = \Sigma a_{ik} * b_{kj}$, $1 \le i,j \le N$; де значення N передбачається достатньо великим.

1. При абстрактному розгляді алгоритму на етапі декомпозиції маємо N^2 підзадач, що кожна обчислюють один елемент результуючої матриці за вищеподаною формулою. На моделі PRAM [24] можемо мати $s_N^2 = N^2$ і $e_N^2 = 1$. Проте схема обмінів по одному числу (a_{ik} і b_{kj}) не може бути задовільною, бо шидкодія процесорів набагато перевищує швидкодію системи зв′язку (~ 10^9 оп/сек проти 10^{6-7} байт /сек).
2. Тому переходимо при такому розгляді до укрупненого подання підзадач. Маємо розбиття матриці N*N на блоки розміром R*R такі, що N=nR. Тепер маємо формулу $C_{ij} = \Sigma A_{ik} * B_{kj}$ обчислень блоків матриці, де операція * вже є матричною операцією. Зауважимо, що таке укрупнення операцій не завжди буває таким "красивим" і легким.
3. Отже, повернувшись до етапів 1-2, маємо блочний алгоритм, оцінка якого на етапі 3 визначається умовою балансу часу обробки і часу передачі одного блока :

$$\pi * R^3 \ge t_s + t_1 * R^2 ,$$

де π – швидкодія процесора (сек); t_s – час встановлення зв′язку (латентність; t_1 – час передачі одного машинного слова. Звідси визначається оптимальний розмір блока R.

4. Переходимо до вивчення ефективності використання процесорів.

Програма обчислення результючого блока:

```
C := 0
for k = 1 to R do {
A [i,k] →U;                                                    (1)
B [k,j] → V;
C := C + U * V;
}
```

має вигляд (1), де матричне множення блоків може виконуватись на векторно-конвейерному обладнанні (якщо воно є в складі архітектури процесорів). Проте підвищення продуктивності можливе і на макро рівні – за рахунок програмної конвейеризації обчислень ітерацій циклу (1).

```
U1 ← A[i,1]; V1 ← B[1j];  /* пролог*/
C = 0;  b := 1;
for k ;= 2 to R do

        if b = 1 then
        U2 ← A [i,k]; V2 ← B [k,j];
        b := 2; C := C + U1 * V1;
else   U1 ← A [i,k];  V1 ← B[k,j];
        b := 1; C := C + U2 * V2;
end if  end for
```

```
if  b = 1  then  C := C + U1 * V1;
else C := C + U2 * V2;
end if
```

Зауваження про продуктивність процесорів
(Комп′ютер пресс., 2002, № 8 , С.99-100).
Спрощена формула:

$$N = n*f,$$

Де N – продуктивність процесора;
n – кількість виконуваних команд за цикл;
f – частота процесора.
Але реальна продуктивність залежить ще від застосування (програми) та якості відкомпельованого коду.
Тактова частота напряму залежить від стійкості командного конвейєра: більше стадій – вища частота. FE: Pentium III – 12 стадій, Pentium IV – 20. Тому f для Pentium IV досягає 2 Ггц. Але при цьому дуже зростають вимоги до залежності команд. В найгіршому випадку довгий конвейєр може чекати 20 тактів завершення, щоб розпочати виконання залежної від попередньої команди. Тому, багатостадійні процесори добре придатні на застосуваннях, що здатні завантажувати конвейєр потоком незалежних команд, наприклад, при обробці мультимедійної інформації.

Частина 2. Операційні середовища.

2.1. Класифікація операційних систем.

Операційна система [24],[26],[27] - це інтерфейс між користувачем і архітектурою [25]. ОС реалізує віртуальну машину, яка (можна сподіватись) простіша для прикладної програми, ніж апаратні засоби.

Подана на рис. 2.1. схема дає абстрактний погляд на системні компоненти.

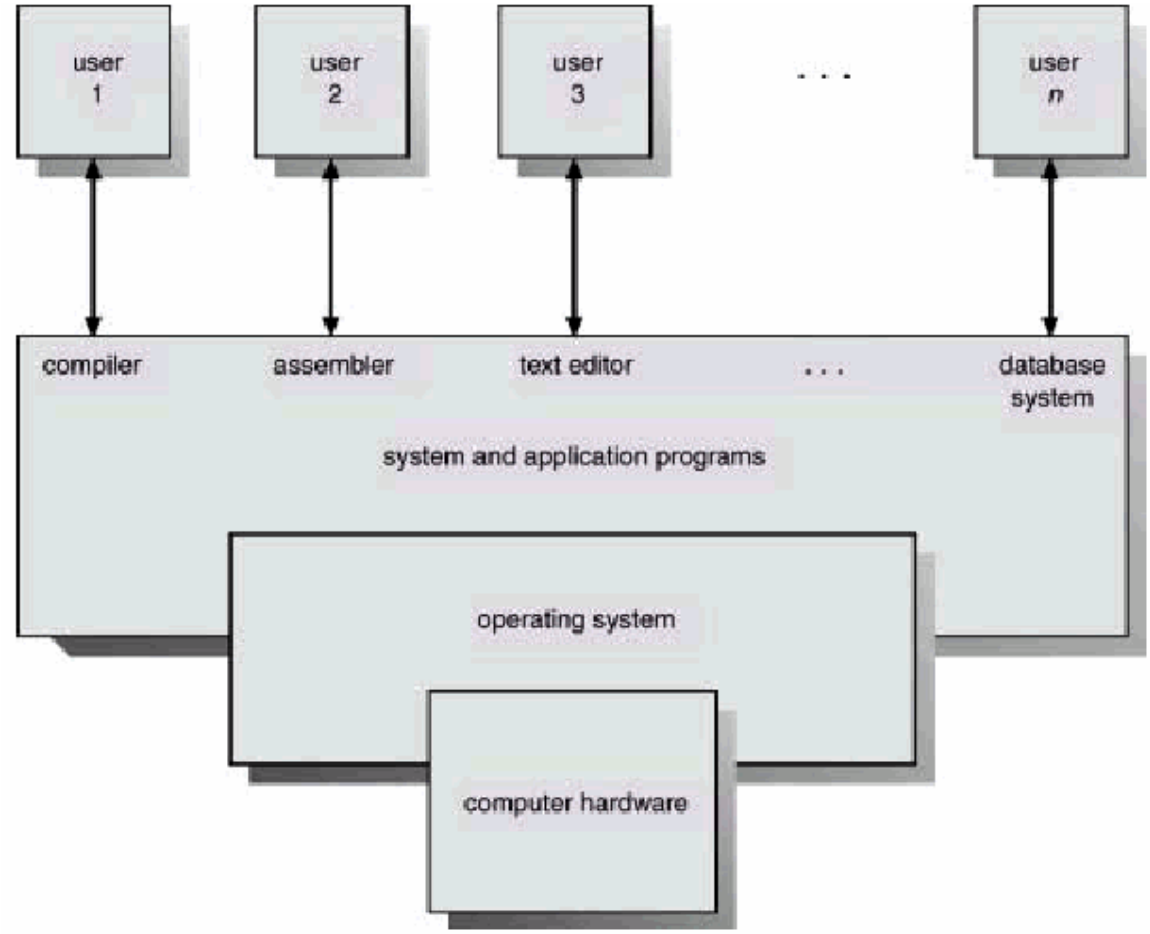


Рис 2.1. Компоненти ОС.

Сервіси: ОС надає стандартні сервіси (інтерфейси) для роботи з апаратними засобами. Ці сервіси включають файлову систему, віртуальну пам'ять, управління процесами, визначання часу та роботу в мережі.

ОС виконує дві, власне кажучи, мало пов'язані функції:

- забезпечення користувачу-програмісту зручностей наданням для нього розширеної машини і
- підвищення ефективності використання комп'ютера шляхом раціонального керування його ресурсами.

ОС як розширена машина

Використання більшості комп'ютерів на рівні машинної мови обтяжливе, особливо це стосується вводу-виводу. Навіть якщо не заглиблюватись в проблему програмування вводу-виводу, зрозуміло, що серед програмістів знайшлося б небагато охочих безпосередньо займатися програмуванням цих операцій.

При роботі з диском програмісту-користувачу досить уявляти його у вигляді деякого набору файлів, кожний з який має ім'я. Робота з файлом полягає в його відкритті, виконанні читання чи запису, а потім у закритті файлу. Питання як це відбувається, не повинні турбувати користувача.

Програма, що ховає від програміста всі реалії апаратури і надає можливість простого, зручного перегляду зазначених файлів, чи читання запису - це, звичайно, ***операційна система***. Операційна система бере на себе всі малоприємні справи, пов'язані з обробкою переривань, керуванням таймерами й оперативною пам'яттю, а також інші низькорівневі проблеми. У кожному випадку та **абстрактна, уявлювана машина**, з якою, завдяки операційній системі, тепер може мати справу користувач, набагато простіша і зручніша в звертанні, ніж реальна апаратура, що лежить в основі цієї абстрактної машини.

З цього погляду функцією ОС є надання користувачу деякої **розширеної чи віртуальної машини**, яку легше програмувати і з якою легше працювати, ніж безпосередньо з апаратурою, що складає реальну машину.

ОС як система керування ресурсами

ОС – деякий механізм, що керує всіма частинами складної системи. Сучасні обчислювальні системи складаються з процесорів, пам'яті, таймерів, дисків, мережної комунікаційної апаратури, принтерів та інших пристроїв. Функцією ОС є розподіл процесорів, пам'яті, пристроїв і даних між процесами, що конкурують за ці ресурси. ОС повинна керувати всіма ресурсами обчислювальної машини таким чином, щоб забезпечити максимальну ефективність її функціонування. Критерієм ефективності може бути, наприклад, пропускна здатність або реактивність системи.

Керування ресурсами включає рішення двох загальних, що не залежать від типу ресурсу, задач:

- **планування ресурсу** - тобто визначення, кому, коли, а для ресурсів, що можуть бути розділені, в якій кількості, необхідно виділити даний ресурс;
- **відстеження стану ресурсу** - тобто підтримка оперативної інформації про те, зайнятий чи не зайнятий ресурс, а для ресурсів, що можуть бути розділені, - яку кількість ресурсу вже розподілено, а яка вільна.

Для розв’язання цих загальних задач керування ресурсами різні ОС використовують різні алгоритми, що в кінцевому рахунку і визначає їхній вигляд у цілому, включно з характеристиками продуктивності, областю застосування і навіть користувальницьким інтерфейсом. Так, наприклад, алгоритм керування процесором у значній мірі визначає, чи є ОС системою поділу часу, системою пакетної обробки чи системою реального часу.

Класифікація ОС

Операційні системи можуть розрізнятися особливостями реалізації внутрішніх алгоритмів керування основними ресурсами комп'ютера (процесорами, пам'яттю, пристроями), особливостями використаних методів проектування, типами апаратних платформ, областями використання і багатьма іншими властивостями.

Нижче наведена класифікація ОС за декількома найбільш значимими ознаками.

Особливості алгоритмів керування ресурсами

Від ефективності алгоритмів керування локальними ресурсами комп'ютера багато в чому залежить ефективність усієї ОС у цілому. Тому, характеризуючи ОС, часто наводять найважливіші особливості реалізації функцій ОС в керуванні процесорами, пам'яттю, зовнішніми пристроями комп'ютера. Так, наприклад, у залежності від особливостей використаного алгоритму керування процесором, операційні системи поділяють на

- багатозадачні й однозадачні,
- багатокористувацькі й однокористувацькі,
- на системи, що підтримують багатониткову (багатопроцесну) обробку і, що не підтримують,
- на багатопроцесорні й однопроцесорні системи.

Підтримка багатозадачності. За числом одночасно виконуваних задач операційні системи можуть бути розділені на два класи:

- однозадачні (наприклад, MS-DOS) і
- багатозадачні (ОС ЕС, OS/2, UNIX, Windows 98, Windows XP).

Однозадачні ОС в основному виконують функцію надання користувачу віртуальної машини, роблячи більш простим і зручним процес взаємодії користувача з комп'ютером.

Однозадачні ОС містять:

засоби керування периферійними пристроями,

засоби керування файлами,

засоби спілкування з користувачем.

Багатозадачні ОС, крім перерахованих вище функцій, керують поділом спільно використовуваних ресурсів, таких як

процесор,

оперативна пам'ять,

файли і зовнішні пристрої.

Підтримка багатокористувацького режиму. По числу користувачів, що одночасно працюють, ОС поділяються на:

- однокористувацькі (MS-DOS, Windows 3.x, ранні версії OS/2);
- багатокористувацькі (UNIX, Windows NT/2000).

Головною відмінністю багатокористувацьких **систем** від однокористувацьких є наявність засобів захисту інформації кожного користувача від несанкціонованого доступу інших користувачів. Варто відмітити, що не всяка багатозадачна система є багатокористувацькою, і не всяка однокористувацька ОС є однозадачною.

Багатозадачність, що витісняє, і, що не витісняє .

Найважливішим поділюваним ресурсом є **процесорний час**. Спосіб розподілу процесорного часу між декількома одночасно існуючими в системі процесами (або нитками – потоками управління) багато в чому визначає специфіку ОС. Серед безлічі існуючих варіантів реалізації багатозадачності можна виділити дві групи алгоритмів:

- багатозадачність, що не витісняє (NetWare, Windows 3.x);
- багатозадачність, що витісняє (Windows NT/2000, OS/2, UNIX).

Основним розходженням між що цими варіантами багатозадачності є ступінь централізації механізму планування процесів:

(не витісняє **)** у цьому випадку механізм планування процесів розподілений між системою і прикладними програмами;

(витісняє**)** тут механізм планування процесів повністю зосереджений в операційній системі.

При багатозадачності, що не витісняє, активний процес виконується доти, поки він сам, за власною ініціативою, не віддасть керування операційній системі для того, щоб та вибрала з черги інший готовий до виконання процес. При багатозадачності, що витісняє, рішення про переключення процесора з одного процесу на іншій приймається операційною системою, а не самим активним процесом.

Підтримка багатонитковості (багатопотоковості). Важливою властивістю операційних систем є можливість розпаралелення обчислень у рамках однієї задачі. Багатониткова ОС розділяє процесорний час не між задачами, а між їх окремими галузями (нитками – потоками управління).

Багатопроцесорна обробка. Іншою важливою властивістю ОС є відсутність чи наявність у ній засобів підтримки багатопроцесорної обробки - мультипроцесування. Мультипроцесування приводить до ускладнення всіх алгоритмів керування ресурсами.

У наші дні стає загальноприйнятим введення в ОС функцій підтримки багатопроцесорної обробки даних.

Багатопроцесорні ОС можуть класифікуватися за способом організації обчислювального процесу в системі з багатопроцесорною архітектурою:

- асиметричні ОС;
- симетричні ОС.

Асиметрична ОС цілком виконується тільки на одному з процесорів системи, розподіляючи прикладні задачі по інших процесорах.

Симетрична ОС цілком децентралізована і використовує весь пул процесорів, розділяючи їх між системними і прикладними задачами.

Вище були розглянуті характеристики ОС, пов'язані з керуванням тільки одним типом ресурсів - процесором. Важливий вплив на вигляд операційної системи в цілому, на можливості її використання в тій чи іншій області роблять особливості й інших підсистем керування локальними ресурсами – підсистем:

керування пам'яттю,

файлами,

пристроями вводу-виводу.

Специфіка ОС виявляється й у тому, яким чином вона реалізує мережні функції:
розпізнавання і перенаправлення у мережу запитів до віддалених ресурсів,
передача повідомлень в мережі,
виконання віддалених запитів.

При реалізації мережних функцій виникає комплекс задач, пов'язаних з розподіленим характером збереження й обробки даних у мережі:
ведення довідкової інформації про усі доступні у мережі ресурси і сервери,
адресація взаємодіючих процесів,
забезпечення прозорості доступу,
тиражування даних,
узгодження копій,
підтримка безпеки даних.

Особливості апаратних платформ

На властивості операційної системи безпосередній вплив мають апаратні засоби, на які вона орієнтована. За типами апаратури розрізняють операційні системи
персональних комп'ютерів,
міні-комп'ютерів,
мейнфреймів,
кластерів і
мереж ЕОМ.

Серед перерахованих типів комп'ютерів можуть бути як однопроцесорні варіанти, так і багатопроцесорні. У будь-якому випадку специфіка апаратних засобів, як правило, відбивається на специфіці операційних систем.

Очевидно, що ОС великої машини є більш складною і функціональною, ніж ОС персонального комп'ютера. Так, в ОС великих машин функції по плануванню потоку виконуваних задач, очевидно, реалізуються шляхом використання складних пріоритетних дисциплін і вимагають більшої обчислювальної потужності, ніж в ОС персональних комп'ютерів. Аналогічна ситуація і з іншими функціями.

Мережна ОС має у своєму складі засоби передачі повідомлень між комп'ютерами по лініях зв'язку, що зовсім не потрібні в автономній ОС. На основі цих повідомлень мережна ОС підтримує поділ ресурсів комп'ютера між віддаленими користувачами, під’єднаними до мережі. Для підтримки функцій передачі повідомлень мережні ОС містять спеціальні програмні компоненти, що реалізують популярні комунікаційні протоколи, такі як IP, IPX, Ethernet і інші.

Багатопроцесорні системи вимагають від операційної системи особливої організації, за допомогою якої сама операційна система, а також підтримувані нею додатки могли б виконуватися паралельно окремими процесорами системи. Паралельна робота окремих частин ОС створює

додаткові проблеми для розробників ОС, тому що в цьому випадку набагато складніше забезпечити
погоджений доступ окремих процесів до загальних системних таблиць,
уникнення ефекту гонок і
інші небажані наслідки асинхронного виконання робіт.

Інші вимоги постають до операційних систем кластерів. **Кластер -** слабко зв'язана сукупність декількох обчислювальних систем, що працюють спільно для виконання загальних додатків, і таких, що уявляються користувачу єдиною системою. Поряд зі спеціальною апаратурою для функціонування кластерних систем необхідна і програмна підтримка з боку операційної системи, що зводиться в основному до синхронізації доступу до поділюваних ресурсів, виявленню відмовлень і динамічної реконфігурації системи.

Поряд з ОС, орієнтованими на зовсім визначений тип апаратної платформи, існують операційні системи, спеціально розроблені таким чином, щоб вони могли бути легко перенесені з комп'ютера одного типу на комп'ютер іншого типу, так називані мобільні ОС. Найбільш яскравим прикладом такої ОС є популярна система UNIX. У цих системах аппаратно-залежні місця ретельно локалізовані, так що при переносі системи на нову платформу переписуються лише вони. Засобом, що полегшує перенос іншої частини ОС, є написання її машинно-незалежною мовою, наприклад, на С, що і була розроблений для програмування операційних систем.

Особливості областей використання

Багатозадачні ОС підрозділяються на три типи відповідно до використаних при їх розробці критеріїв ефективності:

- системи пакетної обробки (наприклад, ОС ЕС),
- системи поділу часу (UNIX, VMS),
- системи реального часу (QNX, RT/11).

Системи пакетної обробки призначалися для розв’язку задач в основному обчислювального характеру, що не потребують швидкого одержання результатів. Головною метою і критерієм ефективності систем пакетної обробки є максимальна пропускна здатність, тобто розв’язок максимального числа задач в одиницю часу. Для досягнення цієї мети в системах пакетної обробки використовуються наступна схема функціонування: на початку роботи формується пакет завдань, кожне завдання містить вимогу до системних ресурсів; з цього пакета завдань формується мультипрограмна суміш, тобто безліч одночасно виконуваних задач. Для одночасного виконання вибираються задачі, що висувають вимоги до ресурсів, що відрізняються так, щоб забезпечувалося збалансоване завантаження всіх складових обчислювальної машини. Наприклад, у мультипрограмній суміші бажана одночасна присутність обчислювальних задач і задач з інтенсивним вводом-виводом. Таким чином, вибір нового завдання з пакета завдань залежить від внутрішньої ситуації, що складається

в системі, тобто вибирається "вигідне" завдання. Отже, у таких ОС неможливо гарантувати виконання того чи іншого завдання протягом визначеного періоду часу. У системах пакетної обробки перехід процесора з виконання однієї задачі на виконання іншої відбувається тільки у випадку, якщо активна задача сама відмовляється від процесора, наприклад, через необхідність виконати операцію вводу-виводу. Тому одна задача може надовго зайняти процесор, що унеможливлює виконання інтерактивних задач. Таким чином, взаємодія користувача з обчислювальною машиною, на якій установлена система пакетної обробки, зводиться до того, що він приносить завдання, віддає його диспетчеру-оператору, а наприкінці дня після виконання всього пакета завдань отримує результат. Очевидно, що такий порядок знижує ефективність роботи користувача.

Системи поділу часу покликані виправити основний недолік систем пакетної обробки - ізоляцію користувача-програміста від процесу виконання його задач.

Кожному користувачу системи поділу часу надається термінал, з якого він може вести діалог зі своєю програмою. Тому що в системах поділу часу кожній задачі виділяється тільки квант процесорного часу, жодна задача не займає процесор надовго, і час відповіді виявляється прийнятним. Якщо квант обраний досить невеликим, то у всіх користувачів, що одночасно працюють на одній і тій же машині, складається враження, що кожний з них одноосібно використовує машину.

Зрозуміло, що системи поділу часу мають меншу пропускну здатність, ніж системи пакетної обробки, тому що на виконання приймається кожна запущена користувачем задача, а не та, котра "вигідна" системі, і, крім того, є додаткові накладні витрати обчислювальної потужності на частіше переключення процесора з задачі на задачу. Критерієм ефективності систем поділу часу є не максимальна пропускна здатність, а зручність і ефективність роботи користувача.

Системи реального часу застосовуються для керування різними технічними об'єктами, такими, наприклад, як верстат, супутник, наукова експериментальна чи установка технологічними процесами, такими, як гальванічна лінія, доменний процес і т.п. У всіх цих випадках існує гранично припустимий час, протягом якого повинна бути виконана та чи інша програма, що керує об'єктом, у противному випадку може статися аварія: супутник вийде з зони видимості, експериментальні дані, що надходять з датчиків, будуть загублені, товщина гальванічного покриття не буде відповідати нормі. Таким чином, критерієм ефективності для систем реального часу є їхня здатність витримувати заздалегідь задані інтервали часу між запуском програми й одержанням результату (керуючого впливу). Цей час називається **часом реакції системи**, а відповідна властивість системи **- реактивністю**. Для цих систем мультипрограмна суміш являє собою **фіксований набір заздалегідь розроблених програм**, а вибір програми на виконання здійснюється виходячи з поточного стану чи об'єкта відповідно до розкладу планових робіт.

Деякі операційні системи можуть сполучати в собі властивості систем різних типів, наприклад, частина задач може виконуватися в режимі пакетної обробки, а частина - у режимі реального часу чи в режимі поділу часу. У таких випадках режим пакетної обробки часто називають **фоновим режимом.**

До ознак корпоративних ОС можуть бути віднесені також наступні особливості.

Підтримка додатків. У корпоративних мережах виконуються складні додатки, що вимагають для виконання великої обчислювальної потужності. Такі додатки розділяються на кілька частин, наприклад, на одному комп'ютері виконується частина додатка, зв'язана з виконанням запитів до бази даних, на іншому - запитів до файлового сервісу, а на клієнтських машинах - частину, що реалізує логіку обробки даних додатка й організовує інтерфейс із користувачем. Обчислювальна частина загальних для корпорації програмних систем може бути занадто об'ємною і важкою для робочих станцій клієнтів, тому додатки будуть виконуватися більш ефективно, якщо їх найбільш складні в обчислювальному відношенні частини перенести на спеціально призначений для цього потужний комп'ютер - ***сервер додатків***.

Сервер додатків повинен базуватися на могутній апаратній платформі (мультипроцесорні системи, кластерні архітектури). ОС сервера додатків повинна забезпечувати високу продуктивність обчислень, а значить підтримувати багатониткову обробку, багатозадачність, що витісняє, мультипроцесування, віртуальну пам'ять і найбільш популярні прикладні середовища (UNIX, Windows). У цьому відношенні мережну ОС NetWare важко віднести до корпоративних продуктів, тому що в ній відсутні майже усі вимоги, поставлені до сервера додатків. У той же час хороша підтримка універсальних додатків у Windows NT/2000 власне і дозволяє їй претендувати на місце у світі корпоративних продуктів.

Довідкова служба. Корпоративна ОС повинна мати здатність зберігати інформацію про всіх користувачів і ресурси таким чином, щоб забезпечувалося керування нею з однієї центральної точки. Подібна великій організації, корпоративна мережа має потребу в централізованому збереженні якомога більш повної довідкової інформації про саму себе (починаючи з даних про користувачів, сервери, робочі станції і закінчуючи даними про кабельну систему). Природньо організувати цю інформацію як **базу даних**. Дані з цієї бази можуть бути запрошені багатьма мережними системними додатками, у першу чергу системами керування й адміністрування. Крім цього, така база корисна при організації електронної пошти, систем колективної роботи, служби безпеки, служби інвентаризації програмного й апаратного забезпечення мережі, та й для практично будь-якого великого бізнес-додатка.

База даних, що зберігає довідкову інформацію, надає все те ж різноманіття можливостей і породжує всі ту ж безліч проблем, що і будь-яка інша велика база даних. Вона дозволяє здійснювати різні операції пошуку,

сортування, модифікації і т.п., що дуже сильно полегшує життя як адміністраторам, так і користувачам. Але за ці зручності доводиться розплачуватися рішенням проблем розподіленості, реплікації і синхронізації.

В ідеалі мережна довідкова інформація повинна бути реалізована як єдина база даних, а не бути набором баз даних, що спеціалізуються на збереженні інформації того чи іншого виду, як це часто буває в реальних операційних системах. Наприклад, у Windows NT є принаймні п'ять різних типів довідкових баз даних.

Головний довідник домена (NT Domain Directory Service) зберігає інформацію про користувачів, що використовується при організації їхнього логічного входу в мережу.

Дані про тих же користувачів можуть міститися й в іншому **довіднику**, використовуваному електронною поштою **Microsoft Mail**.

Ще три бази даних підтримують дозвіл на низькорівневі адреси:

WINS (Windows Internet Naming Service — служба імен Internet для Windows (є базою даних імен комп'ютерів і зв'язаних з ними IP-адрес у середовищі TCP/IP) – встановлює відповідність Netbios-імен IP-адресам,

довідник DNS (**D**omain **N**ame **S**erver – сервер доменних імен (службовий комп'ютер мережі, що переводить імена комп'ютерів у доменних записах в адреси IP)) – сервер імен домена – виявляється корисним при підключенні NT-мережі до Internet, і нарешті,

довідник протоколу DHCP (Dynamic Host Configuration Protocol – протокол динамічної конфігурації хоста (мережний стандарт, що регламентує процес присвоювання сервером IP-адрес та іншої конфігураційної інформації машинам-клієнтам) використовуєть-ся для автоматичного призначення IP-адрес комп'ютерам мережі.

Ближче до ідеалу знаходяться довідкові служби, що поставляються фірмою Banyan (продукт Streettalk III) і фірмою Novell (NetWare Directory Services), що пропонують єдиний довідник для всіх мережних додатків. Наявність єдиної довідкової служби для мережної операційної системи - одна з найважливіших ознак її корпоративності.

Безпека. Особливу важливість для ОС корпоративної мережі мають питання безпеки даних. З одного боку, у великомасштабній мережі об'єктивно існує більше можливостей для несанкціонованого доступу – через децентралізацію даних і велику розподіленість "законних" точок доступу, через велике число користувачів, благонадійність яких важко встановити, а також через велике число можливих точок несанкціонованого підключення до мережі. З іншого боку, корпоративні бізнес-додатки працюють з даними, що мають життєво важливе значення для успішної роботи корпорації в цілому. І для захисту таких даних у корпоративних мережах поряд з різними апаратними засобами використовується весь спектр засобів захисту, наданий операційною системою: мандатні права доступу, складні процедури аутентификации користувачів, програмна шифрация.

Характерними ознаками розподіленої організації ОС є:

наявність єдиної довідкової служби поділюваних ресурсів,
наявність єдиної служби часу,
використання механізму виклику віддалених процедур (RPC) для прозорого розподілу програмних процедур по машинах,
наявність багатониткової обробки, що дозволяє розпаралелювати обчислення в рамках однієї задачі і виконувати цю задачу одночасно на декількох комп'ютерах мережі, а також
наявність інших розподілених служб.

Таким чином, розподілена ОС — це в першу чергу мережна ОС.

2.2. Історія операційних систем.

Період 0: Технічні засоби (hardware) дуже дорогі, експериментальні, операційних систем не існує.

Один користувач у наданий йому період часу на пульті.
Одна функція в один період часу (не суміщаються обчислення і ввод - вивід).
Для відладки користувач застосовує головний пульт комп'ютера.

У 1960 році один з авторів, Синявський О.Л., працював на обчислювальній машині "Київ" і розробив програму "Чисельний прогноз вітру на середньому рівні атмосфери". Операційної системи тоді ще не було, але деякі програми, що покращують інтерфейс між користувачем та апаратурою були і тоді, наприклад, програми переводу чисел з десятичної системи у двоїчну та навпаки. Провів експеримент по розробці відладчика (debugger) і запису його у ПЗУ. Машини М-20, М-220 також відносились до Фази 0. У 1963-68 роках з групою співробітників розробляв на них програмну систему в галузі комп'ютерної механіки. Програми складались безпосередньо у машинному коді, навіть не на ассемблері, але для їх об'єднання у програмний комплекс було створено спеціалізований завантажувальник програм та редактор зв'язків – компоненти, які тепер є в операційних системах.

Період 1: Технічні засоби (hardware) дорогі, люди дешеві.

Пакетна обробка (Batch processing): завантаження (loading) програми, виконання (running), роздрукування результатів та дампів.
Користувач дає свою програму (на перфокартах або магнітній стрічці) людині, яка планує роботи (shedules the jobs) приклади робіт: програми на фортрані і паскалі.
ОС (Resident monitor) завантажує, виконує і дампує роботу користувача.
Архітектурою передбачені канали даних, переривання, суміщення вводу\виводу і обчислень. Відсутність захисту – одна робота в один період часу.

М-222 – був комп'ютером саме такого типу. Для нього була розроблена операційна система (автор Чесалін – співробітник Інституту космічних досліджень АН СРСР). Основним документом тоді був "Лістінг"- роздруковка тексту програм ОС на асемблері. Синявський О.Л. розробив у рамках цієї ОС
програмне забезпечення експериментального зразка графічного дісплея (розробник hardware – С. Клименко з Харківського політехнічного інституту).

Розподіл оперативної пам'яті для систем такого типу наведено на Рис 2.2.

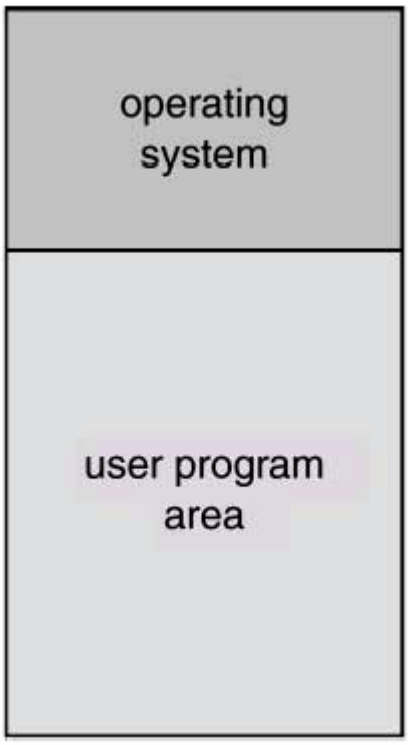


Рис 2.2. Розподіл пам'яті.

Мультипрограмування (Multiprogrammed batch systems) – виконання кількох програм в один період часу.

Захист та перепризначення (Protection and Relocation) пам'яті.

У режимі мультипрограмування декілька програм виконуються одночасно, розділяючи ресурси машини, тобто ввод\вивід та робота центрального процесора частково перекриваються. ОС управляє взаємодією між конкуруючими програмами

Досягнуто покращення використання технічних засобів, але відладка є більш тяжкою.

БЕСМ-6 за своєю архітектурою мала всі можливості, потрібні для ефективної реалізації режиму мультипрограмування, але її рання операційна система (керівник розробки - Томілін) цього ще не дозволяла. Вона була написана не на асемблері, а прямо у машинному коді. Свою першу системну програму – програмне забезпечення плотера (тепер така програма називається драйвером) Синявський О.Л. розробив саме в цій системі. Йому пощастило бути свідком переводу ОС на асемблер, реалізації управління пам’яттю (Paging). Це було зроблено у Дубні – міжнародному центрі фізичних досліджень. З метою підвищення ефективності використання комплекту великоформатних плотерів та графічних дисплеїв розробив бібліотеку програм, що підтримує реалізацію потоків управління та їх синхронізації.

Розподіл оперативної пам’яті для систем такого типу подано на Рис2.3.

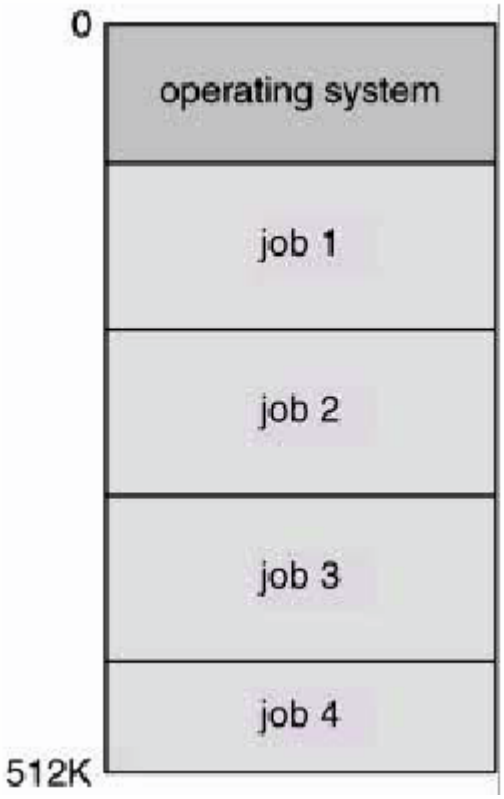


Рис 2.3. Розподіл пам'яті.

Період 2: Технічні засоби не так дорогі, як раніше, люди дорогі.

Мультипрограмування у режимі розділення часу (timesharing).

Кілька терміналів у одного комп’ютера. Всі користувачі взаємодіють з системою одночасно. Диски дешеві, так що програми і дані доступні в “on line”. Відладка є більш зручною.

ICL – ця англійська фірма встановила в Інституті фізики високих енергій АН СРСР (відомому більше під назвою “Серпуховский ускоритель”) систему timesharing.

Виникли нові проблеми:

потреба у перериванному управлінні процесами (preemptive scheduling) для забезпечення адекватного часу відгуку (response time);

потреба у згортанні (swapping) програм і видаленні їх з пам’яті;

потреба у проведенні адекватних заходів по захисту (security).

Success: UNIX developed at Bell Labs so a couple of computer nerds (Thompson, Ritchie) could play Star Trek on an unused PDP-7 minicomputer.

Період 3: Технічні засоби (hardware) дуже дешеві, люди дорогі.

Персональний комп’ютер.

Повернення до простоти. Операційна система спрощена, звільнена від підтримки мультипрограмування та захисту.

Але ця фаза також уже відійшла в історію – ***чому?***

Період 4: **Сучасна функціональність операційних систем.**

1. Операційна система забезпечує узгодження (Cocurrency) виконання багатьох справ (програм, операцій вводу\виводу, процесів і т. п.) одночасно. Кілька користувачів працюють одночасно так, як би вони мали окремі власні машини.
Потоки (□□□□, Threads) управління, одиниці управління в ОС, - один потік в даний момент часу, але багато потоків активні конкурентно.

2. Пристрої вводу\виводу, використовують переривання і тому дозволяють працювати процесору одночасно з ними. Кожний пристрій вводу\виводу має малий процесор і може працювати автономно. Центральний процесор дає команду пристрою вводу\виводу і продовжує свою роботу. Коли пристрій вводу\виводу закінчує свою роботу, він видає переривання. Центральний процесор зупиняє те, що він робив, і ОС обробляє переривання вводу\виводу.

3. Операційна система здійснює управління пам'яттю - координує розподілення пам'яті та переміщує дані між диском та головною пам'яттю.

4. Операційна система здійснює управління файлами - координує використання простору для файлів, порядок розміщення та пошуку багатьох файлів.

5. Реалізовані розподілені системи і мережі - ОС дозволяє групам робочих станцій працювати одночасно.

Сучасний вигляд інтерфейсу між архітектурою і користувачем, представлено у поданій нижче таблиці.

Hardware Example	***OS Services***	***User Abstraction***
Processor	Process management, Scheduling, Exceptions, Protection, Billing, Synchronization.	Process
Memory	Management, Protection, Virtual memory	Address space
I/ O devices	Concurrency with CPU, Interrupt handling	Terminal, Mouse, Printer, (System Calls)
File system	Management, Persistence	Files
Distributed systems	Network security Distributed file system	RPC system calls, Transparent file sharing

RPC – віддалений виклик процедур.

2.3. Функції і структура операційних систем.

Управління процесами (Process Management).

Оперційна система управляє багатьма типами активностей:

- програмами користувачів;
- системними програмами: управління прінтерами (printer spoolers), name серверами, file серверами, та ін.

Кожна з цих активностей інкапсульована в процес.
Процес включає повний контекст виконання (execution context) – команди, дані, значення лічильника команд, регістри, ресурси, що використовуються ОС, та ін.

Процес – це **не** програма. Процес є **одним екземпляром виконання (execution) програми;** в одній програмі можуть виконуватись (running) багато процесів.

Операційна система повинна:

- створювати, знищувати, підсумовувати (resume), та планувати процеси;
- підтримувати міжпроцесорні комунікації та синхронізацію, контролювати deadlock.

Управління пам'яттю (Memory Management).

Головна (Primary, Main) пам'ять забезпечує прямий доступ для процесора (CPU). Процеси під час виконання повинні бути в головній пам'яті.

ОС повиннна:

Механіка:

- слідкувати за використанням пам'яті;
- відслідковувати, яка пам'ять не використовується - unused ("free") memory;
- захищати простір пам'яті;
- відводити та вивільняти простір (Allocate, deallocate space) для процесу;
- згортати (Swap) процеси: пам'ять ↔ диск;

Політика:

- вирішувати, коли завантажувати кожний процес у пам'ять;
- вирішувати, скільки місця в пам'яті виділяти кожному процесу;
- вирішувати, коли процес буде видалений з пам'яті.

Управління файловою системою (File System Management).

Файл – є довготермінове сховище сутностей, іменована як сукупність сталої інформації, яку можна читати або записувати.

Диски (Вторинне сховище – secondary storage) підтримують довготермінове зберігання, але незручні для безпосереднього використання.

Частина ОС, що працює з файлами, називається **файловою системою.** Файлова система підтримує:

- файли і різні операції над ними.
- директорії, які містять файли та інші директорії.

Для файлів та директорій підтримуються: ім'я, розмір, дата створення, дата останної модифікації, власник, та ін.

ОС повинна:

- створювати та видаляти (Create and delete) файли та директорії;
- маніпулювати файлами та директоріями (read, write, extend, rename, copy, protect);
- проводити загальний високорівневий сервіс (backups, accounting, quotas).

Управління дисками (Disk Management).

Диск – це великий, достатній для зберігання всіх програм користувачів та даних, прикладних програм, цілої ОС, технічний пристрій, на якому встановлена файлова система .

Властивість диску: **живучість** (Persistent) – витримування системних відмов.

У процесі управління дисками операційна система повинна:

- управляти дисковим простором на низькому рівні;
- слідкувати за використанням простору;
- відслідковувати, який простір не використовується;
- відслідковувати пошкоджені блоки ("bad blocks");
- керувати низько-рівневими дисковими функціями, такими, як: планування (Scheduling) дискових операцій, переміщення головок.

Зверніть увагу на різницю між управлінням диском і управлінням файловою системою

Системні виклики (System Calls).

Для виконання операцій, що реалізуються не в центральному процесорі, а з застосуванням інших технічних засобів, програма звертається до операційної системи через системні виклики. Основні типи системних викликів є такими.

Управління процесами:

- завешити/видалити (end/abort) цю програму;

- завантажити/виконувати (load/execute) іншу програму;
- створити/знищити (create/terminate) процес;
- надати/встановити (get/set) атрибути;
- чекати призначеного часу (wait specified time);
- чекати подію (event), сигнал події.

Маніпулювання файлами:
- create/open/read/write/close/delete file;
- get/set attributes.

Маніпулювання пристроями (device):
- request/read/write/release device.

Інформація:
- get/set time/date.

2.4. Особливості методів побудови операційних систем.

До базових концепцій структурної організації операційної системи відносяться:

- **Способи побудови ядра системи - монолітне ядро** чи **мікроядерний підхід**. Більшість ОС використовує монолітне ядро, яке компонується як одна програма, що працює в привілейованому режимі і використовує швидкі переходи з однієї процедури на іншу, не потребуючи переключення з привілейованого режиму в користувальницький і навпаки. Альтернативою є побудова ОС на базі мікроядра, що працює також у привілейованому режимі і виконує тільки мінімум функцій з керування апаратурою, у той час як функції ОС більш високого рівня виконують спеціалізовані компоненти ОС - **сервери**, що працюють у користувальницькому режимі. При такій побудові **ОС** працює більш повільно, тому що часто виконуються переходи між привілейованим режимом і користувальницьким, проте система виходить більш гнучкою – її функції можна нарощувати, модифікувати чи звужувати, додаючи, модифікуючи чи видаляючи сервери користувацького режиму. Крім того, сервери добре захищені один від одного, як і будь-які користувацькі процеси.
- Побудова ОС на базі об’єктно-орієнтованого підходу дає можливість використовувати всі його переваги, що добре зарекомендували себе на рівні додатків, всередині операційної системи, а саме:

акумуляцію вдалих рішень у формі стандартних об'єктів,
можливість створення нових об'єктів на базі наявних за допомогою механізму спадкування,

гарний захист даних за рахунок їхньої інкапсуляції у внутрішні структури об'єкта, що робить дані недоступними для несанкціонованого використання ззовні,

структурованість системи, що складається з набору добре визначених об'єктів.

- Наявність декількох прикладних середовищ дає можливість **у** рамках однієї ОС одночасно виконувати додатки, розроблені для декількох ОС. Багато сучасних операційних систем підтримують одночасно прикладні середовища MS-DOS, Windows, UNIX (POSIX), OS/2 чи хоча б деякої підмножини з цього популярного набору. **Концепція множинних прикладних середовищ** найбільше просто реалізується в ОС на базі мікроядра, над яким працюють різні сервери, частина яких реалізують прикладне середовище тієї чи іншої операційної системи.
- Розподілена організація операційної системи дозволяє спростити роботу користувачів і програмістів у мережних середовищах. У розподіленій ОС реалізовані механізми, що дають можливість користувачу представляти і сприймати мережу у видгляді традиційного однопроцесорного комп'ютера.

Структура мережної операційної системи

Мережна операційна система є основою будь-якої обчислювальної мережі. Кожен комп'ютер у мережі значною мірою автономний, тому під мережною операційною системою в широкому змісті розуміють сукупність операційних систем окремих комп'ютерів, що взаємодіють з метою обміну повідомленнями і поділу ресурсів за єдиними правилами - **протоколами**. У вузькому значенні **мережна ОС** - це операційна система окремого комп'ютера, що забезпечує йому можливість працювати в мережі.

Рис. 2.4. Структура мережної ОС

У мережній операційній системі окремої машини можна виділити кілька частин (Рис 2.4):

- **Засоби керування локальними ресурсами комп'ютера**: функції розподілу оперативної пам'яті між процесами, планування і диспетчеризації процесів, керування процесорами в мультипроцесорних машинах, керування периферійними пристроями й інші функції керування ресурсами локальних ОС.

Засоби надання власних ресурсів і послуг у загальне користування - серверна частина ОС (сервер). Ці засоби забезпечують, наприклад, блокування файлів і записів, що необхідно для їхнього спільного використання; ведення довідників імен мережних ресурсів; обробку запитів віддаленого доступу до власної файлової системи і бази даних; керування чергами запитів віддалених користувачів до своїх периферійних пристроїв.

Засоби запиту доступу до віддалених ресурсів і послуг і їхнього використання - клієнтська частина ОС (редиректор). Ця частина виконує розпізнавання і перенаправлення у мережу запитів до віддалених ресурсів від додатків і користувачів, при цьому запит надходить від додатка в локальній формі, а передається в мережу в іншій формі, що відповідає вимогам сервера (**Маршалінг**). Клієнтська частина також здійснює прийом відповідей від серверів і перетворення їх у локальний формат (**Ремаршалінг**), так що для прикладної програми виконання локальних і віддалених запитів не розрізненяється.

Комунікаційні засоби ОС, за допомогою яких відбувається обмін повідомленнями в мережі. Ця частина забезпечує адресацію і буферизацію повідомлень, вибір маршруту передачі повідомлення по мережі, надійність передачі і т.п., тобто **є засобом транспортування повідомлень.**

У залежності від функцій, покладених на конкретний комп'ютер, в його операційній системі може бути відсутньою або клієнтська, або серверна частини.

На рис. 2.5. показана взаємодія мережних компонентів. Тут комп'ютер 1 виконує роль "чистого" клієнта, а комп'ютер 2 - роль "чистого" сервера, відповідно на першій машині відсутня серверна частина, а на другий - клієнтська. На малюнку окремо показано компонент клієнтської частини - редиректор. Саме редиректор перехоплює всі запити, що надходять від додатків, і аналізує їх. Якщо видано запит до ресурсу даного комп'ютера, то він переадресовується відповідній підсистемі локальної ОС, якщо ж це запит до віддаленого ресурсу, то він переправляється в мережу. При цьому клієнтська частина перетворює запит з локальної форми в мережний формат (**маршалінг**) і передає його транспортній підсистемі, що відповідає за доставку повідомлень зазначеному серверу. Серверна частина операційної системи комп'ютера 2 приймає запит, перетворює його (**ремаршалінг**) і передає для виконання своєї локальний ОС. Після того, як результат отриманий, сервер звертається до транспортної підсистеми і направляє

відповідь клієнту, що зробив запит. Клієнтська частина перетворить результат у відповідний формат і адресує його тому додатку, що зробив запит.

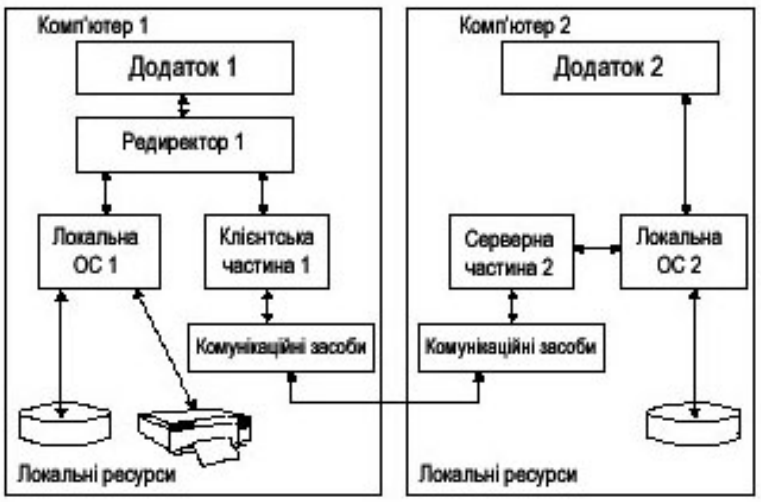


Рис. 2.5. Взаємодія компонентів операційної системи при взаємодії комп'ютерів

На практиці склалося кілька підходів до побудови мережних операційних систем (Рис 2.6).

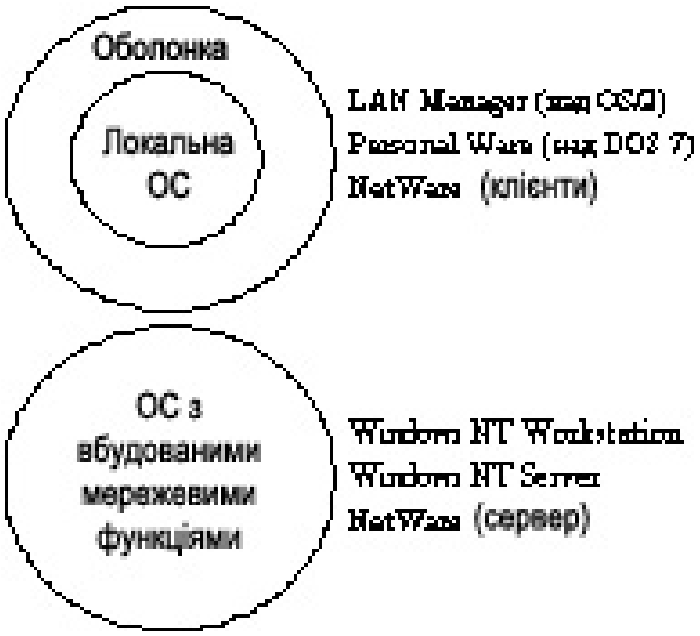


Рис. 2.6. Варіанти побудови мережних ОС

Перші мережні ОС були сукупністю існуючої локальної ОС і надбудованої над нею **мережної оболонки**. При цьому в локальну ОС вбудовувалось мінімум мережних функцій, необхідних для роботи мережної оболонки, що виконувала основні мережні функції.

Однак більш ефективним уявляється шлях розробки операційних систем, призначених для роботи в мережі від початку. Мережні функції в ОС такого типу глибоко *вбудовані* в основні модулі системи, що забезпечує їхню

логічну стрункість, простоту експлуатації і модифікації, а також високу продуктивність. Прикладом такої ОС є система Windows NT/2000 фірми Microsoft, що через вбудованість мережних засобів забезпечує більш високі показники продуктивності і захищеності інформації в порівнянні з мережною ОС LAN Manager тієї ж фірми (спільна розробка з IBM), що є надбудовою над локальною операційною системою OS/2.

Однорангові мережні ОС і ОС з виділеними серверами

У залежності від того, як розподілені функції між комп'ютерами мережі, мережні операційні системи, а отже, і мережі поділяються на **два класи**:

однорангові і дворангові. Останні частіше називають мережами з виділеними серверами.

Якщо комп'ютер надає свої ресурси іншим користувачам мережі, то він відіграє роль **сервера**. При цьому комп'ютер, що звертається до ресурсів іншої машини, є **клієнтом**. Як уже було сказано, комп'ютер, що працює в мережі, може виконувати функції клієнта, або сервера, або сполучати обидві ці функції.

Якщо виконання яких-небудь серверних функцій є основним призначенням комп'ютера (наприклад, надання файлів у загальне користування всім іншим користувачам мереж чи організація спільного використання факсу, чи надання всім користувачам мережі можливості запуску на даному комп'ютері своїх додатків), то такий комп'ютер називається виділеним **сервером**. У залежності від того, який ресурс сервера є поділюваним, він називається **файловим-сервером, факсом-сервером, принт-сервером, сервером додатків** і т.д**.**

Очевидно, що на **виділених серверах** бажано встановлювати ОС, спеціально оптимізовані для виконання тих чи інших серверних функцій. Тому в мережах з виділеними серверами найчастіше використовуються мережні операційні системи, до складу яких входить ОС декількох варіантів , що відрізняються можливостями серверних частин. (наприклад, Windows NT Server **(**для виділеного сервера**)** і Windows NT Workstation **(**для робочої станції**)**).

Наприклад, мережна ОС Novell NetWare має серверний варіант, оптимізований для роботи як файл-сервер, а також варіанти оболонок для робочих станцій з різними локальними ОС, причому ці оболонки виконують винятково функції клієнта.

Іншим прикладом ОС, орієнтованої на побудову мережі з виділеним сервером, є операційна система Windows NT. На відміну від NetWare, обидва варіанти даної мережний ОС - Windows NT Server (для виділеного сервера) і Windows NT Workstation (для робочої станції) - можуть підтримувати функції і клієнта і сервера. Але серверний варіант Windows NT має більше можливостей для надання ресурсів свого комп'ютера іншим користувачам мережі, тому що може виконувати більш широкий набір функцій, підтримує

більша кількість одночасних з'єднань із клієнтами, реалізує централізоване керування мережею, має більш розвинуті засоби захисту.

Виділений сервер не слід використовувати як комп'ютер для виконання поточних задач, не зв'язаних з його основним призначенням, тому що це може зменшити продуктивність його роботи як сервера. У зв'язку з цим **в ОС** Novell NetWare на серверній частині можливість виконання звичайних прикладних програм узагалі не була передбачена, тобто сервер не містив клієнтської частини, а на робочих станціях були відсутніми серверні компоненти. Однак в інших мережних ОС функціонування на виділеному сервері клієнтської частини цілком можливо. Наприклад, під керуванням Windows NT Server можуть запускатися звичайні програми локального користувача, що можуть зажадати виконання клієнтських функцій ОС з появою запитів до ресурсів інших комп'ютерів мережі. При цьому робочі станції, на яких встановлена ОС Windows NT Workstation, можуть виконувати функції невиділеного сервера.

Важливо зрозуміти, що незважаючи на те, що в мережі з виділеним сервером усі комп'ютери в загальному випадку можуть виконувати одночасно ролі і сервера, і клієнта, ця мережа функціонально не симетрична: апаратно і програмно в ній реалізовані два типи комп'ютерів - одні, у більш орієнтовані на виконання серверних функцій і працюють під керуванням спеціалізованих серверних ОС, а інші - в основному виконують клієнтські функції і працюють під керуванням відповідного цьому призначенню варіанта ОС. **Функціональна несиметричність**, як правило, викликає і несиметричність апаратури - для виділених серверів використовуються більш потужні комп'ютери з великими обсягами оперативної і зовнішньої пам'яті. Таким чином, функціональна несиметричність у мережах з виділеним сервером супроводжується несиметричністю операційних систем (спеціалізація ОС) і апаратною несиметричністю (спеціалізація комп'ютерів).

В одноранговых мережах усі комп'ютери рівні за правом доступу до ресурсів один одного. Кожен користувач може по своєму бажанню оголосити який-небудь ресурс свого комп'ютера поділюваним, після чого інші користувачі можуть його експлуатувати. У таких мережах на всіх комп'ютерах установлюється та сама ОС, що надає всім комп'ютерам у мережі потенційно рівні можливості.

В одноранговых мережах також може виникнути функціональна несиметричність: одні користувачі не бажають розділяти свої ресурси з іншими, в такому випадку їхні комп'ютери виконують роль клієнта, за іншими комп'ютерами адміністратор закріпив тільки функції по організації спільного використання ресурсів, а значить вони є серверами, у третьому випадку, коли локальний користувач не заперечує проти використання його ресурсів і сам не виключає можливості звертання до інших комп'ютерів, ОС, що встановлена на його комп'ютері, повинна містити і серверну, і клієнтську частини.

На відміну від мереж з виділеними серверами, в одноранговых мережах відсутня спеціалізація ОС у залежності від переважної функціональної

спрямованості - клієнта чи сервера. Усі варіації реалізуються засобами конфігурування того самого варіанта ОС.

Однорангові мережі простіше в організації й експлуатації, однак вони застосовуються в основному для об'єднання невеликих груп користувачів, що не висловлюють великих вимог до обсягів збереженої інформації, її захищеності від несанкціонованого доступу і до швидкості доступу. При підвищених вимогах до цих характеристик більш придатними є дворангові мережі, де сервер краще вирішує задачу обслуговування користувачів своїми ресурсами, тому що його апаратура і мережна операційна система спеціально спроектовані для цієї мети.

ОС для робочих груп і ОС для мереж масштабу підприємства

Мережні операційні системи мають різні властивості в залежності від того, призначені вони для мереж масштабу робочої групи (відділу), для мереж масштабу кампуса чи для мереж масштабу підприємства.

- **Мережі відділів** - використовуються невеликою групою співробітників, що вирішують загальні задачі. Головною метою мережі відділу є поділ локальних ресурсів, таких як додатки, дані, принтери і модеми. Мережі відділів звичайно не розділяються на підмережі.
- **Мережі кампусів** - з'єднують кілька мереж відділів усередині окремого будинку чи всередині однієї території підприємства. Ці мережі є усе ще локальними мережами, хоча і можуть покривати територію в кілька квадратних кілометрів. Сервіси такої мережі включають взаємодію між мережами відділів, доступ до баз даних підприємства, доступ до факс-серверів і високошвидкісних принтерів.
- **Мережі підприємства (корпоративні мережі)** - поєднують усі комп'ютери всіх територій окремого підприємства. Вони можуть покривати місто, чи регіон навіть континент. У таких мережах користувачам надається доступ до інформації і додатків, що знаходиться в інших робочих групах, інших відділах, підрозділах і штаб-квартирах корпорації.

Головною задачею операційної системи, використовуваної в **мережі масштабу відділу**, є організація поділу ресурсів, таких як додатки, дані, лазерні принтери і, можливо, низькошвидкісні модеми. Зазвичай мережі відділів мають один чи два файлових сервери і не більш ніж 30 користувачів. Задачі керування на рівні відділу відносно прості. В обов’язки адміністратора входить:

- додавання нових користувачів;
- усунення простих відмов;
- інсталяція нових вузлів;
- встановлення нових версій програмного забезпечення.

Операційні системи мереж відділів добре відпрацьовані і різноманітні, як і самі мережі відділів, що вже давно застосовуються і досить налагоджені. Така мережа звичайно використовує одну чи максимум дві мережні ОС.

Найчастіше це мережа з виділеним сервером Windows NT/2000, чи ж однорангова мережа.

Користувачі й адміністратори мереж відділів незабаром усвідомлюють, що вони можуть поліпшити ефективність своєї роботи шляхом одержання доступу до інформації інших відділів свого підприємства. Якщо співробітник, що займається продажами, може одержати доступ до характеристик конкретного продукту і включити їх у презентацію, то ця інформація буде більш свіжої і буде впливати на покупців. Якщо відділ маркетингу може одержати доступ до характеристик продукту, що ще тільки розробляється інженерним відділом, то він може швидко підготувати маркетингові матеріали відразу ж після закінчення розробки.

Отже, наступним кроком в еволюції мереж **є** об'єднання локальних мереж декількох відділів у єдину мережу будинку чи групи будинків. Такі мережі називають **мережами кампусів**. Мережі кампусів можуть простягатися на кілька кілометрів, але при цьому глобальні з'єднання не вимагаються.

Операційна система, що працює в мережі кампуса, повинна забезпечувати для співробітників одних відділів доступ до деяких файлів і ресурсів мереж інших відділів. Послуги, надані ОС мереж кампусів, не обмежуються простим поділом файлів і принтерів, а часто надають доступ і до серверів інших типів, наприклад, до факс-серверів і до маршрутизаторів зовнішніх приєднань. Важливим сервісом, що надається операційними системами даного класу, є доступ до корпоративних баз даних, незалежно від того, чи розташовуються вони на серверах баз даних чи на мінікомп’ютерах.

Саме на рівні мережі кампуса починаються проблеми інтеграції. У загальному випадку, відділи уже вибрали для себе типи комп'ютерів, мережного устаткування і мережних операційних систем. Наприклад, інженерний відділ може використовувати операційну систему UNIX і мережне устаткування Ethernet, відділ продаж може використовувати операційні середовища Novell і устаткування Token Ring. Дуже часто мережа кампуса з'єднує різнорідні комп'ютерні системи, у той час як мережі відділів використовують однотипні комп'ютери.

Корпоративна мережа з'єднує мережі всіх підрозділів підприємства, що знаходяться у загальному випадку на значних відстанях. Корпоративні мережі використовують глобальні зв'язки (WAN links - Wide-Area Network links) для з'єднання локальних мереж чи окремих комп'ютерів.

Користувачам корпоративних мереж потрібні всі ті ж додатки і послуги, що існують в мережах відділів і кампусов, плюс деякі додаткові додатки і послуги, наприклад, доступ до глобальних зв'язків. Коли ОС розробляється для локальної чи мережі робочої групи, то її головним обов'язком є поділ файлів і інших мережних ресурсів (зазвичай принтерів) між локально підключеними користувачами. Такий підхід не застосується для рівня підприємства. Поряд з базовими сервісами, зв'язаними з поділом файлів і принтерів, мережна ОС, що розробляється для корпорацій, повинна підтримувати більш широкий набір сервісів, що зазвичай складається з поштової служби, засобів колективної роботи, підтримки віддалених

користувачів, факс-сервісу, обробки голосових повідомлень, організації відеоконференцій і ін.
Крім того, багато методів, що існують, і підходів до рішення традиційних задач мереж менших масштабів для корпоративної мережі виявилися непридатними. На перший план вийшли такі задачі і проблеми, що у мережах робочих груп, відділів і навіть кампусів або мали другорядне значення, або взагалі не існували. Наприклад, найпростіша для невеликої мережі задача ведення облікової інформації про користувачів виросла в складну проблему для мережі масштабу підприємства. А використання глобальних зв'язків жадає від корпоративних ОС підтримки протоколів, що добре працюють на низькошвидкісних лініях, і відмовлення від деяких традиційно використовуваних протоколів (наприклад, тих, котрі активно використовують широкомовні повідомлення). Особливе значення набули задачі подолання гетерогенності - у мережі з'явилися численні шлюзи, що забезпечують погоджену роботу різних ОС і мережних системних додатків.

2.5. Управління процесами.

Поняття процесу.

Процес, якого іноді називають задачею (task), або роботою (job), є, неформально, програмою у виконанні. “Процес” це не теж саме, що “програма”. Ми помічаємо різницю між пасивною програмою, розміщеною на диску, і активним виконуваним (executing) процесом.

Багато людей можуть виконувати (run) одну програму; кожна виконувана копія кореспондується з певним процесом. Програма (сукупність команд, що складають програму,) є тільки частина процесу; процес містить також стан виконання (execution state).

Зверніть увагу на те, що є процеси користувачів і процеси ОС.

Отже, процес – це виконувана програма, що включає поточні значення лічильника команд, регістрів та змінних. Абстрактно, кожен процес має свій власний віртуальний процесор. Насправді ж, процесор перемикається з виконання одного процесу на інший, використовуючи певний алгоритм планування дла виявлення моменту перемикання, створюючи тим самим ілюзію паралельності. Оскільки час виконання частини процесу є дуже малим, то можна розглядяти набір процесів, що проходять псевдопаралельно. Таке переключення процесора від однієї програми до іншої називається мультипрограмуванням або багатозадачністю.

Створення/Завершення процесу

Підстави для створення процесу:

User logs on;
User starts a program;
OS creates process to provide a service,

(напр. демон прінтера для обслуговування прінтера);

Program starts another process.

Зазвичай при завантаженні ОС створюються процеси. Деякі з них високопріоритетні (взаємодія користувача з комп'ютером), інші – фонові (активізються при здійсненні якоїсь події). Фонові процеси, пов'язані з електронною поштою (активізується при прибутті листа), новинами, web-сторінками і т.п. називаються **демонами.**

Процеси можуть створюватись і пізніше, не тільки при завантаженні ОС, коли є можливість розбити задачу на кілька взаємозв'язаних процесів. Наприклад, потрібно знайти знайти певні дані в системі і обробити їх. Один процес може знаходити і уміщати дані в спільний буфер, а інший буде з цими даними працювати.

Новий процес утворюється так: поточний процес виконує системний запит на створення нового процесу. Системний запит вже містить мінімальні дані про програму, яку потрібно запустити в цьому процесі.

Адресні простори батьківського та дочірнього процесів у Windows відрізняються з самого початку. Проте в UNIX початковий вміст адресного простору дочірнього процесу є точною копією батьківського, хоча самі адресні простори різні.

Підстави для завершення процесу:

Normal completion;

Arithmetic error, or data misuse (e.g.,wrong type);

Invalid instruction execution;

Insufficient memory available, or memory bounds violation ;

Resource protection error;

I/O failure.

Після завершення процесу в UNIX виконується системний запит **exit**, а в Windows - системний запит **ExitProcess**.

Для UNIX має місце поняття „ієрархія процесів" (дерево процесів), тоді як у Windows всі процеси рівноправні.

Виконання процесів (Process Execution)

На Рис. 2.7. Відображено приклад концептуальної моделі квазіпаралельного виконання 4-х процесів у системі з одним процесором.

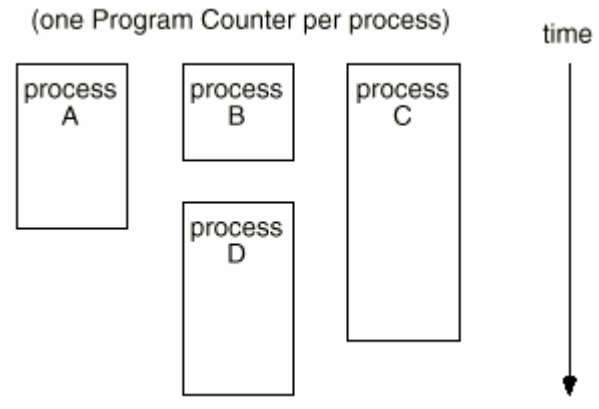


Рис. 2.7. Концептуальна модель виконання.

На Рис. 2.8. відображено актуальне (фактичне) почергове виконання тих же самих 4-х процесів.

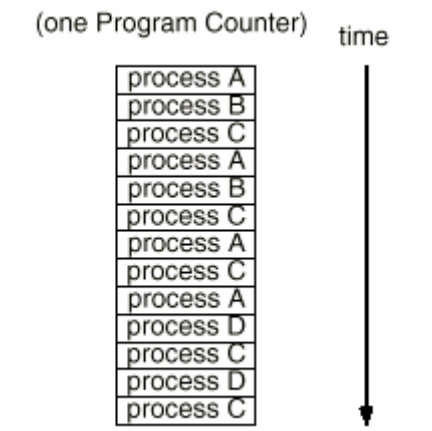


Рис. 2.8. Актуальна модель виконання.

Модель процесу на два стани

Ця модель процесу показує, коли процес виконується (running), або коли не виконується.

Діаграма переходів між станами подана на Рис. 2.9.

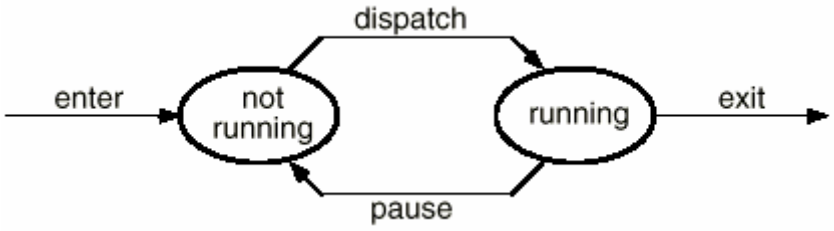


Рис. 2.9. Діаграма переходів.

Діаграма вставання у чергу подана на Рис. 2.10.

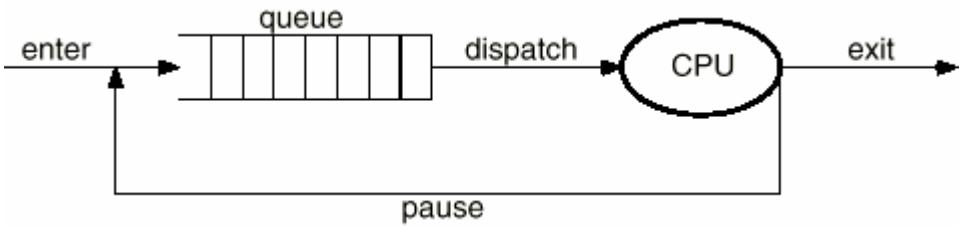


Рис. 2.10. Черга процесів, що чекають на виконання.

Планування використання процесора: round-robin – карусель.

У цьому алгоритмі планування черга іде по принципу: first-in, first-out (FIFO). Планувальник процесора бере процес з голови (head) черги, виконує (run) його протягом одного кванту часу (time slice), після цього ставить його знову у хвіст (tail) черги.

Переходи процесів у моделі процесів на два стани

Коли ОС створює новий процес, останній ініціалізується у **not-running стані** та чекає сприятливої можливості для виконання. В кінці кожного кванту часу планувальник процесора (scheduler) обирає новий процес для виконання (to run). Попередній процес спиняється (is paused) та переміщується з стану **running у** стан **not-running** (у хвіст черги). Новий процес (з голови черги) **диспетчеризується –** переміщується з **not-running** стану у **running** стан. Якщо процес, що виконується (running), завешує своє виконання (execution), він виходить, і планувальник процесора обирає інший. Якщо він не завершується, але його час вичерпано, процес переміщується у **not-running стан** знову, і планувальник процесора обирає новий процес для виконання.

Чекання на те, щоб щось сталось...

Можливі причини, чому процес насильно змінює стан виконання і повинен чекати:

чекає, поки користувач натисне наступну клавішу;

чекає, поки вивід з'явиться на екрані;

програма звернулась до читання файлу;

ОС вирішує, який блок диску читати, і тоді актуалізує читання інформації, що була запитана, у пам'ять;

броузер звертається до наступного гіперзв'язку, чекає, поки ОС визначить адресу запитуваних даних, прочитає пакети, покаже запитану web-сторінку.

П'яти – станова модель процесу.

Розглянемо тепер більш деталізовану модель процесу. ОС повинна визначати різницю між: процесами, що готові виконуватись і чекають для

прокрутки іншого кванту часу та процесами, які чекають, щоб щось сталось (операція ОС, апаратна подія, і т.п.).

Тому стан невиконання у двох – становій моделі має роздвоїтись між станом готовності і блокованим станом.

У п’яти – становій моделі процес може перебувати у стані:

running - зараз виконується;

ready - підготовлений до виконання;

blocked - чекає поки станеться деяка подія (завершення операції вводу\виводу, або ресурс стане доступним);

new - нормально створений;

exit - нормально завершений.

. Діаграма переходів у п’ятистановій моделі процесу подана на Рис. 2.11.

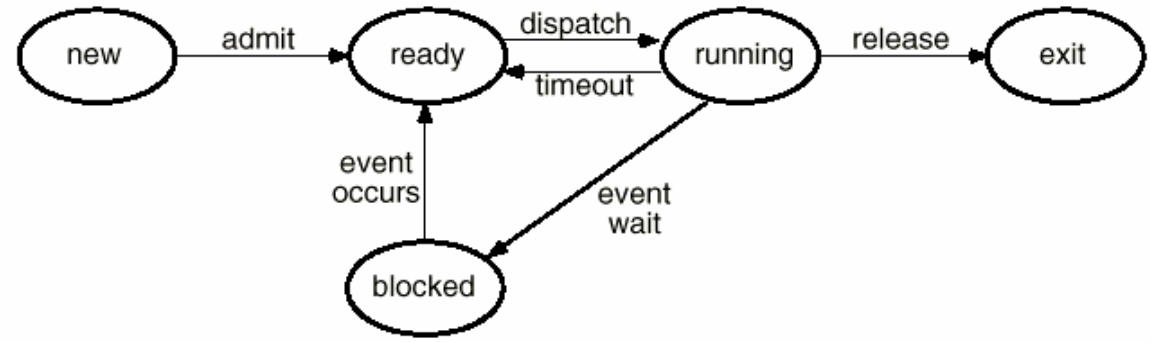


Рис. 2.11. Діаграма переходів у п’ятистановій моделі.

Переходи між станами у п’яти – становій моделі процесу

new - ready

Додавання до черги готових; процес може зараз бути розглянутим планувальником процесора.

ready - running

Планувальник процесора обирає процес для наступного виконання у відповідності з деяким алгоритмом планування (scheduling algorithm).

running - ready

Процес використав свій поточний квант часу.

running - blocked

Процес чекає на певну подію (завершення операції вводу\виводу і т. п.).

blocked - ready

Відбувається, коли подія, на яку процес чекав, сталася.

Інформація, що характеризує стан процесу.

Стан процесу - це інформація, необхідна для продовження виконання процесу якщо воно припинилось тимчасово.

Стан процесу містить (щонайменше):

команди програми;

статичні і динамічні дані програми;

стек викликів процедур програми;

вміст регістрів загального призначення;

вміст лічильника команд – адресу наступної команди, що буде виконуватись;

вміст вказівника стеку (Stack Pointer (SP));

вміст слова стану програми (Program Status Word (PSW)) - статус переривання, коди умов, і т. п.;

ресурси, що використовуються ОС (такі, як пам'ять, відкриті файли, зв'язки з іншими програмами);

визначальну (Accounting) інформацію.

Блок управління процесом (Process Control Block (PCB))

Для кожного процесу ОС підтримує Process Control Block (PCB), структура даних якого репрезентує процес і його стан:

Process id number; User id of owner;

Memory space (static, dynamic);

Program Counter, Stack Pointer, general purpose registers;

Process state (running, not-running, etc.);

CPU scheduling information (e.g., priority);

List of open files;

I/O states, I/O in progress;

Pointers into CPU scheduler's state queues (e.g., the waiting queue)

Перемикання контексту

Зупинка одного процесу і запуск іншого називається перемиканням контексту. Коли ОС зупиняє процес, він запам'ятовує апаратні регістри (PC, SP, etc.) та іншу інформацію стану у process' PCB. Коли ОС готовий виконувати процес, що чекає він завантажує апаратні регістри (PC, SP, etc.) значеннями, що запам'ятовані у новому process' PCB, і відновлює іншу інформацію стану.

Здійснення перемикання контексту є відносно дорога операція. Однак, системи розділення часу (time-sharing) можуть робити 100–1000 перемикань контексту у секунду

Створення процесу в UNIX

В операційній системі Unix один процес може створити інший процес, якщо зробити деяку роботу для цього.

Оригінальний процес називається parent, новий процес називається child; child є (майже) ідентичною копією parent (ті ж команди, ті ж дані і т. п.).

Parent може або чекати, поки child завершиться, або продовжувати виконання паралельно (concurrently) з child.

В Unix parent-процес створює child-процес, використовуючи системний виклик **fork( )**, причому:

у child-процес, fork( ) повертає (returns) 0;

у parent-процес, fork( ) повертає "process id" нового child.

Child звичайно використовує системний виклик **exec( )** для запуску зовсім іншої програми.

Наведемо типовий приклад створення процесу в UNIX.

```
#include <stdio.h>

void main(int argc, char *argv[])
{
int pid;

        /* fork another process */
        pid = fork( );

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);

                printf("Child Complete\n");
                exit(0);
        }
}
```

Модель процесів в UNIX.

Розглянемо модель процесів у Unix, яка стала вже класичною. У цій моделі стан **Running** роздвоюється на стани **User Running** (if in user mode) та **Kernel Running** (if in kernel mode). У зв'язку з застосуванням системи віртуальної пам'яті стан готовності роздвоюється на **Ready to Run, in Memory** та **Ready to Run, Swapped**, а стан **blocked** розділяється на стани **Asleep in Memory** та **Sleep, Swapped.**

На Рис. наведено діаграму переходів між станами процесу у Unix з моменту створення процесу з допомогою виклику fork() до завершення його виконання викликом exit().

У Unix процес може перебувати в одному з 9-ти станів:

1 - **User Running** - зараз виконується у режимі користувача;

2 - **Kernel Running** - зараз виконується у режимі ядра;

3 - **Ready to Run, in Memory** – готовий до виконання, у пам'яті;

4 - **Asleep in Memory -** чекає поки станеться деяка подія (завершення операції вводу\виводу, або ресурс стане доступним), знаходячись в оперативній пам'яті;

5 - **Ready to Run, Swapped -** – готовий до виконання, відкачаний;

6 - **Sleep, Swapped -** чекає поки станеться деяка подія, відкачаний;

7 – **Preempted** – процес «витеснено»;

8 – **Created** - нормально створений;

9 **– Zombie** – процес завершений, але залишається у системі, поки parent – процес не отримає код завершення процесу.

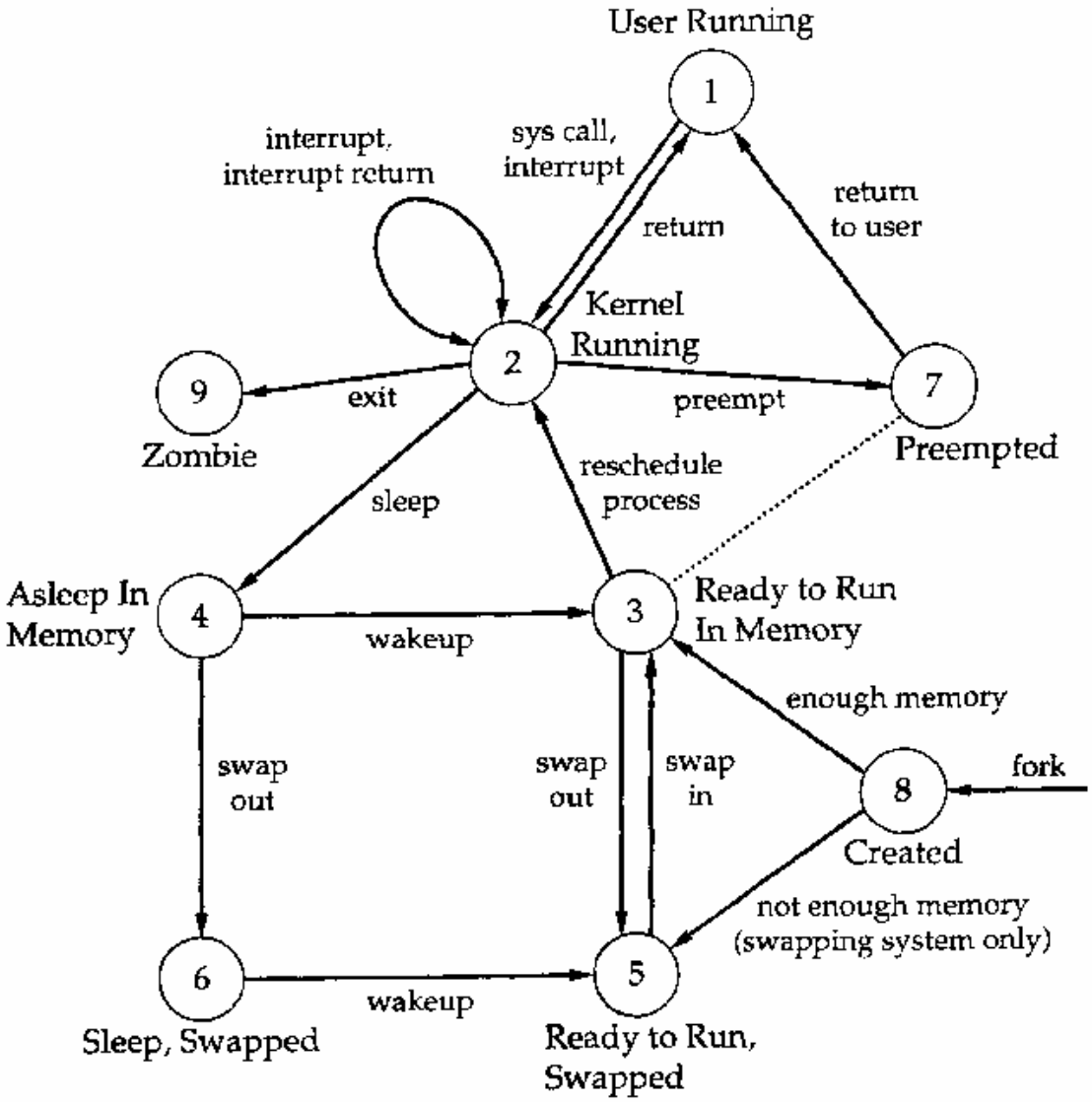


FIGURE 3.16 UNIX process state transition diagram [BACH86]

Figure from *Operating Systems*, 2nd edition, Stallings, Prentice Hall, 1995

Original diagram from *The Design of the UNIX Operating System*, M. Bach, Prentice Hall, 1986

Рис. 2.12. Діаграма станів процесу у Unix.

На Рис. 2.12. Показано, що відбувається старт процесу у стан **Created**, перехід до: **Ready to Run, in Memory** або до **Ready to Run, Swapped** (Out), якщо у пам'яті немає місця для нового процесу. **Ready to Run, in Memory** є по суті таким же станом, як **Preempted** (пунктирна лінія). **Preempted** означає, що процес повертається до user mode, але ядро переключилось до іншого процесу (another process instead).

Коли виконується планування, відбувається перехід до **User Running** (if in user mode) або **Kernel Running** (if in kernel mode), перехід до **Asleep in**

Memory коли очікується деяка подія, перехід до **Ready to Run, in Memory,** коли вона трапляється, перехід до **Sleep, Swapped** у випадку свопінгу.

Розподілений простір процесів.

Поняття «дерево процесів» може бути узагальнено на розподілені системи, у яких група комп'ютерів, що з'єднується у кластер у так званому Beowulf – стилі.

Кластер Beowulf'у це сукупність комп'ютерів, поєднаних в окрему локальну мережу, що можуть працювати як один великий комп'ютер, що виконує паралельні обчислення (Рис. 2.13.). Це досягається використанням програмного забезпечення, що втілює передачу повідомлень між копіями однієї програми, яка виконується на кожному з комп'ютерів ("вузлів") мережі. Кожен вузол працює над окремим фрагментом задачі, і коли виконує, то результат посилається головному комп'ютеру (master), який збирає результати. Передача повідомлень може здійснюватись на практично будь-яких комп'ютерах, поєднаних в мережу, якщо вони можуть спілкуватись між собою. Відрізняє кластер Beowulf від інших кластерів те, що у Beowulf'у власна локальна мережа для спілкування вузлів і всі вузли (окрім головного) не є звичайними робочими станціями. Вони повністю призначені для того, щоб бути виконуючим вузлом кластеру.

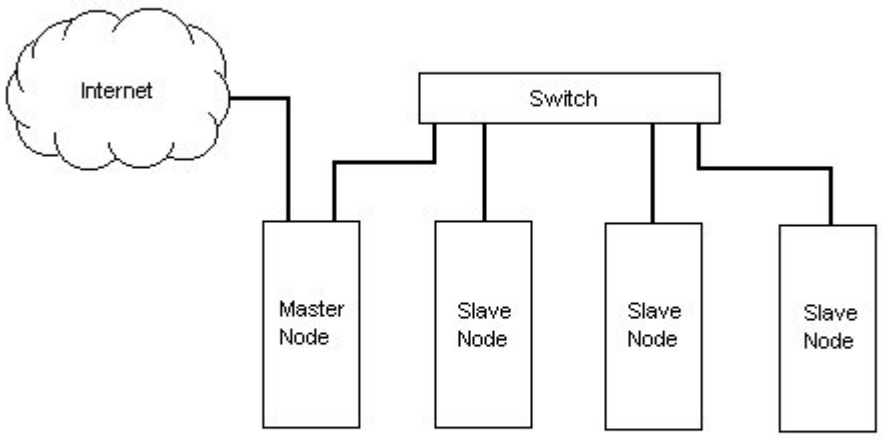


Рис. 2.13. Топологія кластеру Beowulf.

Виконуючі вузли можна сконфігурувати по-іншому з точки зору безпеки, ніж звичайні робочі станції, оскільки вони у своїй власній мережі. Завдяки цьому спілкування між вузлами пришвидшується, а додаткова робота між ними зменшується. Це також дозволяє кластеру здаватися одним комп'ютером для користувачів. Користувачам лише потрібно ввійти на головний вузол, для того, щоб послуговуватись цілим кластером. На цьому гголовному вузлі містяться всі скриньки користувачів, компілятори, диски для зберігання інформації та програмне забезпечення для передачі повідомлень, що використовується у кластері.

Розподілений простір процесів Beowulf'у (BProc) – це модифікація ядра Linux, яка адресує утворення процесу та управління ним за допомогою надання одного і того ж системного відображення для всіх процесів в кластері.

BProc розширює існуючу інфраструктуру управління процесами в Unix\Linux, щоб залучити виконання процесів на інших машинах.

Це означає, що всі існуючі утиліти управління процесами в Unix\Linux працюють для всіх паралельних завдань. Користувачі можуть побачити стан всіх паралельних процесів за допомогою команди “ps” на клієнтській машині. Припинення виконання всіх процесів таке ж легке, як посилання сигналу групі процесів або виконання стандартної команди Unix\Linux “killall”.

У кластері BProc, є одна головна (Master) машина і багато залежних (Slave) машин, які отримують і виконують процеси від головного комп’ютера. Всі процеси, розподілені між залежними машинами, видимі на головній так само, як і будь-який інший процес створений безпосередньо на головній машині. Існуючі утиліти візуалізації процесів в Unix\Linux показують віддалені процеси без модифікації. Сигнали передаються без змін до віддалених процесів. Звичайні зв'язки типу parent - child між процесами підтримуються незважаючи на можливе розташування процесів у різних вузлах кластеру.

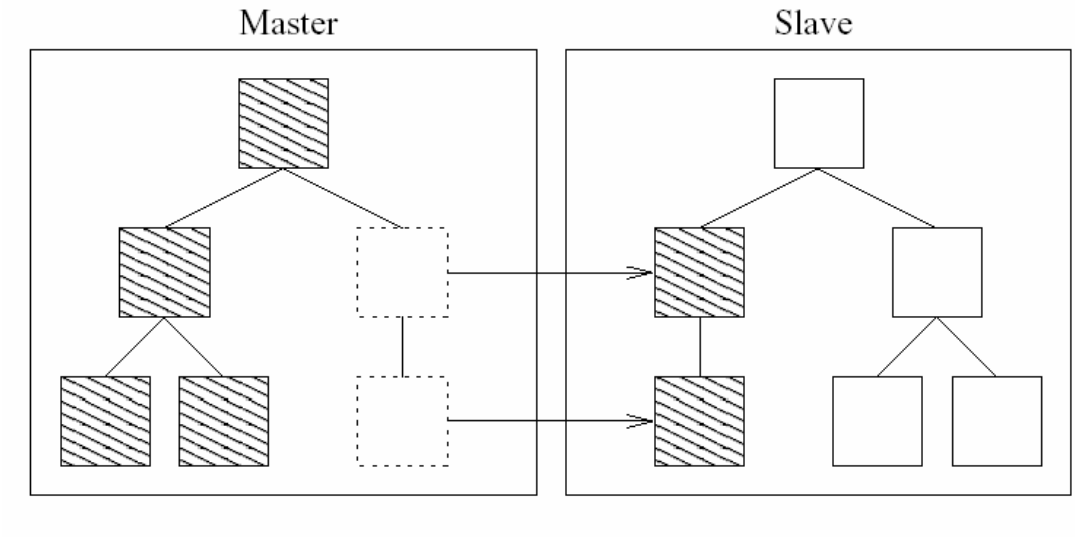


Рис. 2.14. Приклад дерева процесу, яке охоплює дві машини.

На Рис. 2.14. Подано приклад дерева процесу, яке охоплює дві машини. Затінені процеси на дублюючій машині існують в просторі процесів головної машини. Два відмічені пунктиром блоки в дереві

процесів головної машини є процеси-невидимки, які використовуються для резервування місця для відповідних віддалених процесів, які виконуються на дублюючому вузлі.

2.7. Планування навантаження центрального процесора (CPU Scheduling).

Довготерміновий планувальник (планувальник робіт – job scheduler)

Планувальник цього типу обирає роботу з підготовлених робіт (from spooled jobs), і завантажує її у пам'ять (loads). Виконання переривається тільки тоді, коли процес залишає систему. Управління до певної міри мультипрограмне.

У більшості сучасних систем розділення часу такий планувальник реально не існує.

Середньомермінове планування

У системах розділення часу, на віміну від довготермінового планування використовується для можливсті тимчасово видаляти (swap) процеси з пам'яті.

Короткотерміновий планувальник(CPU scheduler)

Виконується швидко, біля сотні разів на секунду. Працює, коли:
процес створюється або завершується;
процес перемикається з виконання на блокування;
виникає переривання;
процес обирається процес серед готових для виконання і виділяється CPU для цього процесу.

Алгоритми планування.

Планувальник процесора (the CPU scheduler or the dispatcher or short-term scheduler) обирає процес з черги готових (ready) і відсилає (dispatch) його на виконання (running) у процесорі.

Для розробки алгоритмів роботи планувальника процесора прийняті наступні спрощуючі припущення про характер (behavior) виконання процесів**:**

- є тільки один процес на кожного користувача;
- є тільки один потік управління на кожний процес;
- процеси незалежні, і конкурують за ресурси (включаючи CPU);
- процес виконується у пакетному (burst) циклі:
  - виконує деякий час обчислення у процесорі (CPU);
  - після цього виконує деякий ввод\вивід (I/O);
  - продовжує ці дві дії повторно.

Робиться також припущення, що є два типи процесів:

стримувані процесором – великий об'єм обчислень (long CPU burst) і дуже малий ввод\вивід;
стримувані вводом\виводом – великий ввод\вивід і дуже малі обчислення (short CPU burst).

Цілі планування завантаження центрального процесора.

Планувальник процесора мусить вирішувати, як довго процес має виконуватись та у якому порядку процеси будуть виконуватись.

Орієнтовані на користувача цілі політики планування:

- мінімізувати середній час реакції (час від отримання запиту до початку відповіді);
- мінімізувати час, коли максимальна кількість інтерактивних користувачів отримують адекватні відповіді;
- мінімізувати час обороту (час від старту процесу до завершення тобто час виконання плюс час чекання);
- мінімізувати варіацію середнього часу відповіді: прогнозованість є важливою;
- забезпечити, щоб процес завжди виконувався деяку кількість часу незалежно від завантаження системи.

Системно – орієнтовані цілі політики планування:

- максимізація пропускної здатності (кількості процесів, що закінчуються в одиницю часу);
- максимізація використання процесора (процент часу, протягом якого процесор зайнятий - CPU is busy);
- справедливість – у відсутності впливу користувача або ОС, процеси повинні оброблятись однаково, і ні один процес не повинен терпіти відмови від обслуговування (може бути зменшена справедливість у порядку мінімізації середнього часу відповіді!);
- баланс ресурсів – економія усіх ресурсів системи (CPU, memory, disk, I/O);
- сприяння процесам, що не завантажують напружених ресурсів (stressed resources).

Порівняння перериванного (Preemptive) інеперериванного (Non-Preemptive) алгоритмів планування.

Неперериване (Non-preemptive) планування – має місце, якщо планувальник працює тільки тоді,коли процес завершується або перемикається з стану running у стан blocked.

Перериване (Preemptive) планування – планувальник може працювати у (майже) будь який час. Планування виконується, коли процес завершується або перемикається з стану running у стан blocked, а також коли:

- процес створюється;
- блокований процес повертається у стан готовності;
- трапляється переривання по таймеру.

При перериванному плануванні більше накладних витрат, але можливе запобігання монополізації CPU довгими процесами.

Зауважимо, що ядро ОС повинно **не перериватись** коли воно обслуговує системний виклик (наприклад читає файл) або перебуває у невизначеному стані.

Алгоритм планування: Першим прийшов – першим обслуговується (First-Come-First-Served - FCFS).

Інші назви цього алгоритму планування: First-In-First-Out (FIFO), Run-Until-Done (виконується до закінчення).

Політика: обирається процес з черги готових у порядку їх прибуття, і виконується цей процес non-preemptively (неперериванно, без витеснення).

Ранні FCFS планувальники були надмірно непереривaнними: процес не поступався процесором поки не закінчувався, навіть коли він виконував ввод\вивід. Зараз непереривaність означає, що планувальник обирає інший процес тільки тоді, коли перший закінчується (exit) або блокується (blocked).

Дамо оцінку алгоритму FCFS. Цей алгоритм непереривaний (Non-preemptive). Час відгуку – повільний, якщо є велика варіація у часі виконання процесів. Якщо один процес стїть у черзі перед багатьма короткими процесами, він обирається для виконання, а короткі процеси мають чекати довгий час. Завантаження CPU and I/O device низьке. Пропускній здатності – не надається значення. Не виконується прицип справедливості – у невигідному положенні короткі процеси та критичні до вводу\виводу (I/O bound) процеси. Зависання – неможливе. Накладні витрати – мінімальні.

Алгоритм планування Round-Robin (карусель).

Політика:

встановлення фіксованого кванту часу (time slice, time quantum);

обирання процесу з голови черги готових;

виконання цього процесу протягом щонайбільше одного кванту часу, і якщо він не завершиться, додавання його у хвіст черги готових;

якщо цей процес завершиться або заблокується до закінчення кванту часу, обрання іншого процесу з голови черги готових і виконання цього процесу протягом щонайбільше одного кванту часу .

Для впровадження алгоритму планування Round-Robin використовується апаратний таймер, що періодично видає переривання (interrupts) та FIFO черга готових (add to tail, take from head);

Дамо оцінку алгоритму Round-Robin.

Він перериванний (Preemptive - at end of time slice). Час відгуку – хороший для коротких процесів.

Довгі процеси можуть мати чекання n* q одиниць часу до іншого кванту часу, де:

n = кількість інших процесів;

q = величина кванту часу.

Пропускна здатність – залежить від кванту часу:

дуже малий квант – дуже багато перемикань контексту;

дуже великий квант – наближається до FCFS.

Справедливість неповна– у невигідному положенні критичні до вводу\виводу (I/O-bound) процеси (може не використати повний квант часу).

Зависання – неможливе. Накладні витрати – низькі.

Алгоритм планування «Найкоротша робота – першою» (Shortest-Job-First - SJF).

Інша назва: Найкоротший процес – наступним (Shortest-Process-Next).

Політика:

Обирається для виконання процес, що має найменший наступний пакет операцій у процесорі. Shortest-Job-First є приоритетним плануванням, де пріоритет змінюється пропорційно довжині наступного CPU burst.

Планування по пріоритетам (Priority Scheduling).

Політика:

з кожним процесом асоціюється пріоритет, який може бути зовнішньо визначений, основаним на значимості, грошах, політиці і т. п., або внутрішньо визначеним, основаним на вимогах до пам’яті, файлів, співвідношенні у потребах процесору та вводу\виводу;

обирається процес, що має найвищий пріоритет і виконується у процесорі одним з двох способів: перериванно (preemptively), або неперериванно (non-preemptively).

Оцінка:

зависання – можливе для низько пріоритетних процесів;

можна ухилитись від старіння процесів: збільшувати пріоритет якщо вони марно витрачають час у системі.

Багаторівневі черги планування (Multilevel Queue Scheduling).

Політика:

використовується кілька черг готових, і з різними чергами асоциюються різні пріоритети;

обирається процес, що зайняв чергу, що має найвищий пріоритет, і виконується у процесорі одним з двох способів: перериванно (preemptively), або неперериванно (non-preemptively).

Новий процес завжди призначається до однієї з особливих черг:

Foreground, background, System, interactive, editing, computing; кожна черга може мати свою політику планування.

Багаторівневі черги планування з зворотнім зв'язком (Multilevel Feedback Queue Scheduling).

Політика:

використовується кілька черг готових, і з різними чергами асоціюються різні пріоритети;

обирається процес, що має найвищий пріоритет і виконується у процесорі одним з двох способів: перериванно (preemptively), або неперериванно (non-preemptively).

кожна черга може мати свою політику планування;

планувальник може переміщати процеси між чергами;

стартує завжди процес з найбільш приоритетної черги; коли він закінчить свій CPU burst, він переміщається у чергу з нижчим пріоритетом;

запобігання старінню – переміщення старіших процесів до черги з вищим пріоритетом;

зворотній зв'язок = використання минулого для передбачення майбутнього – сприяння роботам, які не використовували процесора.

Планування процесора у UNIX з використанням Multilevel Feedback Queue Scheduling

Політика:

кілька черг, кожна з яких з своїм значенням пріоритету (low value = high priority);

процеси ядра мають негативні значення пріоритету; процеси користувачів (роблять обчислення) мають позитивні значення пріоритету;

обирається процес з черги з найвищим пріоритетом, і виконується цей процес **перериванно**, використовуючи таймер (типовий квант часу біля 100ms);

у кожній черзі застосовується алгоритм планування **Round-robin**;

процеси переміщуються між чергами у залежності від часу їх знаходження у системі.

Планування в сердовищі JAVA.

Середовище JAVA підтримує досить простий алгоритм планування. Цей алгоритм базується на пріоритеті Thread-ів (підпроцесів, потоків) відносно інших потоків. Планування, або встановлення приоритетів, реалізується в JAVA за допомогою методу setPriority(). Цій метод встановлює приоритет підпроцесу, який задається цілим значенням параметра, що передається методу. В класі Thread є декілька приоритетів-констант: MIN_PRIORITY, NORM_PRIORITY і MAX_PRIORITY. Вони відповідають значенням відповідно 1, 5, 10. більшість користувацьких програм повинно виконуватись на рівні NORM_PRIORITY плюс-мінус 1. Пріоритет фонових завдань, наприклад, мережевого вводу-виведення чи перемальовки екрану, треба встановлювати в MIN_PRIORITY. Запуск підпроцесів на рівні MAX_PRIORITY потребує обережності. Якщо в підпроцесах з таким рівнем приоритету відсутні виклики sleep чи yield, середовище може перестати реагувати на зовнішні виклики.

Виклик методу **yield** приводить до того, що система перемикає виконання підпроцесу на наступний підпроцес.

При виклику методу sleep(int n) система блокує підпроцес, що виконується, на n мілісекунд.

В якості прикладу наведемо програму з двома підпроцесами різного пріоритету, що не ведуть себе однаково на різних платформах. Пріоритет одного з підпроцесів з допомогою виклику setPriority встановлюється на два рівня вище від Thread.NORM_PRIORITY, тобто, того, що за замовченням. У другого підпроцесу встановлюється пріоритет на два рівня нижче. Підпроцеси запускаються і працюють на протязі 10 секунд. Кожний виконує цикл, в якому збільшує значення лічильника. Через 10 секунд після запуску основний підпроцес зупиняє їх роботу, присвоює умові завершення цикла while значення true і виводить значення лічильників, показуючи, скільки ітерацій встиг виконати кожен з підпроцесів.

```
class Clicker implements Runnable {
int click = 0;
private Thread t;
private boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
} }
public void stop() {
running = false; }
public void start() {
t.start();
```

```
} }
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try Thread.sleep(-10000) {
}
catch (Exception e) {
}
lo.stop();
hi.stop();
System.out.println(lo.click + " vs. " + hi.click);
} }
```

Виконаємо програму і переконаймося, що на виконання менш приоритетного процесу йде на 25 відсотків менше часу.

```
C:\>java HiLoPri


304300 vs. 406666
```

Планування для мультипроцесорів зі спільною пам’яттю.

У мультипроцесорних системах зі спільною пам’яттю для кожного процесора може підтримуватись локальна черга виконання модулів управління призначеними йому периферійними пристроями. Крім того у системі, як правило, є одна або декілька глобальних черг. Згідно з однією з можливих стратегій планування, коли звільняється черговий процесор, система спочатку переглядає його локальну чергу, і якщо в ній немає готових процесів, обирає процес з глобальних черг. Причому для множини процесів, що є складовими однієї паралельної програми, указується інформація для планування - кількість процесів, що можуть одночасно виконуватись, а також їх відносні приоритети.

Якщо планувальник просто призначає перший у черзі процес процесору, що звільнився, то ігнорується можливість того, що процес міг недавно виконуватись на певному процесорі і у кеші. Так, у випадку переривання по «відмові сторінки» (описаному у розділі про віртуальну пам’ять) виконання процесу, безумовно краще продовжити на тому ж процесорі. Для цього можна дозволити процесору «чекати у стані зайнятості» (busy waiting). При цьому процес, що викликав переривання по «відмові сторінки», залишається призначеним процесору – його блокування та

перемикання контексту не відбувається. Такий процес звичайно виконує пустий цикл в очікуванні переривання, що сповіщає про завершення «підкачки» стрінки.

Ще одним фактором, який треба враховувати, є відношення між групами процесів. Якщо певні процеси тісно взаємодіють один з одним, краще всього їх паралельно виконувати на різних процесорах мультипроцесорної системи. Але у випадку, коли ці процеси використовують спільні дані, то, якщо вони виконуються на одному процесорі, то можуть користуватись одним кешом. Якщо ж їх призначати різним процесорам, то треба утримувати кеши у погодженому стані, що вимагає певних витрат.

Розподілене планування та міграція процесів. (Distributed scheduling and Process Migration).

При плануванні у централізованих системах ресурсом, що розподіляється, є CPU, а споживачем ресурсу – процес. Метою планування є - приписати кожен процес до якогось періоду часу CPU.
При плануванні у розподілених системах [28],[29] ресурсом є процесор / робоча станція а споживачем - обчислювальна задача (computation task).
Мета планування - приписати кожну обчислювальну задачу певному процесору, розподілити задачі по множині процесорів, та таким чином оптимізувати деякі витрати. Розподіл навантаження (load distribution) це прийняття рішення, які процеси переміщати від одного процесора до іншого та коли це робити.

Мотивації для розподілу навантаження

Маємо ситуацію, яка наочно відображена на Рис. 2.15

.

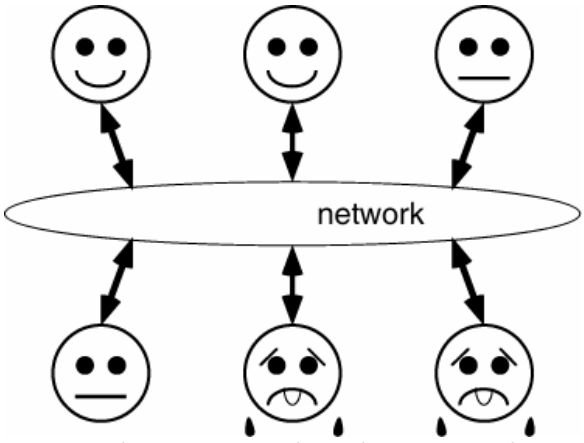


Рис. 2.15. Ситуація у розподіленій системі.

Хочемо забезпечити, щоб процеси обмінювались повідомленями типу представлених на Рис. 2.16. та відповідно реагували на них.

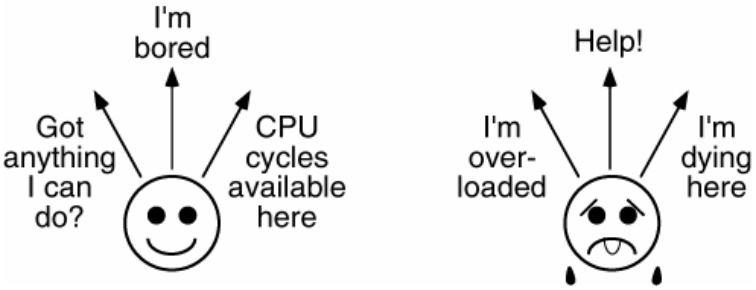


Рис. 2.16. Типи повідомлень

Перерозподіл навантаження може надати такі переваги:

зменшення часу відгуку для процесів шляхом переміщення процесів до легко навантажених вузлів;

прискорення індивідуальних робіт завдяки переходу процесів до більш швидких вузлів або розділення процесів;

збільшення пропускної здатності за рахунок досягнення балансу системного навантаження, тобто змішування процесів, пов'язаних інтенсивним використанням CPU, та процесів з великим вводом/ виводом;

підвищення ефективності використання ресурсів, шляхом переміщення процесів до вузла, де зберігаються ресурси;

зменшення трафіку у мережі за рахунок створення кластерів пов'язаних процесів на одному вузлі.

Міграція процесів

Міграція процесів – це переміщення процесів з їх поточного місця розташування (вузол джерела) до іншого вузла (вузол призначення):

Переміщення може відбуватись з перериванням (preemptive) – після того, як процес почався, або без переривання (non-preemptive) – до початку процесу.

Механіка міграції процесів:

вибір процесу, що буде мігрувати;

вибір вузла призначення;

передача процесу з вузла джерела до вузла призначення.

Для передачі треба заморозити процес на вузлі джерелі, та розпочати його знову на вузлі призначенні, передати адресний простір процесу, передавати повідомлення, призначені для процесу, підтримувати комунікацій з мігруючими процесами.

Механізм міграції процесів полягає у виконанні наступних кроків.

a. Заморозити та розпочати процес знову:
зберігати той самий ідентифікатор процесу після міграції
здійснити негайне блокування, якщо не виконується системний виклик, або якщо системний виклик виконується, але може перериватись;
чекати на завершення швидких операцій вводу/ виводу, але не чекати на завершення повільних операцій вводу/ виводу;
зберігати доріжки файлів (track of files), переключати на локальні файли вузла призначення, якщо це можливо.

b. Здійснити передачу адресного простору.
Тут можуть бути такі варіанти.
Тотальне замороження
У цьому варіанті необхідно зупинити виконання під час передачі.
Треба передати цілий стан процесу: регістри, інформація про планування, таблиці пам'яті, стани вводу/ виводу, ідентифікатор процесу, інформація про файли та ін.
Треба передати адресний простір, що включає: код, дані, стек, купу
Передача може зайняти багато часу!

Продовження виконання під час передачі
Можна продовжити виконання під час передачі адресного простору, а тоді заморозити процес та передати модифіковані у період передачі сторінки. Результатом буде зменшення загального часу переривання виконання процесу.

Передача за посиланням
Можна залишити адресний простір на вузлі джерелі, а передавати лише сторінки пам'яті, коли на них процес посилається при продовженні виконання на вузлі призначенні.

c. Передача повідомлень

Можуть бути три типи повідомлень, що передаються:

1) повідомлення, отримані на вузлі джерелі після виконання, зупиненого там, але перед тим, як виконання розпочалося на вузлі призначенні;

2) повідомлення, отримані на джерелі, після того, як виконання розпочалося на призначенні;

3) повідомлення, надіслані до процесу пізніше

d. Підтримка комунікацій з мігруючими процесами.
Тут можуть бути такі варіанти.

Перепосилання повідомлення

Можна або повернути або відкинути повідомлення типів 1 та 2, сподіваючись, що відправник повторно перешле їх пізніше

Механізм початкового сайту (origin site mechanism)

Повідомлення відправляються до сайту початкового джерела, який передає їх знову, як належить

Механізм відслідковування зв'язків

Повідомлення типу 1 є частиною міграції. Повідомлення типів 2 та 3 перенаправляються за новою адресою процесу, отриманою останнім у результаті міграції.

Міграція процесів у гетерогенній системі

Передача процесів у гетерогенній системі можлива до початку виконання процесу, якщо у вузлі призначенні є своя версія тієї ж самої програми, що і на вузлі джерелі, але для цього необхідно перетворити дані.

Байти та слова можуть потребувати перетворення з коду ASCII в код EBDCDIC та ін.

При представленні зовнішніх даних для передачі доцільно використовувати стандартне представлення. Є різні техніки для міграції порядку та мантиси чисел з плаваючою комою.

Однак, багато систем для узгодженості зараз використовують формат плаваючої коми, стандартизований IEEE:

одиночна точність = 32 біти (1 знак, 8 експонента, 23 мантиса);

подвійна точність = 64 біти (1 знак, 10 експонента, 53 мантиса).

Потрібно також мати справу з нескінченністю зі знаком та нулем зі знаком, якщо ці значення підтримуються одним або обома вузлами.

2.8. Нитки (потоки управління).

Два погляди на процеси

Процес може розглядатися двома способами. Як одиниця монопольного використання ресурсів процес має адресний простір, що містить програмний код і дані; процес може мати відкриті файли, може використовувати пристрої вводу-виводу, і т.д.

Процес може розглядатись також як одиниця планування. Планувальник CPU дисптечеризує (посилає на виконання CPU) у поточний момент часу один з готових для виконання процесів; із процесом пов'язані: значення лічильника команд PC, вказівника стеку SP, і інших регістрів.

Приблизно в 1988 році стало зрозуміло, що ці два прогяди на процес звичайно пов'язані, але так не повинно бути.

У багатьох недавніх операційних системах (UNIX, Windows NT), є дві незалежні сутності:

- **Процес** = одиниця монопольного використання ресурсів
- **Нитка** = одиниця планування

Процеси та нитки управління

Процес = одиниця монопольного використання ресурсів (unit of resource ownership).

Процес (що іноді називається великоваговим процесом) має:

адресний простір;
програмний код;
глобальні змінні, купу, стек;
OS ресурси (файли, пристрої вводу-виводу, і т.д.).

Нитка = одиниця планування (unit of scheduling).

Нитка (іноді називається маловаговим процесом) є просто послідовним потоком (stream) виконання всередині процесу.

На Рис. 2.17. показано приклад розміщення двох ниток управління в адресному просторі процесу.

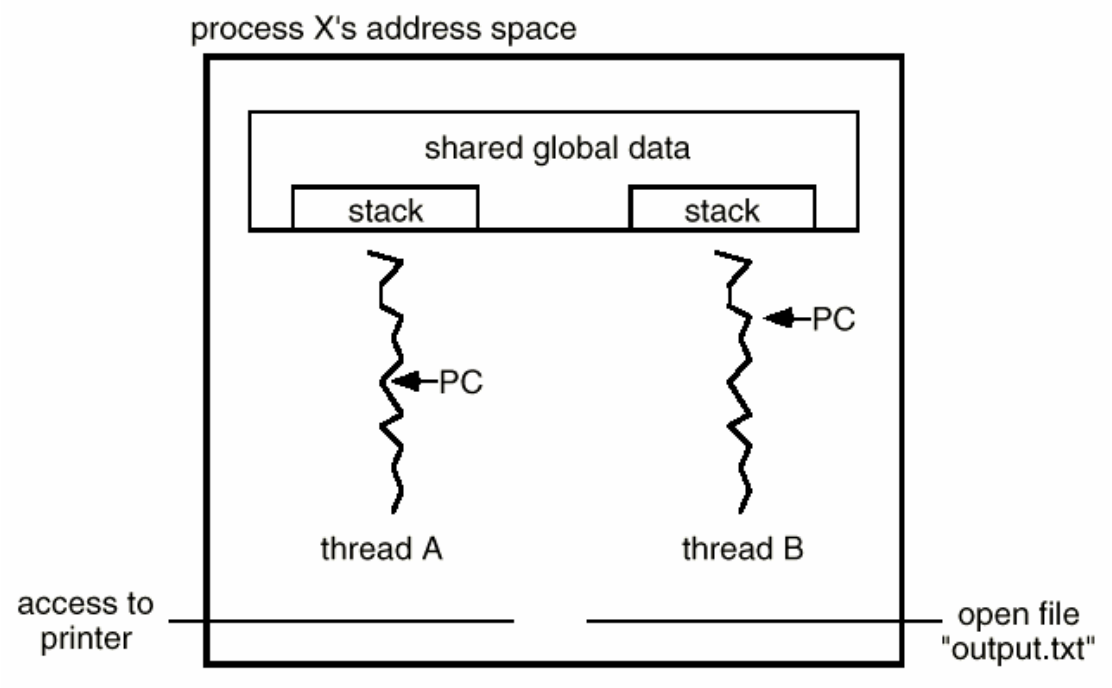


Рис. 2.17. Приклад розміщення двох ниток управління **thread A** та **thread B** в адресному просторі процесу **X**.

Нитка спільно використовує з іншими нитками адресний простір, програмний код, глобальні змінні, купу, OS ресурси (файли, пристрої вводу/виводу). Нитка має власні регістри, лічильник команд (PC), стек, покажчик вершини стека (SP). Нитка зв'язана з певним процесом. Процес усередині себе може мати багато ниток управління. Нитки можуть блокуватись, створювати дочірні нитки, і т.д.

Чому доцільно використовувати нитки?

Процес із множиною ниток створює великий сервер (наприклад, сервер прінтера) має один серверний процес та у його рамках багато ниток – «виконавців». Якщо одні нитки блокуються (наприклад, на читанні), інші можуть усе ще продовжувати виконання. Нитки можуть спільно використовувати загальні дані, і тому не повинні використовувати міжпроцессорний зв'язок. Створюючи нитки у рамках одного процесу, можна користуватися перевагою багатопроцесорних систем.

Нитки "дешеві"!

Нитки дешево створювати – вони потребують тільки стек і пам'ять для збереження значень регістрів, використовують дуже невелику кількість ресурсів, не потребують нового адресного простору, глобальних даних, програмного коду, або OS ресурсів. Перемикання контексту – швидке, треба зберігати / відновлювати тільки значення PC, SP, і регістрів

Але ... ніякого захисту між нитками!

Які види програм можуть бути багатопоточними?

Програми, що добре реалізуються як багатопоточні, це програми з багатьма незалежними задачами. Наприклад, відладчик повинен виконувати і контролювати програму, зберігати графічний інтерфейс активним, і відображати інтерактивну інспекцію даних та самозапис динамічних викликів.

Періодично повторювані числові задачі – велику проблему, типу, наприклад, погодного прогнозування, розділяють на менші фрагменти і призначають для кожного фрагменту окрему нитку

Програми, що важко реалізуються як багатопоточні

Це програми, що не потребують будь-якого мультипроцессорного опрацювання (99 % усіх програм), програми, що не потребують множини процесів.

Використання Ниток у Сервері

У Моделі **диспетчер - виконавець** нитка диспетчер одержує всі запити, вручає кожному виконавцю нитку, нитка виконавця опрацьовує запит. Нитки виконавців створюються динамічно або, коли сервер стартує, створюється фіксованого розміру пул виконавців.

У моделі групи **всі нитки – рівноцінні**; кожна нитка опрацьовує власні вхідні запити. Ця модель гарна для опрацювання множини типів запитів усередині одного сервера

У **конвеєрній** моделі перша нитка частково опрацьовує запит, потім вручає його другій нитці, що опрацьовує дещо більше, потім передає його до третьої нитки, і т.д.

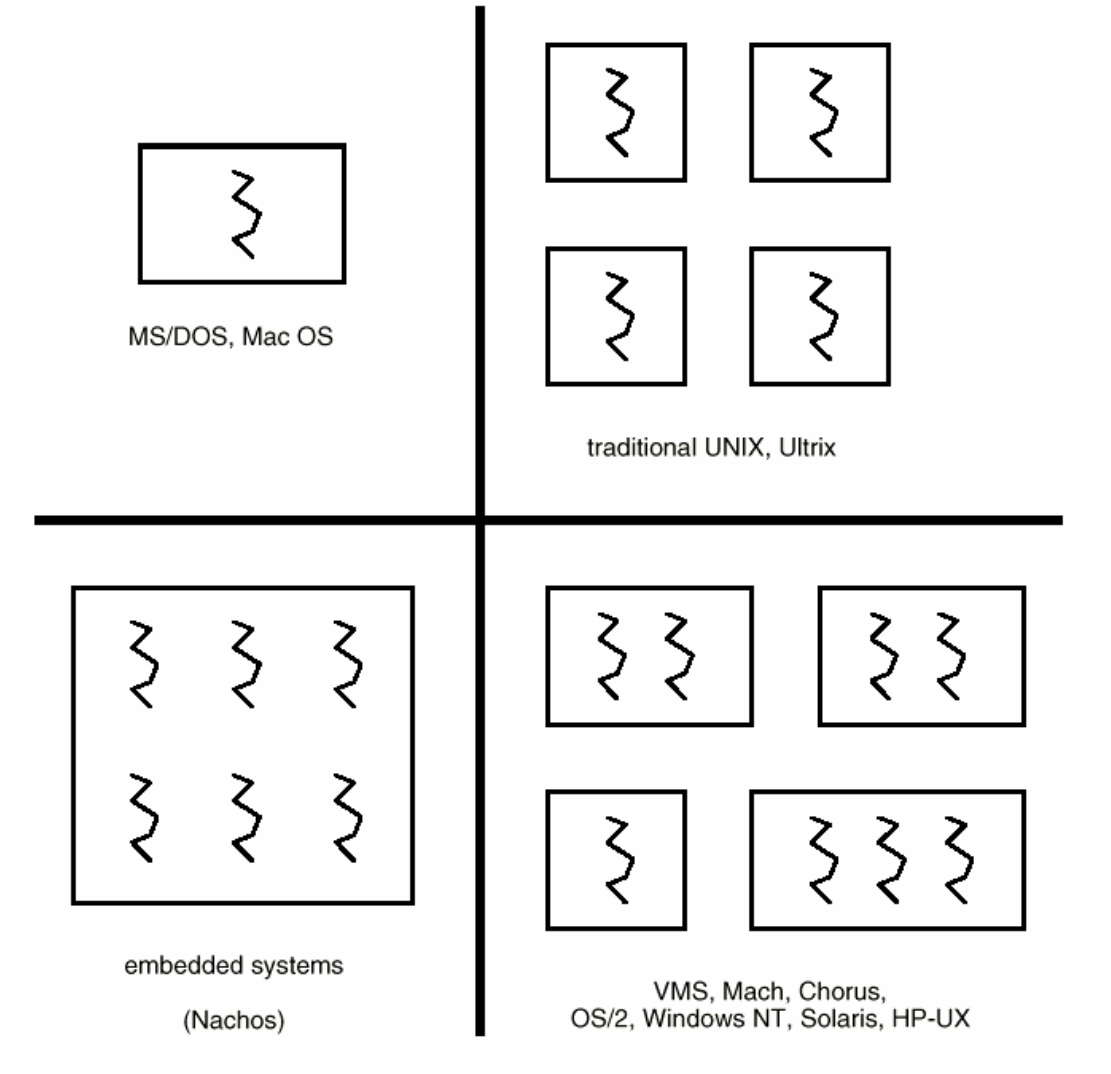


Рис. 2.18. Класифікація операційних систем по відношенню до ниток.

На Рис. 2.18. подана класифікація операційних систем по відношенню до ниток.

Нитки Рівня користувача

Нитки рівня користувача забезпечують бібліотеку функцій, щоб дозволити процесам користувачів створювати і управляти їхніми власними нитками. Вони не потребують модифікації ОС, мають простe представлення. Кожна нитка представлена просто PC, регістрами, стеком, і малим керуючим блоком, що повністю зберігається в адресному просторі процесу користувача.

Нитки рівня користувача мають просте керування. Створення нової нитки, перемикання між нитками, і синхронізація взаємодії між нитками – усе може бути виконане без втручання ядра. перемикання ниток не на багато дорожче, ніж виклик процедури. планування CPU (серед таких ниток) може бути реалізовано у відповідності з потребами алгоритму.

Нитки рівня ядра

При реалізації ниток на рівні ядра, ядро забезпечує системні виклики, щоб створювати нитки і керувати нитками. Ядро має повну інформацію щодо всіх ниток. Тому, наприклад, планувальник може дати процессу з 10 нитками більшу кількість часу, ніж процесу з тільки 1 ниткою. Це добре для додатків, що часто блокуються (наприклад, коли сервер опрацьовує часті межпроцессорні зв'язки).

Але операції з нитками на рівні ядра у 100 разів повільніші, ніж для ниток рівня користувача. Значні непродуктивні витрати, збільшується складність ядра – ядро повинно справлятися як з процесами, так і з нитками, і тому потребує повного керуючого блока нитки (TCB) для кожної нитки.

Потоки рівня користувача та рівня ядра.

У ряді сучасних ОС реалізована двохрівнева модель потоків (Рис. 2.19). До них належать Digital UNIX, Solaris, IRIX, HP-UX.

Потоки рівня користувача забезпечуються бібліотекою функцій що дозволяє процесам користувача створювати свої власні потоки та управляти ними. Одночасно операційна система підтримує і потоки рівня ядра, тобто у ядрі перебачена підтримка системного виклику для створення потоків та управління ними.

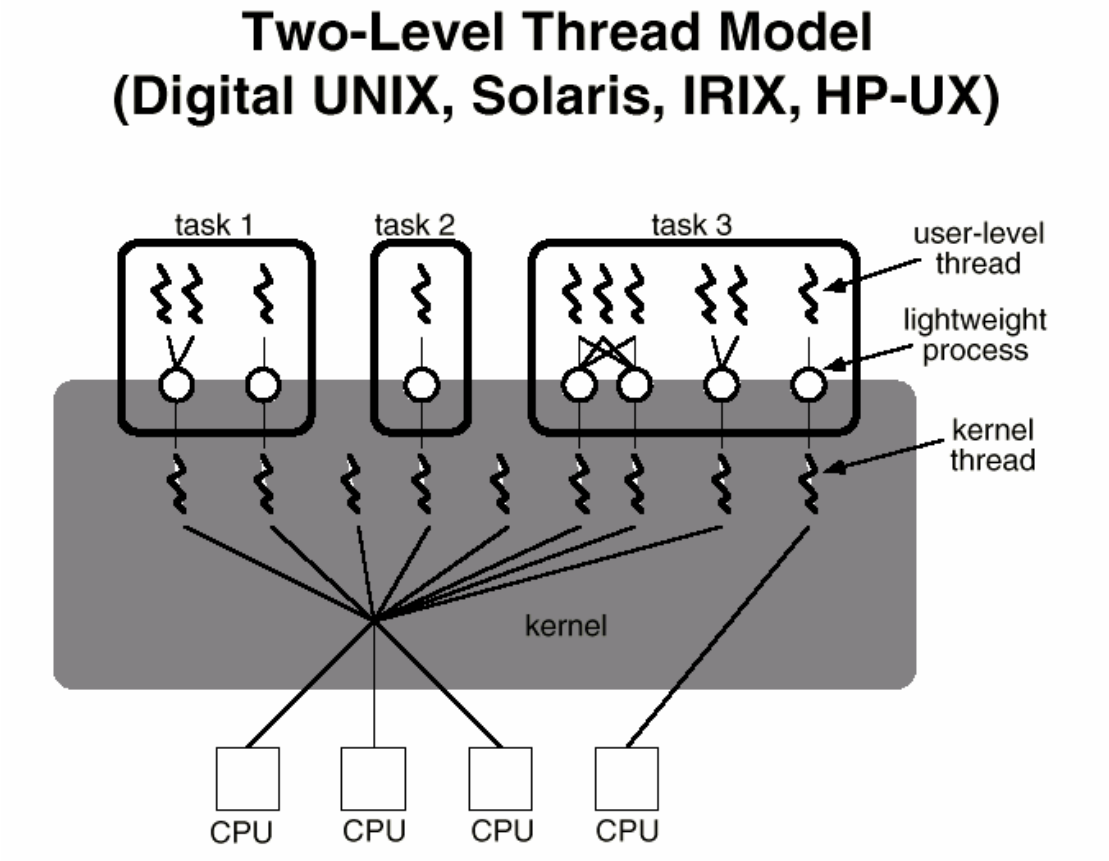


Рис. 2.19. Двохрівнева модель реалізації ниток

Для процесів користувача створюються нитки рівня користувача. “Легковаговий процес” (LWP) служить “Віртуальним CPU”, де можуть виконуватися нитки користувача

Нитки рівня ядра для використання ядром – це одна нитка для кожного LWP а також інші, що виконують задачі, не пов'язані LWPs

2.9. Комунікації між процесами або потоками управління

Взаємодія процесів

Процеси або нитки, що належать одному процесу, можуть взаємодіяти один з одним. Взаємодія процесів може покращувати якість функціонування системи за рахунок перекриття активностей або здійснювання паралельної роботи. Стосовно прикладних програм можна досягти кращої структури програми, будуючи її як множину взаємодіючих процесів, з яких кожен є меншим, ніж одна монолітна програма.

Розглянемо на прикладах, як процеси взаємодіють та як процеси розділяють дані.

Проблема Виробник - Споживач

Один процес є виробником інформації; інший процес є споживачем цієї інформації. Процеси взаємодіють через обмежений (фіксованого розміру) циклічний буфер.

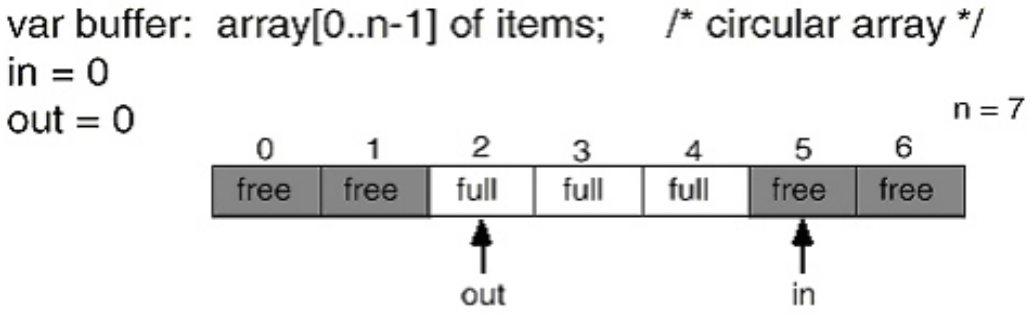


Рис. 2.20. Циклічний буфер.

```
/* producer */                    /* consumer */
repeat forever                    repeat forever
```

```
    …                                       while (in = = out)
    produce item nextp                            do nothing
    …                                       nextc = buffer[out]
    while (in+1 mod n = = out)              out = out+1 mod n
          do nothing                        …
    buffer[in] = nextp                      consume item nextc
    in = in+1 mod n                         …
end repeat                              end repeat
```

Реалізація у системі Java.

Приклад програмування на мові Java.

Producer генерує ціле між 0 і 9 (включно), завантажує його в об'єкт CubbyHole і друкує згенероване число. Producer спить протягом довільного часу між 0 і 100 мілісекундами перед повтором числа:

```
public class Producer extends Thread {
   private CubbyHole cubbyhole;
   private int number;

   public Producer(CubbyHole c, int number) {
      cubbyhole = c;
      this.number = number;
   }

   public void run() {
      for (int i = 0; i < 10; i++) {
         cubbyhole.put(i);
         System.out.println("Producer #" + this.number
                           + " put: " + i);
         try {
            sleep((int)(Math.random() * 100));
         } catch (InterruptedException e) { }
      }
   }
}
```

Consumer, поглинає всі числа з CubbyHole (з точно того ж об'єкта в який Producer поміщає числа) як тільки вони стають доступними.

```
public class Consumer extends Thread {
   private CubbyHole cubbyhole;
   private int number;

   public Consumer(CubbyHole c, int number) {
```

```
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}
```

Ось невелике окреме застосування Java, що створює об'єкт CubbyHole, Producer, Consumer, потім запускає Producer і Consumer.

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Засоби синхронізації у системі Java.

Producer і Consumer у цьому прикладі ділять дані через спільний об'єкт CubbyHole. Важливо, що ні Producer ні Consumer абсолютно не докладають зусиль, щоб перевіряти, що Consumer одержує кожне вироблене число. Синхронізація між цими двома потоками дійсно відбувається на більш низькому рівні, за допомогою методів get і put об'єкта CubbyHole. Проте, давайте звернемо увагу, що ці два потоки не підтримують ніякої синхронізації і поговоримо про потенційні проблеми, що можуть виникнути в цій ситуації.

Одна проблема виникає коли Producer працює швидше, ніж Consumer і генерує два числа перш, ніж Consumer має можливість поглинути перше. Таким чином Consumer буде пропускати число. Частина виводу може виглядати так:

- `. . .`
-
- `Consumer #1 got: 3`
- `Producer #1 put: 4`

```
Producer #1 put: 5
Consumer #1 got: 5
 
    . . .
```

Інша проблема, що могла виникнути, це коли Consumer працює швидше, ніж Producer і поглинає ту ж величину двічі. У цій ситуації, Consumer повинен друкувати ту ж величину двічі і можливий такий вивід:

```
    . . .
 
Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5
 
    . . .
```

У будь-якому випадку, результат неправильний. Треба, щоб Consumer отримав кожне число, вироблене Producer-ом, точно один раз.

Діяльність Виробника і Споживача повинна бути синхронізована в двох напрямках. Спочатку, два потоки не повинні одночасно мати доступ до CubbyHole. У системі Java потік може не дати цьому відбутися, замикаючи об'єкт. Коли об'єкт замкнений одним потоком, а інший потік намагається викликати метод з використанням цього об’єкта, другий потік заблокується поки об'єкт не буде розблокований.

Далі, два потоки повинні діяти узгоджено. Тобто, Producer повинен мати спосіб указувати Consumer-у, що число готове а Consumer повинен мати спосіб указувати, що число взяте.

У системі Java синхронізація підтримується класом Thread. Клас Thread забезпечує набір методів - wait, notify, і notifyAll - щоб допомагати потокам чекати події і повідомляти інший потік коли подія відбувається.

Використання моделі Клієнт/Сервер для передачі повідомлень

Передача повідомлень з використанням передачі і прийому - Send & Receive.

Передача з блокуванням: передача повідомлення іншому процесу, після цього блокування (тобто подача в ОС-у системного виклику призупинення), до того часу, як провідомлення буде прийнято.

Прийом з блокуванням: блокування до того часу, як провідомлення буде прийнято (можливо на хвилини,години, …)

Прямі та непрямі комунікації

Прямі комунікації – задається пряме імя процесу, з яким ви взаємодієте:

Send (destination-process, message)

Receive (source-process, message)

З'єднання асоціюється точно з двома процесами. Між кожними двома процесами може існувати щонайбільше одне з'єднання. З'єднання може бути непрямим, але є звичайно двонаправленим.

Непрямі комунікації – комунікації з використанням поштових скриньок (що належать отримувачу):

Send (mailbox, message)

Receive (mailbox, message)

З'єднання асоціюється з двома або більше процесами, які розділяють поштову скриньку. Між кожними двома процесами у цьому випадку може бути кілька з'єднань. З'єднання може бути непрямим або двонаправленим.

Буферизація

Зв'язок може мати певну ємність, що визначається кількістю повідомлень, що можуть одночасно перебувати у черзі.

Нульова ємність: (черга довжиною 0). Передавач повинен чекати поки приймач прийме повідомлення – ця синхронізація обміну даними називається **рандеву (rendezvous).**

***Обмежена ємність*:** (черга довжиною n). Якщо черга у приймача не заповнена, нове повідомлення додається до черги, і передевач може продовжувати виконання негайно. Якщо черга повна, передавач мусить блокуватись, поки з'явиться місце у черзі.

Необмежена ємність: (необмежена черга). Передавач завжди може продовжувати.

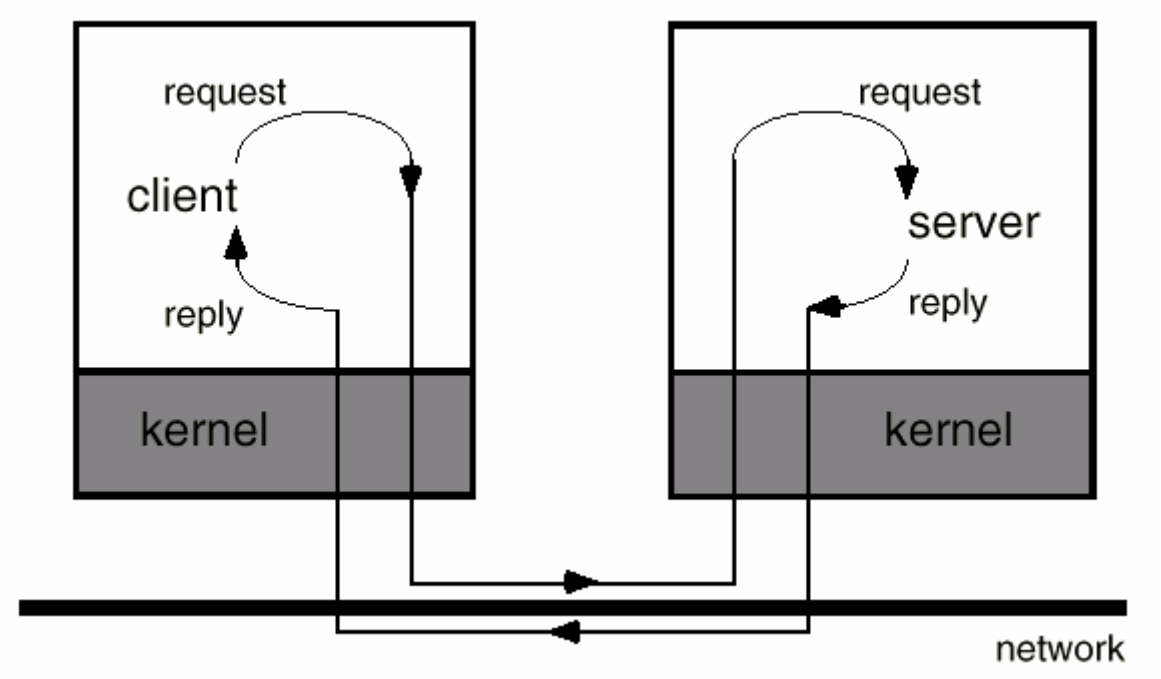


Рис. 2.21. Модель Клієнт/Сервер для передачі повідомлень.

Сервер = процес (або сукупність процесів), який забезпечує сервіс. Приклади: name service, file service. **Клієнт** (Client) – процес, який використовує сервіс.

Протокол запит/відповідь (request / reply):

Клієнт посилає серверу повідомлення – запит (**request**), просячи його виконати деякий сервіс. Сервер виконавши сервіс , посилає відповідь.

Відповідь містить результати виконання або код помилки.

2.10. Віддалений виклик процедур (RPC - Remote Procedure Call)

RPC – парадигма.

Чому передача повідомлень не Ідеал?

Хиби (незручності) зв'язку клієнт/сервер через передачу повідомлень:

- Передача повідомлень є скоріше орієнтованою на ввод/вивід, ніж такою, що орієнтується запит/результат
- Програміст повинен явно кодувати всю синхронізацію
- Програмісту, мабуть, прийдеться кодувати перетворення формату, керування потоком даних, і перевірку промилок

Мета – гетерогенність – підтримка різноманітних машин, різноманітних ОС

- Мобільність - додатки (applications) повинні тривіально переноситись до машин інших виробників

- Спроможність до взаємодії – клієнт повинен завжди отримувати те ж саме обслуговування, незалежно від того, як виробник імплементував те обслуговування

- OS повинен здійснювати перетворення даних між різноманітними типами машин

Модель Клієнт/Сервер з використанням віддаленого виклику процедур.

Механізм Remote Procedure Call (RPC): прихована від програміста передача повідомлень. Видимість (майже) подібна виклику процедури – але клієнт активізує процедуру на сервері. RPC виконується так (див. Рис. 2.22. , високорівневий погляд):

- процес, який здійснив виклик (client), призупиняється;
- параметри процедури передаються через мережу до викликаного процесу (server);
- сервер виконує процедуру, параметри повернення (return) посилає назад через мережу;
- процес, що викликав процедуру, продовжується.

Винайдено: Birrell & Nelson at Xerox PARC, описано у лютому 1984 (ACM Transactions on Computer Systems)

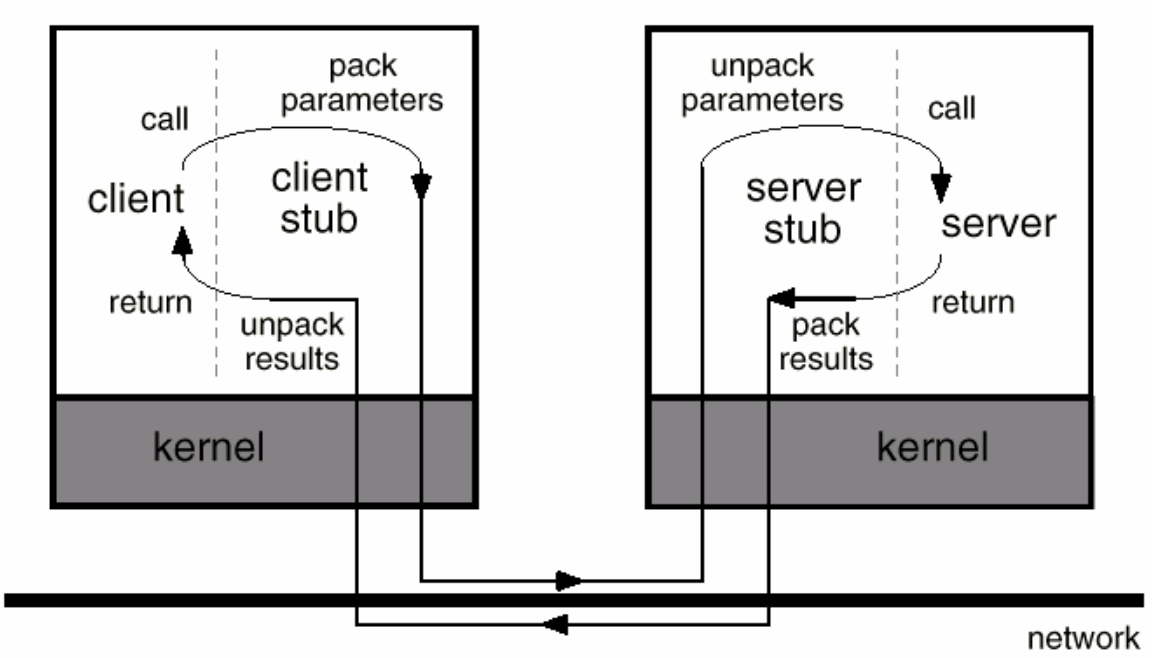


Рис. 2.22. Модель Клієнт/Сервер з використанням віддаленого виклику процедур.

Кожний RPC виклик клієнтського процесу викликає клієнтську заглушку (stub) , яка будує повідомлення і посилає його до серверної заглушки. Серверна заглушка використовує повідомлення для генерації локального виклику процедури на сервері. Якщо локальний виклик процедури повертає значення, сервер будує повідомлення і посилає його заглушці клієнта, яка приймає його і повертає результат(и) клієнту.

RPC Виклик (Більш Деталізований опис)

1. Клієнтська процедура викликає клієнтську заглушку

2. Клієнтська заглушка упаковує параметри в повідомлення й викликає внутрішнє переривання, тобто звертається до ядра

2. Ядро посилає повідомлення віддаленому ядру

4. Віддалене ядро передає повідомлення заглушці сервера

5. Заглушка сервера розпаковує параметри і звертається до сервера

6. Сервер виконує процедуру і повертає результат(и) до заглушки сервера

7. Заглушка сервера упаковує результат(и) виконання процедури у повідомлення і, ініціюючи внутрішнє переривання, входить у режим ядра

8. Віддалене ядро посилає повідомлення локальному ядру

9. Локальне ядро передає повідомлення клієнтській заглушці

10. Клієнтська заглушка розпаковує результат(и) і повертає їх клієнту

Реалізація RPC.

Передача параметрів

Параметр **marshaling** - клієнтська заглушка упаковує параметри в повідомлення, як показано на Рис .

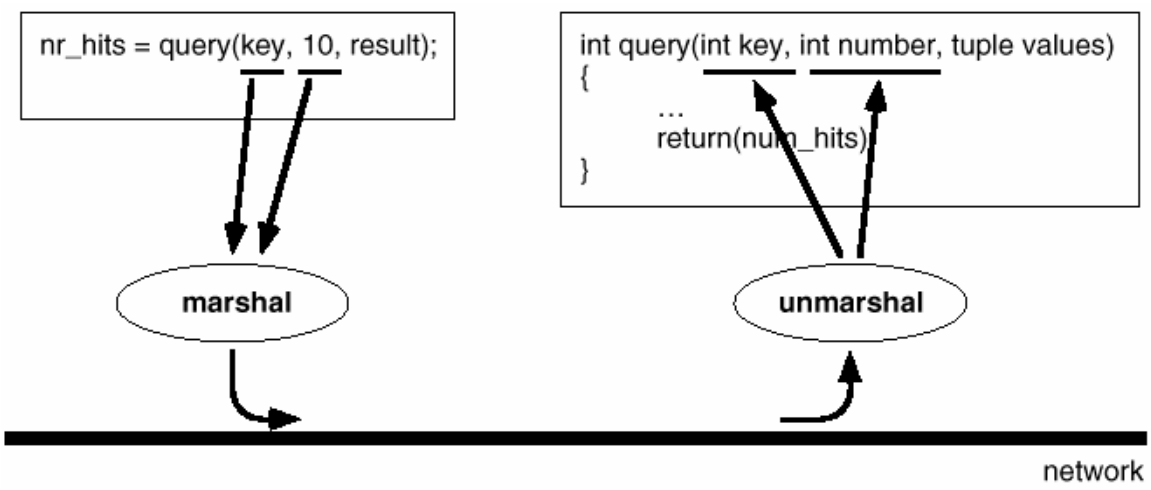


Рис.2.23. Упаковка параметрів.

Параметр **unmarshaling** (рис) - заглушка серверу розпаковує параметри для локальної процедури

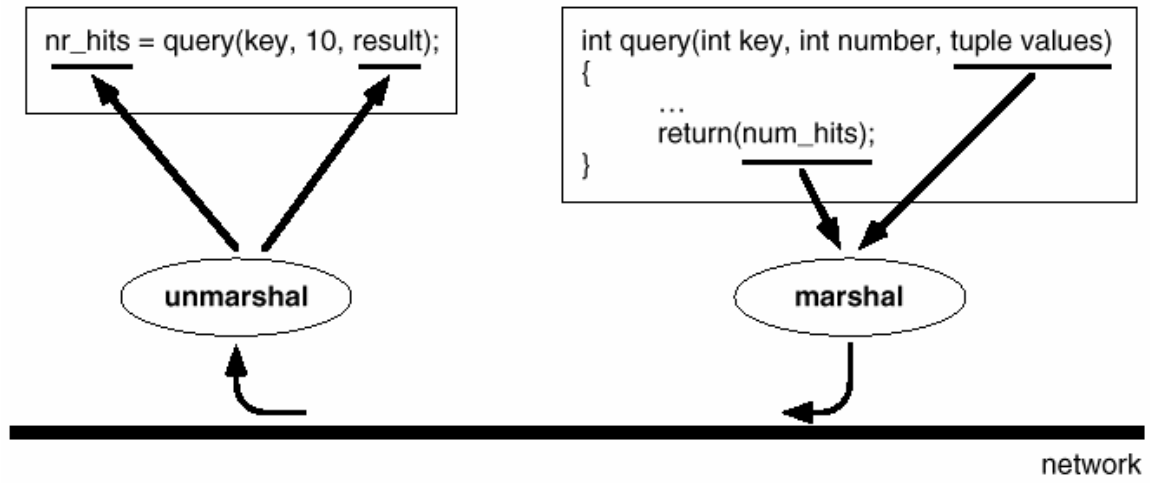


Рис.2.24 Розпаковка параметрів.

Передача параметрів (продовження).

Опрацьовуються різноманітні внутрішні представлення

- ASCII або EBCDIC, або ...
- Фісована кома або плаваюча кома
- ...

Які типи передачі забезпечуються?

- Віддалена процедура не може звертатися до глобальної перемінної – вона повинна сама передати всі необхідні дані

- " Передача параметра за значенням " (процедура отримує копію даних) - параметри передаються в повідомленні

- " Передача параметра по посиланню " (процедура одержує покажчик на дані)
  - Неможлива " передача параметра по посиланню "
    Замість покажчика, передається значення, на яке він вказує

Генерація Заглушок

Автоматична генерація заглушок не може бути описана на C і C++ тому, що не можна явно вказати:

- Які параметри є вхідними, вхідними-вихідними, а які вихідними

- Точно, який розмір параметрів (наприклад, ціле число, масиви)

- Що це означає, якщо передати покажчик

Щоб описати процедуру породження заглушок використовується OSF's DCE Interface Definition Language (IDL)

Зв'язування

Зв'язування = визначення серверу і віддаленої процедури, яку треба викликати.

Статичне зв'язування - адреси серверів є апаратними (наприклад, адреса Ethernet - □□□□□□□□):

- Негнучке, якщо сервер змінює розташування
- Недостатнє, якщо є множинні копії серверів

Динамічне зв'язування – динамічно визначається ім'я сервера

- Клієнти передають Broadcast – повідомлення «, де - сервер? », та чекають відповідь із серверу
- Або клієнти використовують сервер зв'язування (binder). Сервери регіструють/дерегіструють свої послуги на сервері зв'язування.

- Коли клієнт викликає віддалену процедуру перший раз, виконується запит до серверу зв'язування для того, щоб викликати зареєстрований сервер

Підтримка сервером інформації стану

(Приклад = Файловий сервер)

Stateful сервер - підтримує інформацію стану для кожного клієнта та для кожного файла
(open file, read / write file, close file)

- Діє (enable) оптимізація роботи серверу, наприклад при читанні (упередження) і закритті файла
- Важко відновлювати стан після аварійної відмови

Stateless сервер – сервер, що не підтримає інформацію про стан для кожного клієнта

- Кожний запит автономний (файл, позиція, доступ)
  - Без установлення логічного з'єднання (виконання open та close припускається)
- При збоях серверу, клієнти можуть просто періодично передвати запити, поки робота не відновиться
- Ніяких оптимізацій серверу подібно вищезгаданим
- операцій з файлами повинні бути идемпотентними

2.11“RMI - Виклик віддаленого методу” в Java.

Реалізація об’єктів та “Виклик віддаленого методу” в Java.

Що таке RMI? **RMI** (Remote Method Invocation) означає “Виклик віддаленого методу”. Це техніка використовування об’єктів, що знаходяться на інших комп’ютерах.

a) Виклик віддаленого методу.

По-перше, необхідно визначити, що таке “Виклик віддаленого методу” (RMI). За допомогою серіалізації об’єкта можна передавати його за допомогою потоку. RMI – дочірній процес, який дозволяє викликати методи, що розташовані на віддалених системах.
Іншими словами, RMI дозволяє створювати об’єкти Java, методи яких можуть бути викликані віртуальною машиною, що працює на іншому комп’ютері. Ця техніка нагадує віддалений виклик процедури, який часто практикується в інших системах.

b) Створення віддаленого об’єкту.

Щоб методи об’єкта можна було викликати з іншого комп’ютера, об’єкт повинен реалізувати інтерфейс Remote. Такі об’єкти називаються віддаленими.

Віддалений об’єкт реалізується за п’ять кроків:

1. Визначити інтерфейс, породжений від інтерфейсу Remote. В кожному методі цього нового інтерфейсу повинно бути оголошено, що він збуджує RemoteException.
2. Визначити клас, який реалізує цей інтерфейс. Оскільки новий інтерфейс породжений від Remote, це задовольняє вимогу зробити новий клас об’єктом Remote. Клас повинен надати спосіб розташування посилань на екземпляри класу. Зараз UnicastRemoteServer є єдиним класом, який це реалізує.
3. Програмою rmic згенерувати “функції-перехідники”, необхідні для віддалених реалізацій.
4. Створити програму-клієнт, яка буде посилати RMI-виклики на сервер.
5. Запустити програму Registry та виконати програму-сервер та програму клієнт.

Приклад RMI програми.

Перший крок в написанні RMI програми – **створення інтерфейсу, породженого від інтерфейсу Remote**. Кожний з методів цього інтерфейсу можна буде викликати з віддаленого комп’ютера.

Приклад інтерфейсу, породженого від Remote.

```
public interface RemoteInterface extends java.rmi.Remote {
String message (String message) throw java.rmi.RemoteException;
}
```

Створення RMI-серверу

На другому кроці треба визначити клас, який реалізує інтерфейс RemoteInterface.

Приклад серверу, що приймає та посилає String

```
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteServer;
import java.rmi.RemoteExeception;
import java.rmi.server.StubSecurityManager;
public class RemoteServer extends UnicastRemoteServer
implements RemoteInterface{
String name;
public RemoteServer(String name) throws RemoteException{
super();
this.name=name;
```

```
}
public String message(String message) throws RemoteException{
return “My Name is: ”  +name+  ”,thanks for your message:” +message;
}
public static void main(String args[]){
System.setSecurityManager(new StubSecurityManager());
try{
String myName=”Server Test”;
RemoteServer theServer=new RemoteServer(myName);
Naming.rebind(myName,theServer);
} catch (Exception e){
System.out.println(“An Exception occurred while creating server”);
}
}
}
```

Необхідно відмітити ряд ключових моментів відносно класу RemoteServer. По-перше, він породжений від UnicastRemoteServer. В рамках цього розділу можна вважати UnicastRemoteServer аналогом java.applet.Applet для RMI-серверів. По друге, сервер реалізує RemoteInterface з попереднього прикладу.

В кожному методі в RemoteServer, який можна викликати за допомогою RMI, повинно бути оголошено, що він збуджує RemoteExeption. Слід звернути увагу, що навіть метод constructor( ) повинен збуджувати RemoteExeption.

RemoteServer повинен визначати метод message ( ) інтерфейсу RemoteInterface, тому що він реалізував цей інтерфейс. Це важливий метод для даного прикладу, так як саме він буде викликаний через RMI. Щоб не ускладнювати ситуацію, метод message ( ) просто повертає String с повідомленням, яке він отримав. Якщо програма-клієнт отримує рядок, можна бути впевненим, що сервер отримав початкове повідомлення.

Метод main( ) класу RemoteServer посто створює екземпляр серверу, так що можна зв’язатися з ним.

Компіляція RemoteServer

Як і у випадку сереалізації об’єкта, при компіляції RemoteServer необхідно включити додаткові класи. Тому classpath необхідно встановити наступним чином:

```
set classpath=c:\java\lib\classes.zip;c:\objio.zip;
```

На даний момент немає потреби вмикати файл objio.zip, але краще це зробити зараз.

Тепер можна компілювати RemoteServer, вводячи команду:

Javac RemoteServer.java

Наступним кроком в створенні серверу RMI є генерування “функций-переходників” для RemoteServer. Це можна зробити за допомогою компілятора rmic, вводячи команду:

Rmic RemoteServer

Як неважко помітити, її синтаксіс майже такий самий, що у команди java. Компілятор rmic створить два файли:
RemoteServer_Skel.class
RemoteServer_Stub.class

Створення програми-клієнта

Наступним кроком буде створення клієнта, який буде викликати віддалені методи.

Приклад програми-клієнта, яка взаємодіє з класом RemoteServer.

```
import java.rmi.server.StubSecurityManager;
import java.rmi.Naming;
public class RemoteClient{
public static void main(String args[]){
System.setSecurityManager(new StubSecurityManager());
try{
RemoteInterface server=(RemoteInterface) Naming.lookup(“Server Test”);
String serverString=server.message(“Hello There”);
System.out.println(“The serversays :\n”+serverString);
}catch(Exception e){
System.out.println(“Error while perfoming RMI”);
}
}
}
```

Найважливішою частиною класа RemoteClient є два рядки в середині блоку try-catch:
RemoteInterface server = (RemoteInterface) Naming.lookup(“Server Test”);
String serverString = server.message (“Hello There”);

Перша шукає в системному реєстрі “функцію-перехідник” на ім’я “Server Test” (в програмі RemoteServer було використане це ім’я). Створивши екземпляр RemoteInterface, програма викликає метод message з рядком “Hello There”. Фактично, це виклик методу з іншої системи! Він повертає рядок, який зберігається в serverString та в подальшому роздруковується.

Тепер можна відкомпілювати програму-клієнта, аналогічно до того, як була відкомпільована RemoteServer:

Javac RemoteClient.java

При цьому, звичайно, вважається, що змінна classpath для класу RemoteClient вже встановлена.

Запуск Registry та виконання коду

Перед виконанням класів RemoteServer та RemoteClient необхідно запустити RMI-програму Registry на комп'ютері, на якому буде працювати RemoteServer. Але в даному випадку RemoteServer та RemoteClient будуть двома процесами на одному й тому ж комп'ютері. Щоб запустити Registry, треба ввести:

java java.rmi.registry.RegistryImpl

в системі UNIX можна виконувати цю програму в фоновому режимі, вводячи замість попередньої команду:

java java.rmi.registry.RegistryImpl &

В середовищі Windows для запуску сервера необхідно отримати запрощення від MS-DOS. Зробивщи це (або повернувшись до командного рядку UNIX), можна ввести:

java RemoteServer

Як і програму Regisry, на UNIX-машині сервер можна виконувати у фоновому режимі. Для цього треба ввести:

java RemoteServer &

і на кінець, слід відкрити ще одне запрошення MS-DOS та запустити RemoteClient командою:

java RemoteClient

Вивід виглядає так:

The Server Says:
My name is : Server Test, thank for your message:Hello There

2.12. Час у комп'ютерних системах.

Чому нас хвилює “час” у комп’ютерній системі?

Можливо потрібно знати час доби, коли на певному комп’ютері трапилися якась подія.

Потрібно синхронізувати годинник того комп’ютера з деяким зовнішнім надійним джерелом часу (синхронізація з зовнішнім годинником).
Наскільки важко це зробити?

Можливо потрібно знати інтервал часу, або відносний порядок, між двома подіями, що трапилися на різних комп’ютерах

Якщо їх годинники синхронізовані до певного відомого ступеню точності, ми можемо виміряти час, відносно кожного локального годинника (синхронізація інтервалів годинників).
Чи це завжди несуперечливо?

Вимірювання великих відрізків часу.

Проблема відліку і вимірювання часу займала людей завжди, і вже більш 4 тисяч років люди намагаються якось упорядкувати облік часу, створюючи різні календарні системи і пристрої виміру часу.

Для того щоб міряти час розширення всесвіту чи розпаду протона необхідно було створити стандартну схему нумерації днів. За рішенням Міжнародного астрономічного Союзу був прийнятий стандарт на секунду і юліанську систему нумерації днів JDN. Стандартний день містить 86,400 стандартних секунд, а стандартний рік складається з 365,25 стандартних днів.

У схемі JDN, запропонованої в 1583 французьким ученим Джозефом Юлиусом Скалигером, JDN 0.0 відповідає 12 годинам(полудню) першого дня юліанської ери - 1 січня 4713 до нашої ери. Роки до нашої ери підраховуються відповідно до юліанського календаря, у той час як роки нашої ери нумеруються по григорианскому календарю. 1 січня 1 року після різдва христова в григорианскому календарі відповідає 3 січня 1 року юліанського календаря [DER90].

Фізичні стандарти часу.

Еталоном часу може стати будь-як циклічний процес з досить стабільним періодом.

У 1967 році був прийнятий цезієвий (Cs^{133}) стандарт часу, де одна секунда відповідає 9192631770 періодам переходу між рівнями в атомі цезію UTC (Universal Time Coordinated). UTC майже в мільйон разів точніше астрономічного середнього часу за Гринвічем. Помилка UTC складає менш 0,3 нсек за добу .

У 1940-х роках було встановлено, що період обертання землі не постійний - земля сповільнює обертання через припливи й атмосферу. Геологи вважають, що 300 мільйонів років тому в році було 400 днів. Відбуваються і зміни тривалості дня з інших причин. Тому стали обчислювати за тривалий період середню сонячну секунду.

З винаходом у 1948 році атомних годинників з'явилася можливість точно вимірювати час незалежно від коливань сонячного дня. Сьогодні 50 лабораторій у різних точках землі мають годинники, що базуються на частоті випромінювання Цезію-133. Середнє значення показань цих годинників є міжнародним атомним часом (TAI), що обчислюється з 1 липня 1958 року.

Відставання TAI від сонячного часу компенсується додаванням секунди тоді, коли різниця більше 800 мксек. Цей скоректований час, що називається UTC (Universal Coordinated Time), замінив колишній стандарт (Середній час за Гринвічем - астрономічний час). При оголошенні про додавання секунди до UTC електричні компанії змінюють частоту з 60 Hz на 61 Hz (с 50 на 51) на період часу в 60 (50) секунд. Для забезпечення знання точного часу передаються сигнали WWV короткохвильовим передавачем (Fort Collins, Colorado) на початку кожної секунди UTC. Є й інші служби часу, наприклад, інститут ім. Штернберга у Москві.

Мережний протокол часу - NTP.

Призначення протоколу - синхронізації клієнта чи сервера з сервером - джерелом точного часу (радіо або атомний годинник). Стандартизована (Draft Standard) версія 3 протоколу NTP (RFC-1305), але поточна реализація підтримує як 3-тю, так і 4-ту версії. Синхронізується не тільки поточне значення часу, але і частота відліку таймера. Забезпечує точність до мілісекунди в межах LAN і десятків мілісекунд при з'єднанні за допомогою WAN. Передбачено криптографічний захист (шифрування контрольної суми), одночасне підключення до декількох серверів на випадок аварії, алгоритми усереднення і т.д. Підтримує ієрархічну архітектуру мережі, що самонадбудовується. Головні сервери (прямо приєднані до джерела точного часу) утворять перший шар (stratum), сервери, приєднані безпосередньо до них - другий шар, і т.д. Для обміну інформацією використовується протокол UDP (порт 123). Використовуються досить складні алгоритми фільтрації, селекції і комбінування пакетів на принципах максимальної імовірності. Протокол забезпечує підтримку безлічі резервних серверів і шляхів передачі (вибір кращого на основі алгоритму зваженого голосування). Точність первинного сервера, що досягається - мілісекунди. Типовий інтервал опитування - від 1 хвилини (на початку роботи) до 17 хвилин (якщо все добре). Сервер безупинно

коректує хід локальних годинників, використовуючи обчислену інформацію про відхилення їхньої частоти від точної. Це дозволяє зменшити частоту опитування й утримувати незначне відхилення показань годинників від точного часу при тимчасових збоях мережі. Підстроювання частоти забезпечує пристойну точність годинників навіть при модемному з'єднанні з Інтернет. При великих відхиленнях (більш 128 мс) місцевого часу від часу обраного сервера корекція виробляється стрибком, інакше шляхом підстроювання частоти місцевих годинників.

Як значення часу використовується беззнакове 64-бітове число з фіксованою точкою, - число секунд у UTC. Перші 32 біта - число секунд, наступні 32 біта - дробова частина. Точність складає 232 пикосекунди. Старший біт зведений десь у 1968 році, переповнення наступить у 2036 році. 0 означає невизначений час.

Можливі такі класи обслуговування:

- multicast: для використання у швидкій локальній мережі з безліччю клієнтів і без необхідності у високій точності, один чи більш NTP-серверів розсилають broadcast, клієнти визначають час виходячи з припущення, що затримка складає кілька мілісекунд; сервер не приймає відповідних NTP повідомлень

- procedure-call: в умовах коли потрібна висока точність, а multicast недоступний; NTP-клієнт посилає NTP-запит на сервер, що обробляє його і негайно посилає відповідь; сервер не синхронізується з клієнтом

- symmetric: ієрархія серверів, яка динамічно реконфігурується; кожен сервер синхронізується зі своїми сусідами відповідно до правил вибору сусідів; активний режим використовується серверами нижчого рівня із преконфігурованими адресами сусідів, пасивний режим використовується серверами, близькими до кореня; для кожної пари серверів, що обмінюються повідомленнями, створюється асоціація.

Типова конфігурація в невеликій організації включає 3 місцевих сервери, кожний з який підключений до трьох зовнішніх серверів (9 різних зовнішніх серверів!), місцеві сервера з'єднані між собою. Не рекомендується з'єднувати між собою більш 10 серверів. Клієнти підключені до кожного з трьох місцевих серверів.

Список серверів точного часу.

- time-nw.nist.gov
- ntp.ise.canberra.edu.au
- ntp.cs.mu.oz.au
- ntp.mel.nml.csiro.au
- ntp.nml.csiro.au
- ntp.per.nml.csiro.au
- tick.usask.ca
- swisstime.ethz.ch
- ntp0.fau.de
- ntp1.fau.de
- ntp2.fau.de
- ntp3.fau.de
- ntps1-0.cs.tu-berlin.de
- ntps1-0.uni-erlangen.de
- ntps1-1.cs.tu-berlin.de
- ntps1-1.rz.uni-osnabrueck.de
- ntps1-1.uni-erlangen.de
- ntps1-2.uni-erlangen.de
- clock.cuhk.edu.hk
- tempo.cstv.to.cnr.it
- time.ien.it
- clock.nc.fukuoka-u.ac.jp
- clock.tl.fukuoka-u.ac.jp
- cronos.cenam.mx
- ntp0.nl.net
- ntp1.nl.net
- ntp2.nl.net
- ntp.certum.pl
- vega.cbk.poznan.pl
- ntp1.sp.se
- ntp2.sp.se
- time1.stupi.se
- time2.stupi.se
- chronos.csr.net
- clock.isc.org
- clock.via.net
- nist1.aol-ca.truetime.com
- ntp-cup.external.hp.com
- timekeeper.isi.edu
- usno.pa-x.dec.com
- navobs1.usnogps.navy.mil
- navobs2.usnogps.navy.mil
- tick.usno.navy.mil
- tock.usno.navy.mil
- ntp1.connectiv.com

- bonehed.lcs.mit.edu
- navobs1.wustl.edu
- terrapin.csc.ncsu.edu
- lerc-dns.lerc.nasa.gov
- now.okstate.edu
- otc1.psu.edu
- wwv.otc.psu.edu
- nist1.aol-va.truetime.com

Синхронізація фізичних годинників у комп'ютерних системах.

Кожний комп'ютер містить фізичний годинник.

Годинник (або таймер) – це електронний пристрій, що рахує осциляції на кристалі з певною частотою у регістрі – лічильнику

Комп'ютер, усупереч розповсюдженій думці - хронометр дуже неточний, і якщо його періодично не поправляти, те за тиждень він запросто може збитися на кілька хвилин. На щастя, цей "залізний" недолік легко зрівноважується програмним шляхом, і існує кілька способів привести системний час комп'ютера у відповідність зі стандартом.

При розробці програмного забезпечення синхронізації фізичних годинників треба вирішити дві проблеми - годинник не повинні ходити назад (треба чи прискорювати чи сповільнювати їх для проведення корекції) і ненульовий час проходження повідомлення про час (можна багаторазово заміряти час проходження і брати середнє).

Централізовані алгоритми

Використати часовий сервер з пристроєм отримання UTC та синхронізувати все з цим часом

Клієнт встановлює час Tserver+Dtrans

T_{server}= час сервера

D_{trans}= затримка передачі (transmission delay)

Ця затримка непередбачувана через непостійність завантаження мережі.

Алгоритм з осередненням.

Вузли посилають запит до часового сервера, вимірюється час D_{trans} для отримання відповіді T_{server}

Вузли встановлюють локальний час $T_{server}+(D_{trans}/2)$

Точність $\pm((D_{trans}/2)–D_{min})$

Покращення: треба зробити кілька запитів та взяти середнє значення T_{server}

Припущення:

Мережна затримка досить постійна (fairly consistent)

Запит та відповідь займають однаковий час

Проблеми:

Не працює, якщо трапляється поломка на сервері часу

Немає захисту від несправності серверу часу, або зловмисної поломки серверу часу

Алгоритм з координатором.

Обирається комп'ютер-координатор, який буде поводитися як господар (master).

Господар періодично опитує слуг (slaves) – інші комп'ютери, чиї годинники мають бути синхронізовані з господарем.

Слуги посилають значення своїх годинників до господаря. Господар спостерігає затримки передачі та оцінює час їх локальних годинників. Господар усереднює час усіх годинників (включаючи свій власний). Потім бере помилко-стійке середнє (fault-tolerant average) – воно ігнорує читання з годинників, що погано стабілізовані (дрейфують), або ті, що вийшли з ладу та виробляють значення, що виходять за межі діапазонів інших годинників. Господар посилає кожному із слуг певне значення (позитивне або негативне), на яке слуга повинен скорегувати (adjust) свій годинник.

Розподілені алгоритми

Всі вузли мають пристрій для отримання UTC, але бажано, щоб була і внутрішня синхронізація

Глобальне усереднення:

Кожен вузол періодично передає "в ефір" свій час і збирає показники часу, що передаються іншими вузлами, записуючи, коли отримана кожна передача, та різницю між власним годинником та іншими. Потім від бере помилко-стійке середнє різниці та встановлює відповідно локальний годинник.

Проблема: Велике завантаження мережі.

Локалізоване усереднення:

Структурузувати вузли певним чином (кільце, дерево та ін.), щоб кожний вузол усереднював значення невеликої підмножини загальної кількості вузлів.

Відшкодування збитків, заданих відхиленням годинника (time drift)

Порівняти час T_S, наданий часовим сервером, з часом T_C на комп'ютері C

Якщо $T_S > T_C$ (наприклад, 9:07 > 9:05):
Можна збільшити час комп'ютера C до TS
Можна пропустити деякі тіки годинника; можливо все гаразд

Якщо $T_S < T_C$ (наприклад, 9:07 < 9:10)
Не можна зменшити (roll back) час комп'ютера C до TS

Багато додатків (applications) припускають, що час завжди збільшується (advances)!

Можна примусити (cause) годинник комп’ютера С йти повільно поки не відбудеться його ресинхронізація з часом серверу.

Не можна змінити частоту коливань годинника, бо тоді мусить змінитися інтерпретація програмним забезпеченням регістру лічильника годинника

$$T_{software} = a\, T_{hardware} + b$$

Можна визначити константи a та b

2.13. Логічні годинники.

Чи достатньо синхронізувати фізичні годинники?

У розподілених системах, де немає загального годинника, ми мусимо:

використовувати атомний годинник, щоб мінімізувати відхилення годинника;

синхронізувати з серверами часу, що мають пристрої отримання UTC, намагаючись компенсувати непередбачувану затримку мережі.

Чи цього достатньо?

Значення, отримане з пристрою отримання UTC, точне до 0.1-10 мілісекунд. У кращому випадку ми можемо синхронізувати годинники з точністю до 10-30 мілісекунд. Ми маємо синхронізовувати часто, щоб запобігти відхиленню локального годинника. За 10 мс 100 MIPS машина може виконати 1 мільйон інструкцій. Це достатньо точно as time-of-day, але недостатньо точно, щоб визначити відносний порядок подій на різних комп'ютерах у розподіленій системі.

Від фізичних до логічних годинників

Ідея – відмовитися від поняття фізичного часу

Існує багато причин, через які важливо знати порядок, в якому трапляються події.

Лемпорт (Lamport) (1978) – ввів поняття логічного (віртуального) часу, синхронізації логічних годинників

Події та впорядкування подій

Існує багато причин, через які важливо знати порядок, в якому трапляються події.

Подією може бути виконання інструкції, виконання функції та ін. Події включають в себе обмін повідомленнями

Всередині одиночного процесу або між двома процесами на одному комп'ютері порядок, в якому відбуваються дві події, може бути визначений, з використанням фізичного годинника

Між двома різними комп'ютерами у розподіленій системі порядок, в якому відбулися дві події, не можна визначити,

використовуючи локальні фізичні годинники, оскільки ці годинники не можуть бути точно синхронізовані

Відношення “сталося раніше”

Лемпорт (Lamport) визначив відношення “сталося раніше” (позначене “$\rightarrow$”), що описує причинне впорядкування (casual ordering) подій:

якщо a та b – події одного процесу, та a трапилася раніше за b, то $a \rightarrow b$

якщо a – це подія, що посилає повідомлення m, одного процесу, а b – це подія, що отримує повідомлення m, іншого процесу, то $a \rightarrow b$

якщо $a \rightarrow b$ та $b \rightarrow c$, то $a \rightarrow c$ (тобто, відношення “$\rightarrow$” транзитивне)

Причинний зв’язок:

Минулі події впливають на майбутні події

Цей вплив серед причинно впорядкованих подій (тих, що можуть бути впорядковані за відношенням “$\rightarrow$”) відноситься до причинних впливів

Якщо $a \rightarrow b$, подія a причинно впливає на подію b.

Паралельні події:

Дві різні події a та b називаються паралельними (позначення “a || b”), якщо ні $a \rightarrow b$, ні $b \rightarrow a$

Іншими словами, паралельні події не впливають причинно одна на одну

Для будь-яких двох подій a та b в системі вірно: $a \rightarrow b$, або $b \rightarrow a$, або a || b.

Логічний годинник Лемпорта

Щоб впровадити відношення “$\rightarrow$” у розподіленій системі, Лемпорт (1978)[ ] представив концепцію логічних годинників

Кожен процес P_i має логічний годинник C_i

Годинник C_i може приписати значення $C_i(a)$ будь-якій події у процесі P_i

Значення $C_i(a)$ називається часовою поміткою події a у процесі P_i

Значення C(a) називається часовою поміткою події a, незалежно від того, в якому процесі вона трапляється

Часові помітки не мають ніякого відношення до фізичного часу, що веде до терміну логічний годинник

Логічні годинники приписують часові помітки, що монотонно збільшуються, та можуть бути реалізовані за допомогою простих лічильників

Умови (conditions), що задовольняються за допомогою логічних годинників

Параметри синхронізації: якщо $a \rightarrow b$, тоді $C(a)<C(b)$

Якщо подія a трапляється раніше події b, тоді значення годинника (часова помітка) події a має бути менше, ніж значення годинника події b

Однак, не можна сказати: якщо $C(a)<C(b)$, то $a \rightarrow b$

Умови правильності (повинні бути задоволені за допомогою логічних годинників, щоб задовольнити параметрам синхронізації, згаданим вище):

[C1] Для будь-яких двох подій a та b одного й того ж процесу P_i, якщо a трапляється раніше за b, то $C_i(a)<C_i(b)$

[C2] Якщо подія a – це подія, що посилає повідомлення m у процесі P_i, а подія b – це подія, що отримує те саме повідомлення m у іншому процесі P_k, тоді $C_i(a)<C_k(b)$

Впровадження (implementation) логічних годинників

Правила впровадження (Implementation Rules) (гарантують, що логічні годинники задовольняють умовам правильності):

[IR1] Годинник C_i повинен бути збільшений (incremented) між двома будь-якими успішними подіями у процесі P_i:
$C_i := C_i + d$ $(d>0)$ (звичайно $d=1$)

[IR2] Якщо подія a – це подія, що посилає повідомлення m у процесі P_i, тоді повідомленню m приписується часова помітка $t_m = C_i(a)$
Коли те саме повідомлення m отримує інший процес P_k, C_k визначається наступним чином:
$C_k := \max(C_k, t_m + d)$
$(d>0)$ (звичайно $d=1$)

Приклад логічних годинників

Процес коректування часових поміток логічних годинників з використанням методу Лемпорта подано на Рис. 2.25. На цій діаграмі "enn" – подія; "(n)" – значення часової помітки.

Початкове значення часової помітки = 0, d=1. У більшості випадків значення часових поміток збільшуються згідно з правилом IR1. При відправленні повідомлень з вузлів e12, e22, e16, та e24 використовується правило IR1, а при отриманні повідомлень у вузлах e23, e15, та e17 часові помітки приймають значення Ck у відповідності з правилом IR2. При отриманні повідомлення у вузлі e25 часова помітка приймає значення tm + d = 6+1 = 7

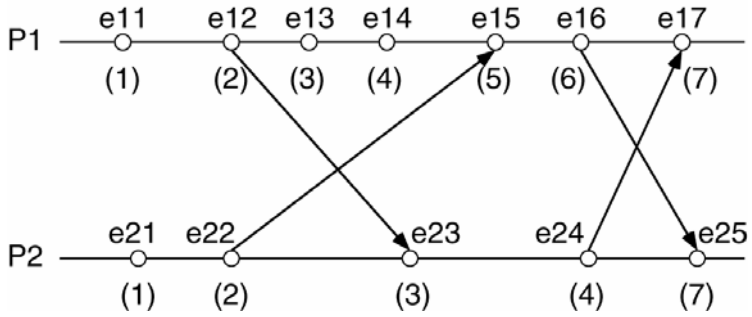


Рис. 2.25. Процес коректування логічних годинників з використанням методу Лемпорта

Обмеження логічних годинників

Згідно з означенням логічних годинників Лемпорта, якщо $a \rightarrow b$, то $C(a)<C(b)$. Але якщо події a та b трапляються у різних процесах і якщо $C(a) < C(b)$, то не обов'язково $a \rightarrow b$

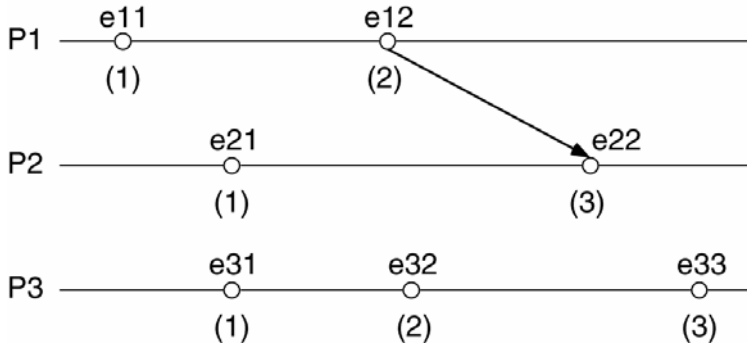


Рис. 2.26. Діаграма, що ілюструє обмеження логічних годинників.

Приклад, поданий на Рис. 2.26, ілюструє це обмеження:

C(e11) < C(e22), та e11→e22 є правильним

C(e11) < C(e32), але e11→e32 є хибним

Не можна визначити з часових поміток чи пов'язані причинно дві події.

2.14. Проблеми синхронізації.

Семафори.

Суть проблеми.

Суть проблеми снхронізації наочно ілюструє поданий нижче приклад, взятий нами з [ ].

Too Much Milk!

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	leave for grocery	
3:15		Arrive home
3:20	Arrive grocery	Look in fridge, no milk
3:25	Buy milk, leave	Leave for grocery
3:30		
3:35	Arrive home	Arrive at grocery
3:36	Put milk in fridge	
3:40		Buy milk, leave
3:45		
3:50		Arrive home
3:51		Put milk in fridge

3:51 *Oh, no! Too much milk!*

Проблема тут у тому, що операції описані строками від "Look in fridge, no milk" до "Put milk in fridge" не є атомарними **(atomic)** операціями.

Термінологія синхронізації.

Синхронізація – використання атомарних (неподільних) операцій щоб гарантувати взаємодію між потоками управління.

Взаємне виключення (Mutual exclusion) – гарантує, що тільки один потік активний у даний час – всі інші потоки виключені з активності.

Критична секція (регіон) – програма, у якій тільки один потік може виконуватись в один час (наприклад, програма, що модифікує спільні дані).

Замок – механізм, що охороняє один потік від одного:

a. замиканням до входу у критичну секцію;

b. відмиканням після виходу з критичної секції – це полягає в тому, що потік, який чекає входу у зачинену критичну секцію, мусить чекати, поки вона відімкнеться.

Реалізація Взаємного виключення (Mutual Exclusion).

Методи реалізації взаємного виключення:

орієнтовані на користувача – потоки мають явно координувати один одного;

орієнтовані на ОС – ОС надає підтримку для взаємного виключення (mutual exclusion);

орієнтовані на технічні засоби - технічні засоби надають архітектурну підтримку для взаємного виключення.

Рішення повинно:

запобігати зависанню – ікщо потік почне тяжко отримувати доступ до критичної секції, все-таки він в кінці кінців досягне мети;

запобігати взаємоблокіровці (deadlock) - якщо деякілька потоків почнуть входити у свої критичні секції, тоді один з них мусить в кінці кінців досягти мети.

Ми будемо припускати що потік може зупинитись у своїй некритичній секції, яле не у своїй критичній секції.

Один з можливих алгоритмів.

Неформальний опис [ ]:

Є іглу з дошкою всередині. Тільки одна персона (потік - thread) може увійти в іглу в один час. В іглу є дошка на якій може бути записано тільки одне значення. Потік, який хоче виконати критичну секцію заходить в іглу, і перевіряє дошку. Якщо його номера немає на дошці, він покидає іглу, виходить назовні, і бігає деякий час навколо іглу. Через деякий час він повертається всередину і перевіряє дошку знову. Це активне чекання (“busy waiting”) продовжується поки в кінці кінців його номер з’явиться на дошці. Якщо його номер є на дошці, він покидає іглу і іде у

критичну секцію. Коли він повертається з критичної секції, він входить в іглу і пише номер іншого потока на дошці.

Семафори - підпримка Взаємного виключення (Mutual Exclusion) операційною системою.

Семафори були введені Дійсктрою [ ] у 1965, і можуть мислитись як загальний механізм замикання. Семафори підтримують дві атомарні операції: **P / wait** та **V / signal.**

Початкове значення семафору = 1. До входу у критичну секцію потік викликає "**P(semaphore)**", або іноді "**wait(semaphore)**". Після виходу з критичної секції потік викликає "**V(semaphore)**", або іноді "**signal(semaphore)**".

Неформальний опис[ ]:

Іглу, містить дошку і дуже великий морозильник.

Wait – потік входить в іглу, перевіряє дошку і декрементує значення, яке він бачить там. Якщо нове значення є 0, потік заходить у критичну секцію. Якщо нове значення від'ємне, потік вповзає у морозильник і зимує (створюючи місце для того,щоб інші входили в іглу).

Signal – потік входить в іглу, перевіряє дошку і інкрементує значення, яке він там бачить. Якщо нове значення є 0 або від'ємне, тоді потік чекає у морозильнику, а також розтоплює один з заморожених потоків, який тоді іде у критичну секцію.

Приклад «Too much milk» з використанням семафорів приймає наступний простий і симетричний вигляд:

```
Thread A                    Thread B

milk –> P( );                   milk –>P( );
if (noMilk)                 if (noMilk)
        buy milk;                   buy milk;
milk –>V( );                milk –>V( );
```

Тут "noMilk" є семафор з початковим значенням 1.

Виконання:

After:	*s*	*queue*	*A*	*B*
			1	
A: noM –> P( );	0		in CS	
B: noM –> P( );	-1	B	in CS	waiting

A: noM –> V( ); 0 finish ready, in CS
B: noM –> V( ); 1 finish

Деталі операцій з семафорами

Семафор “s” ініціалізується значенням 1 до входу у критичну секцію потік викликає “**P(s)**” або “**wait(s)**”

wait (s):
s = s – 1
if (s < 0)
блокує потік, що викликав wait(s) у черзі, що асоціюється з семафором s, або дозволяє потоку, що викликав wait(s), продовження укритичній секції.

Після покидання критичної секції потік викликає “**V(s)**” або “**signal(s)**”

signal (s):
s = s + 1
if (s .LE. 0) then
просинається один з потоків, які викликали wait(s), і виконується, якщо він може, у критичній секції.

Значення семафорів:

Позитивний семафор = число (додаткових) потоків яким може бути дозволений вхід у критичну секцію.
Від’ємний семафор = число блокованих потоків (пам’ятайте – тільки один є у критичній секції).
Двійковий семафор має початкове значення 1. Обчислювальний семафор має початкове значення більше, ніж 1.

Замки, умовні змінні та монітори.

Від семафорів до замків (Locks) та умовних змінних (Condition Variables)

Семафори виконують дві функції: ***взаємне виключення*** – захист спільних даних та ***синхронізацію*** –узгодження подій у часі (один потік чекає чого – небудь, інший потік сигналізує коли це наступає).

***Ідея*:** ввести дві окремі конструкції:
замки (Locks), що забезпечують взаємне виключення;

умовні змінні (Condition variables), що забезпечують синхронізацію.

Як і семафори, замки і умовні змінні є мовно – незалежними, і реалізовані у багатьох програмних середовищах.

Замки (Locks).

Замки (Locks) – забезпечують взаємно виключений доступ до спільних даних. Замок може бути “locked” або “unlocked” (іноді говорять “busy” та “free”).

Операції з замками:

Lock(*name) – create a new (initially unlocked) Lock with the specified name;

Lock::Acquie( ) – wait (block) until the lock is unlocked; then lock it;

Lock::Release( ) – unlock the lock; then wake up (signal) any threads waiting on it in Lock::Acquire( ).

Можуть бути імплементовані:

тривіально як двоїчні семафори (create a private lock semaphore, use P and V);

з кострукцій нижчого рівня, дуже подібних на ті, що використовуються при імплементації семафорів.

Погодження:

до доступу до спільних даних, викликається Lock::Acquire( ) для визначеного замка;

після доступу до спільних даних, викликається Lock::Release( ) для того самого замка.

Приклад використання замків для взаємного виключення (тут “milk” є замком):

```
Thread A                Thread B
Milk –>Acquire( );      milk –>Acquire( );
If (noMilk)             if (noMilk)
      Buy milk;               buy milk;
Milk –>Release( );      milk –>Release( );
```

Замки проти умовних змінних.

Розглянемо таку програму:

```
Queue::Add( )  {
        Lock –>Acquire( );
        Add item
        Lock –>Release( );
}
```

```
Queue::Remove( )  {
            lock –>Acquire( );
        if item on queue
            remove item
        lock –>Release( );
        return item;
}
```

Queue::Remove поверне *item* тільки тоді, коли він уже є у черзі. Коли черга пуста, для *Queue::Remove* треба чекати і не можна іти спати.

Рішення: умовні змінні (condition variable) будуть залишати потік спати всередині критичної секції, для відмикання замка, поки потік спить.

Умовні змінні (Condition Variables).

Умовні змінні узгоджують події. Операції з умовними змінними :

Condition(*name) – create a new instance of class Condition (a condition variable) with the specified name

After creating a new condition, the programmer must call Lock::Lock( ) to create a lock that will be associater with that condition variable

Condition::Wait(conditionLock) – release the lock and wait (sleep); when the thread wakes up, immidiately try to re-acquire the lock; retrn when it has the lock

Condition::Signal(conditionLock) – if threads are waiting on the lock, wake up one of those threads and put it on the ready list; otherwise do nothing

Condition::Broadcast(conditionLock) – if threads are waiting on the lock, wake up all of those threads and put them on the ready list; otherwise do nothing

Важливо: потік мусить зачинити замок до виклику Wait, Signal, or Broadcast.

Може бути імплементовано:

високорівневими конструкціями (create and queue threads, sleep and wake up threads as appropriate);

двійковими семафорами (create and queue semaphores as appropriate, use P and V to synchronize);

конструкціями нижчого рівня, дуже подібно до того, як імплементовані семафори.

Використання замків та умовних змінних.

Замки і умовні змінні є асоційованими з структурою даних. До того, як програма виконує операцію з структурою даних, вона оволодіває замком. Якщо потрібно чекати, поки інша операція приведе структуру даних у відповідний стан, для чекання використовуються умовні змінні.

Приклад: Producer-consumer з необмеженим буфером.

```
Lock *lk;
Condition *c;

/*producer*/
while (1) {
        lk –> Acquire( );
        produce next item
        avail++;
        c –>Signal(lk)
        lk –>Release( );
}
```

```
int avail = 0;

/*consumer*/
while (1) {
        lk –> Acquire( );
        if (avail = = 0)
                c –>Wait(lk);
        consume next item
                avail --;
        lk –>Release( );
}
```

Дві різновидності умовних змінних

1) **Hoare-style** (named after C.A.R. Hoare, used in most textbooks including OSC):

коли потік виконує Signal( ), він віддає замок (і процесор);

чекаючий потік є відібраним як наступний потік, що береться для виконання.

Попередні приклади використовували Hoare-стиль умовних змінних.

2) **Mesa-style** (used in Mesa and most real operating systems):

коли потік виконує Signal( ), він зберігає замок (і процесор);

чекаючий потік стає у чергу готових без спеціального приоритету.

Це не гарантує, що він буде обраним як наступний потік для виконання, інший потік може стати виконуваним.

Коли використовується Mesa-стиль, завжди Wait( ) оточується “while” петлею.

Монітори (Monitors).

Монітор є абстракцією мови програмування, яка автоматично зв'язує замки і умовні змінні з даними. Монітор включає приватні дані (private data) і множину атомарних операцій (member functions).

Одночасно тільки один потік може виконувати програму монітора. Функції монітора мають доступ тільки до даних монітора. Дані монітора недоступні ззовні.

Монітор також має замок, і (optionally) одну або більше умовних змінних. Компілятор автоматично включає операцію acquire на початку кожної функції, і release у кінці.

Спеціальні мови, що підтримують монітори були популярними у 1980-х роках, але недовго. Зараз більшість ОС (OS/2, Windows NT, Solaris) передбачають замки та умовні змінні.

2.15. Розподілене виключне виконання (Distributed Mutual Exclusion)

Взаємне виключення у розподіленому середовищі

У розподіленому середовищі виключне виконання може реалізуватись на основі централізованих або розподілених алгоритмів.

Централізовані алгоритми:

на основі використання центрального фізичного годинника;

на основі центрального координатора.

Розподілені алгоритми з використанням базованого на часі (time-based) впорядкування подій:
розподілений алгоритм Лемпорта;

децентралізований алгоритм на основі часових поміток;

алгоритм широкомовний маркерний.

Розподілені алгоритми з використанням передачі поміток:
алгоритм token-ring (логічне кільце);
алгоритм логічного дерева.

Взаємне виключення у розподіленому середовищі – загальні вимоги

Якщо N процесів поділяють один ресурс та вимагають взаємно виключного доступу, то мають задовольнятися такі умови:

процес, що утримує ресурс, має звільнити його до того, як він (ресурс) може бути наданий іншому процесу;

запити на захоплення ресурсу мають виконуватися у тому порядку, у якому вони були зроблені;

якщо кожен процес, що займає ресурс, в кінцевому випадку звільняє його, тоді кожен запит врешті буде виконано.

Припущення:

- **a.** Повідомлення між двома процесами отримуються у порядку їх посилання
- **b.** Кожне повідомлення в кінцевому випадку отримується
- **c.** Кожен процес може надіслати повідомлення будь-якому іншому

12.1 Центральний фізичний годинник

Впроваджується центральний фізичний годинник так само як у централізованих системах. Процеси отримують фізичні помітки від цього годинника та використовують їх, щоб впорядкувати події.

Перевагою такого алгоритму є простота, але він має такі істотні недоліки:

годинник має бути завжди доступним, щоб забезпечувати надання поміток, що вимагаються;

помилки передачі можуть зашкодити правильному впорядкуванню;

вимагається точна оцінка затримки передачі;

ступінь точності може бути не такою високою, як потрібно.

Центральний координатор

Щоб увійти до критичної секції, процес посилає повідомлення-запит до центрального координатора та чекає на відповідь (див. Рис)

Коли координатор отримує запит:

якщо немає іншого процесу у критичній секції, він посилає назад повідомлення-відповідь;

якщо у критичній секції є інший потік, то координатор додає запит до хвоста своєї черги та не відповідає.

Коли процес, що запитує, отримує відповідь від координатора, він входить до критичної секції. Коли він виходить з критичної секції, то посилає координатору повідомлення-звільнення.

Коли координатор отримує повідомлення-звільнення, він видаляє запит з голови своєї черги та посилає повідомлення-відповідь відповідному процесу.

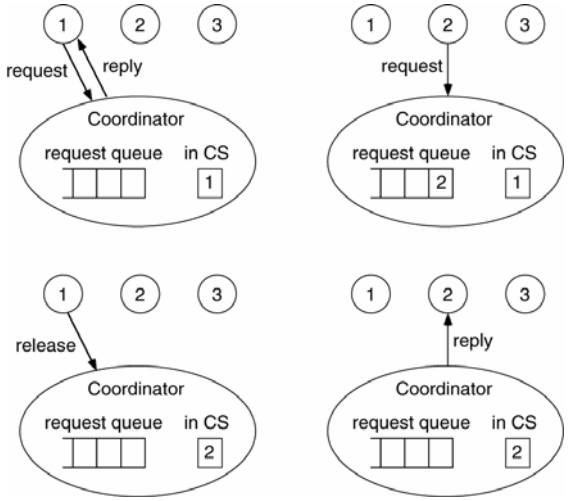


Рис. 2.27. Приклад реалізації взаємного виключення у розподіленому середовищі; алгоритм з центральним координатором; CS (Critical Section) – критична секція

Розподілений алгоритм Лемпорта **[(1978)].**

Кожен процес підтримує свою власну чергу запитів, впорядкованих за значеннями часових поміток.

Запит до критичної секції (КС): Коли процес хоче увійти до критичної секції, він додає запит до своєї власної черги та посилає повідомлення-запит, помічене часовою поміткою, усім процесам у множині запитів цієї КС. Коли процес отримує повідомлення-запит, він додає запит до своєї власної черги запитів та повертає повідомлення-відповідь, помічене часовою поміткою.

Виконання у критичній секції: Процес входить до КС коли його власний запит знаходиться на вершині його власної черги (його запит найбільш ранній) і, він отримав повідомлення-відповідь з часовою поміткою більшою, ніж його запит, від усіх інших процесів у множині запитів.

Звільнення критичної секції:

Коли процес звільняє критичну секцію, він видаляє свій власний (задоволений) запит з вершини своєї власної черги запитів та посилає повідомлення-звільнення, помічене часовою поміткою усім процесам у множині запитів.

Коли процес отримує повідомлення-звільнення, він видаляє (задоволений) запит з своєї власної черги запитів. Можливо зміщення його власного повідомлення до вершини черги, що дозволяє йому нарешті увійти до критичної секції.

На Рис. 2.28. показано конкретний приклад дії даного алгоритму. Обидва процеси 0 та 2 посилають запити до КС. Кожен відповідає, процес 0 входить до КС, бо його запит був першим.

Процес 0 звільняє КС, процес 2 входить до неї.

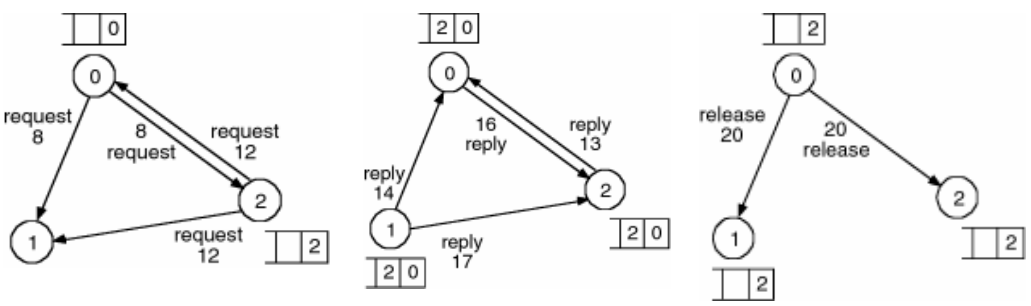


Рис. 2.28. Приклад реалізації взаємного виключення у розподіленому середовищі; розподілений алгоритм Лемпорта.

Децентралізований алгоритм на основі часових поміток.

Алгоритм запропоновано в [Ricart&Agrawal] і є поліпшенням алгоритму, що запропонував Лемпорт.

Потрібно глобальне упорядкування всіх подій у системі за часом.

Вхід у критичну секцію: Коли процес бажає ввійти в критичну секцію, він посилає всім процесам повідомлення-запит, що містить ім'я критичної секції, номер процесу і поточне час. Після посилки запиту процес чекає, поки усі дадуть йому дозвіл. Після одержання від усіх дозволу, він входить у критичну секцію.

Поводження процесу при прийомі запиту: Коли процес одержує повідомлення-запит, у залежності від свого стану стосовно зазначеної критичної секції він діє одним з наступних способів. Якщо одержувач не знаходиться усередині критичної секції і не запитував дозвіл на вхід у неї, то він посилає відправнику повідомлення "OK". Якщо одержувач знаходиться усередині критичної секції, то він не відповідає, а запам'ятовує запит. Якщо одержувач видав запит на входження в цю секцію, але ще не ввійшов у неї, то він порівнює тимчасові мітки свого запиту і чужого. Перемагає той, чия мітка менше. Якщо чужий запит переміг, то процес посилає повідомлення "OK". Якщо в чужого запиту мітка більше, те відповідь не посилається, а чужий запит запам'ятовується.

Вихід із критичної секції: Після виходу із секції він посилає повідомлення "OK" усім процесам, запити від який він запам'ятав, а потім стирає всі запам’ятовані запити.

Кількість повідомлень на одне проходження секції - 2(n-1), де n - число процесів. Крім того, одна критична точка замінилася на n точок (якщо якийсь процес перестане функціонувати, то відсутність дозволу від нього усіх зупинить). І, нарешті, якщо в централізованому алгоритмі є небезпека перевантаження координатора, то в цьому алгоритмі перевантаження будь-якого процесу приведе до тих же наслідків. Деякі поліпшення алгоритму (наприклад, чекати дозволу не від усіх, а від більшості) вимагають наявності неподільних широкомовних розсилань повідомлень.

Розширення для К спільних ресурсів.

Процес може увійти до КС як тільки він отримає N-K повідомлень-відповідей. Алгоритм, загалом, такий самий як [P&A (алгоритм Рікарта та Агравала)] з однією різницею: N-K повідомлень-відповідей надходять, коли процес чекає, щоб увійти до КС; нші К-1 повідомлень-відповідей можуть надходити, коли процес знаходиться у КС, після того, як він залишає КС або, коли він чекає, щоб увійти до КС знову. Необхідно зберігати у пам'яті системи кількість очікуючих процесів.

Алгоритм Token-Ring

Процеси розташовані у логічне кільце. На початку одному з процесів надається маркер. Маркер циркулює по колу у визначеному напрямку за допомогою "двоточкових" (point-to-point) повідомлень. Коли процес має маркер, він має право увійти до КС. Після виходу з КС, він передає маркер далі.

Алгоритм широкомовний маркерний

Маркер містить чергу запитів та масив **LN[1...N]** з номерами останніх удоволених запитів.

Вхід у критичну секцію. Якщо процес **Pk**, що запитує критичну секцію, не має маркера, то він збільшує порядковий номер своїх запитів **RNk[k]** і посилає широкомовно повідомлення "ЗАПИТ", що містить номер процесу (**k**) і номер запиту (**Sn=RNk[k]**). Процес **Pk** виконує критичну секцію, якщо має (чи коли одержить) маркер.

Поводження процесу при прийомі запиту. Коли процес **Pj** одержить повідомлення-запит від процесу **Pk**, він установлює **RNj[k]=max(RNj[k],Sn)**. Якщо **Pj** має вільний маркер, то він його посилає **Pk** тільки в тому випадку, коли **RNj[k]=LN[k]+1** (запит не старий).

Вихід із критичної секції процесу **Pk**. Процес установлює **LN[k]** у маркері рівним **RNk[k]**. Для кожного **Pj**, для якого **RNk[j]=LN[j]+1**, він додає його ідентифікатор у маркерну чергу

запитів. Якщо маркерна черга запитів не порожня, то з її віддаляється перший елемент, а маркер посилається відповідному процесу (запит якого був першим у черзі).

Алгоритм деревоподібний маркерний

Усі процеси представлені у виді збалансованого двоїчного дерева. Кожен процес має чергу запитів від себе і сусідніх процесів (1-го, 2-х чи 3-х) і покажчик у напрямку власника маркера.

Вхід у критичну секцію. Якщо є маркер, то процес виконує КС. Якщо немає маркера, то процес оміщає свій запит у чергу запитів та посилає повідомлення "ЗАПИТ" у напрямку власника маркера і чекає повідомлень.

Поводження процесу при прийомі повідомлень Процес, що не знаходиться усередині КС повинний реагувати на повідомлення двох видів -"МАРКЕР" і "ЗАПИТ".

Якщо прийшло повідомлення "МАРКЕР" то:

узяти 1-ий запит з черги і послати маркер його автору (концептуально можливо собі);

поміняти значення покажчика убік маркера;

виключити запит з черги;

якщо в черзі залишилися запити, то послати повідомлення "ЗАПИТ" убік маркера.

Якщо Прийшло повідомлення "ЗАПИТ", помістити запит у чергу. Якщо немає маркера, то послати повідомлення "ЗАПИТ" убік маркера, інакше (якщо є маркер) - перейти до обробки повідомлення "МАРКЕР" .

Вихід із критичної секції. Якщо черга запитів порожня, то при виході нічого не робиться, інакше - перейти до обробки повідомлення "МАРКЕР" .

Алгоритми вибору (Election Algorithms).

У розподілених системах багато алгоритмів вимагають наявності постійного або тимчасового лідера:

Розподілене взаємне виключення:

Алгоритм центрального координатора вимагає наявності координатора

Алгоритм token-ring, алгоритм Сузукі-Касамі та алгоритм дерева Реймонда вимагає наявності елементу, що на початку буде мати маркер (token).

Розподілене виявлення тупиків (deadlock) – вимагає наявності утримувача (maintainer) глобального графа очікування (wait-for graph).

Якщо лідер виходить з ладу, повинен бути обраний новий лідер

Алгоритми вибору припускають, що кожен потік має унікальний пріоритетний номер

Мета: обрати лідером потік з найвищим пріоритетом, повідомити усім активним потокам

Друга мета: дозволити поновленому лідеру відновити контроль (або, хоча б, розпізнати поточного лідера)

Алгоритм Гарсіа-Моліна (Garcia-Molina's)

У цьому алгоритмі є три типи повідомлень:

вибір – анонсує про вибір;

відповідь – підтверджує прийом повідомлення про вибір;

координатор – анонсує про нового координатора.

Вибір:

Процес починає вибір якщо до нього надходить інформація, що координатор вийшов з ладу (див. Рис). Щоб це зробити, він посилає *вибір*-повідомлення усім потокам з вищими пріоритетами. Потім він чекає на відповідь (від діючого потоку з вищим

пріоритетом). Якщо повідомлень деякий час немає, він проголошує себе координатором та посилає повідомлення-*координатор* усім потокам з нижчими пріоритетами.

Якщо відповідь надходить, він чекає деякий час на повідомлення-*координатор* від нового координатора. Якщо немає відповіді, він знову починає вибір.

Результат вибору:

Якщо процес отримує повідомлення-координатор, він приймає нового координатора

Участь у виборі:

Якщо процес отримує повідомлення - *вибір*, Він посилає назад повідомлення – *відповідь.*

Пошкоджені процеси.

Коли пошкоджений процес відновлюється, він починає вибір, якщо тільки він не знає, що має найвищий пріоритет. Якщо відновлений процес має найвищий пріоритет, він лише розсилає повідомлення *координатор*, щоб відновити контроль.

Розрахунки: N-2 повідомлень у найкращому випадку $O(N^2)$ повідомлень у найгіршому випадку

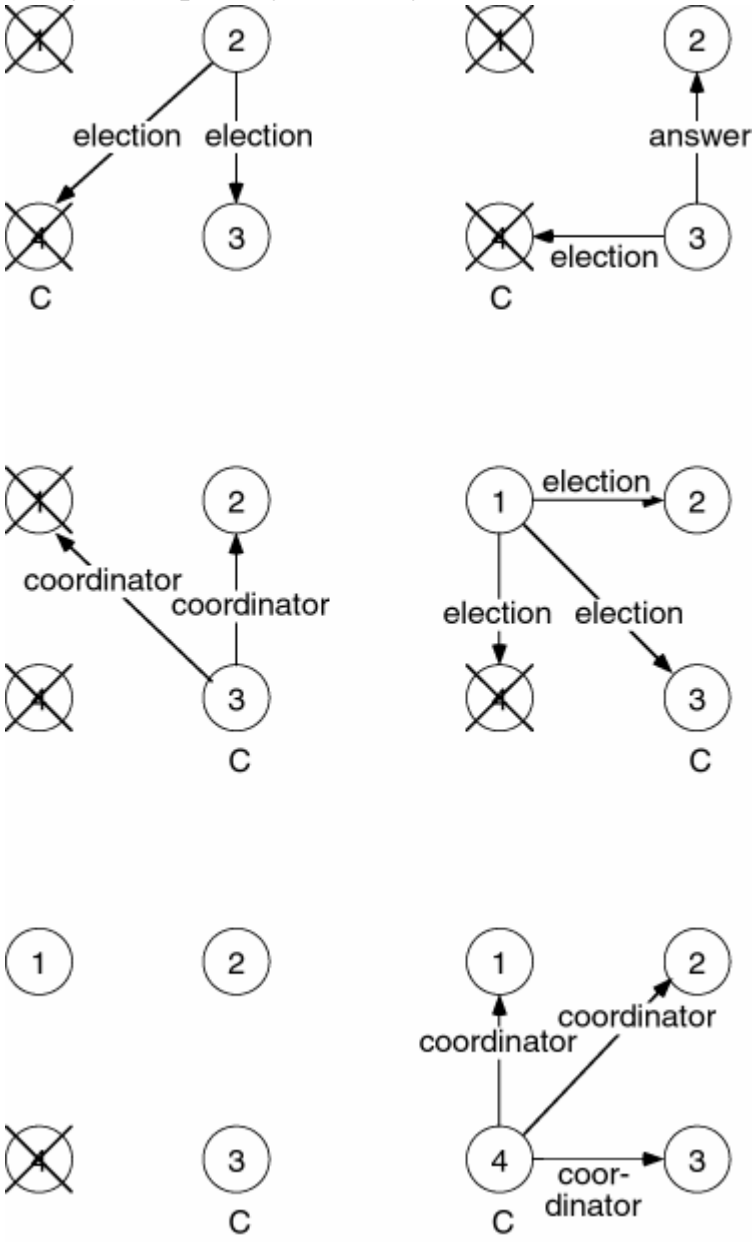


Рис. 2.29. Алгоритм вибору координатора

Алгоритм кільця Чанга та Робертса (Chang, Roberts)

Процеси розташовані у логічне кільце (див. Рис).

Кожен процес спочатку не-учасник (non-participant)

Процес починає вибір тим, що помічає себе як *учасника* (participant). Посилає повідомлення - *вибір*, що містить його ідентифікатор, своїм сусідам.

Коли процес отримає повідомлення - *вибір*, він порівнює ідентифікатор, що надійшов у повідомленні зі своїм власним. Якщо ідентифікатор, що надійшов, більше, тоді процес, якщо він не *учасник*, помічає себе як *учасника* та передає далі повідомлення своєму сусідові. Якщо ідентифікатор, що надійшов, менше і якщо процес не *учасник*, він помічає себе як *учасника,* прописує свій власний ідентифікатор у повідомлення та відсилає його далі. Якщо він вже *учасник*, він нічого не робить.

Коли процес отримує повідомлення про вибір, він аналізує отриманий ідентифікатор. Якщо ідентифікатор, що надійшов, є ідентифікатором самого потоку, тоді його ідентифікатор найбільший, отож він стає координатором. Він помічає себе як *не-учасника* знову, та посилає *виборне* повідомлення до свого сусіда, проголошуючи про результати вибору та свій ідентифікатор

Коли процес отримує *виборне* повідомлення, він помічає себе як *не-учасника* та передає повідомлення далі до свого сусіда.

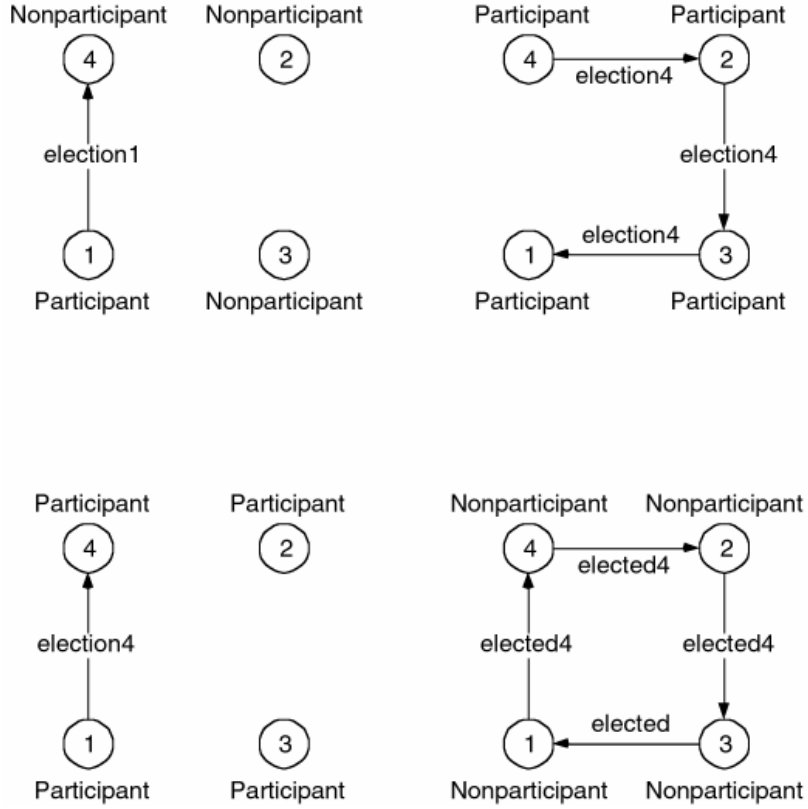


Рис.2.30. Вибір координатора на основі алгоритму «логічне кільце».

Розрахунки:

3N-1 повідомлень у найгіршому випадку

N-1 повідомлень *вибір*, щоб досягти найближчого сусіда у неправильному напрямку, N повідомлень *вибір*, щоб вибрати його, потім N *виборних* повідомлень, щоб проголосити про результат.

2.16. Тупикові ситуації (Deadlocks).

Тупикова ситуація (Deadlock).

Тупикова ситуація виникає, коли кожний з двох або більше процесів чекають на подію яка не може наступити, тому що вона може бути генерована тільки іншим процесом з цієї сукупності.

Тупикова ситуація є однією з найбільш складних проблем, що зустрічаються проектувальникам ОС.

Умови виникнення тупикової ситуації.

Наступні 4 умови є необхідними і достатніми для того, щоб наступила тупикова ситуація:

Mutual exclusion (виключне виконання) – якщо процес захопив ресурс, то інші процеси, що запитують цей ресурс, мусять чекати, поки процес звільнить його.

Hold and wait (захоплення і чекання) – процеси можуть володіти одним (або більше) ресурсом і перебувати у чеканні захоплення (acquire) додаткових ресурсів що знаходяться у володінні інших процесів.

No preemption (непереривність) – ресурси вивільняються добровільно; ні інший процес ні ОС не можуть принудити процес до звільнення ресурсів.

Circular wait (циклічне чекання) – мусить існувати множина чекаючих процесів, таких, що P0 чекає на ресурс, захоплений P1, P1 чекає на ресурс, захоплений P2, ...Pn-1 чекає на ресурс, захоплений Pn, і Pn чекає на ресурс, захоплений P0.

Граф розподілу ресурсів (Resource-Allocation Graph).

Умови критичної ситуації можуть бути модельовані з використанням направлених графів, що називаються **Resource-Allocation Graph.**

Є 2 різновидності вузлів:
Прямокутники (Boxes) – представлають ресурси
Кола (Circles) – представляють потоки/процеси

Є 2 різновидності (направлених) ребер:

Ребро запиту (Request edge) – від процесу до ресурсу – показує, що процес запитав ресурс і чекає, щоб захопити його
Ребро призначення (Assignment edge) – від ресурсу до процесу – показує, що процес володіє ресурсом.

Коли робиться запит, у граф додається ребро запиту. Коли запит виконано, ребро запиту трансформується у ребро призначення. Коли процес вивільняє ресурс, ребро призначення видаляється.

Інтерпретація RAG з одним екземпяром ресурсу.

Рис. 2.31. Якщо граф не містить циклу, тупикової ситуації немає.

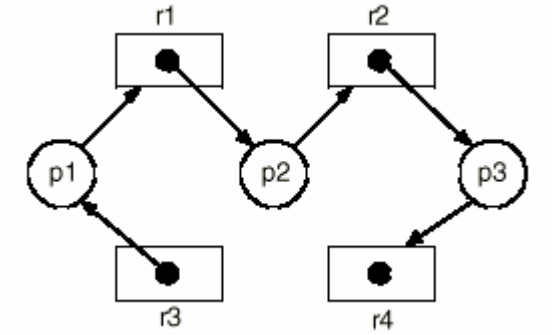
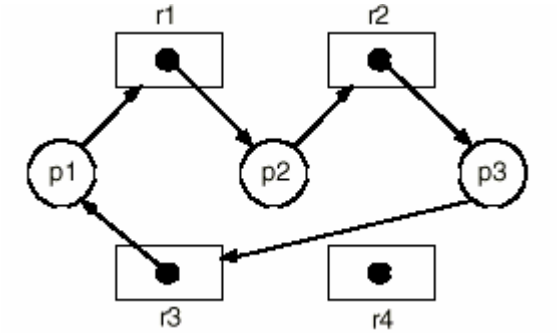


Рис. 2.32. Якщо граф містить цикл, існує тупикова ситуація.



У випадку одного екземпляру кожного ресурсу
цикл є необхідною і достатньою умовою тупикової ситуації.

Поведінка у тупиковій ситуації.

1) Страусова поведінка – заховати свою голову під крило і ігнорувати проблему.

2) Попередження тупикових ситуацій - через виключення попадання в одну з 4-х умов виникнення тупикової ситуації.

3) Алгоритми визначення тупикової ситуації – визначають коли має наступити тупикова ситуація.

4) Алгоритмі відновлення – розривають тупикові ситуації.

5) Алгоритми запобігання - розглядають доступні у даний час ресурси, розміщені для кожного потоку, і можливо запитувані у майбутньому, і виконують тільки запити, що не приведуть до тупикової ситуації.

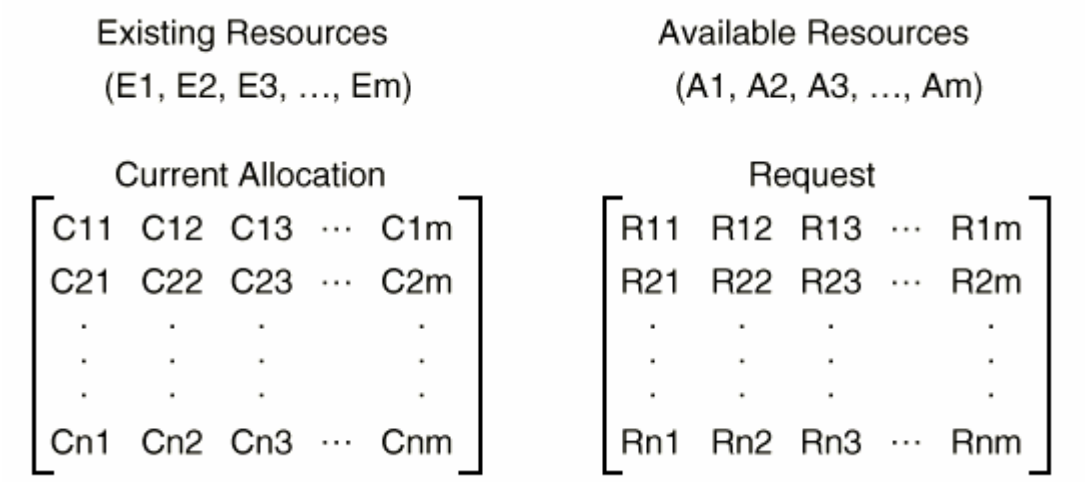
Виявлення тупикових ситуацій (один ресурс кожного типу).

Якщо всі ресурси мають тільки по одному екземпляру, тупикова ситуація може бути виявлена пошуком циклів у графі розміщення ресурсів.

Один простий алгоритм: Стартувати у кожному вузлі, і іти шукати в глибину від нього. Якщо пошук повертається до вузла що уже зустрічався, то знайдено цикл.

Виявлення тупикових ситуацій (багато ресурсів кожного типу).

Цей алгоритм (Coffman, 1971) використовує такі структури даних:



Розглядається n процесів, m типів ресурсів.

Existing Resources – вектор показує кількість існуючих ресурсів кожного типу.

Available Resources – вектор показує кількість доступних ресурсів кожного типу (не призначених ніякому процесу).

i-ий рядок **Current Allocation матриці показує** кількість ресурсів кожного типу, призначених процесу **i**.

Кожний ресурс є або призначеним або готовим (available).

Кількість ресурсів типу **j**, які були виділені всім процесам, плюс
кількість ресурсів типу **j**, які готові, дорівнює кількості існуючих ресурсів типу **j**.

Процеси можуть мати невиконані запити. **i**-ий рядок **Request -** матриці показує кількість ресурсів процесу **i** кожного типу, що запитані, але ще не отримані.

Алгоритм визначення тупикових ситуацій (багато ресурсів кожного типу).

Операція:

Кожний процес спочатку є немаркованим.

Коли алгоритм просувається, будуть маркуватись процеси, які здатні завершитись без тупикової ситуації.

Коли алгоритм завершиться, будь-які немарковані процеси створюють тупикові ситуації.

Алгоритм:

1. Дивіться на немаркований процес Pi коли i-ий рядок **Request –** матриці менше або дорівнює **Available –** вектору

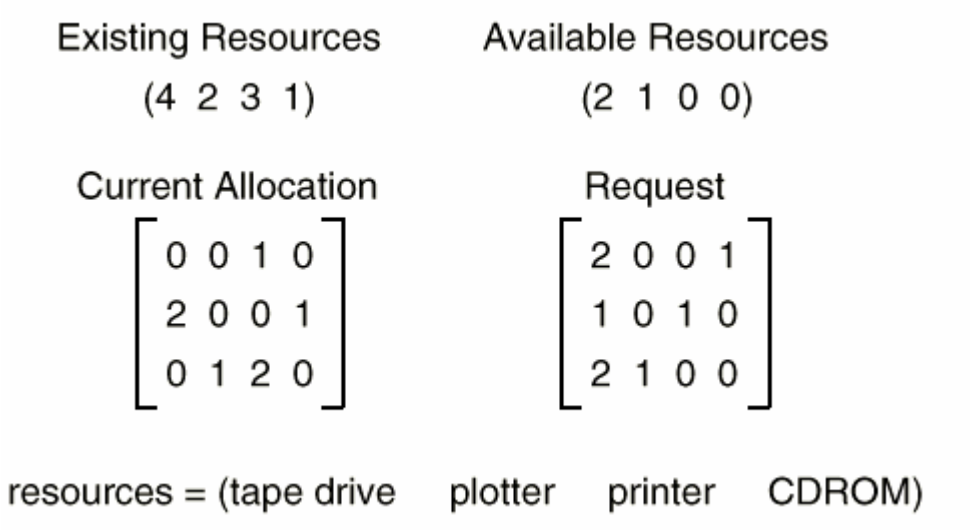
Notation: comparing vectors

If A and B vectors, the relation $A \leq B$ means that each element of A is less than or equal to the corresponding element of B (i.e., $A \leq B$ iff $A_i \leq B_i$ for $0 \leq i \leq m$)

Furthermore, $A < B$ iff $A \leq B$ and $A \neq B$

2. Якщо такий процес знайдено, додайте i-ту строку **Current –** матриці до **Available –** вектора, маркуйте процес і перейдіть до кроку 1.
3. Якщо такого процесу не існує, алгоритм завершено.

Приклад визначення тупикових ситуацій
(багато ресурсів кожного типу).



Whose request can be fulfilled?

- Process 1 — no — no CDROM available
- Process 2 — no — no printer available
- Process 3 — yes — give it the requested resources, and after it completes and releases those resources, A = (2 2 2 0)
- Process 1 still can't run (no CDROM), but process 2 can run, giving A = (4 2 2 1)
- Process 1 can run, giving A = (4 2 3 1)

Приклад тупикової ситуації та її уникнення.

Взаємоблокування

Спеціальний тип помилки, яку треба уникати і яка відноситься до багатозадачності, це виникнення тупикової ситуації - взаємне *блокування.* Взаємоблокування — важка помилка для налагодження по двох причинах:

взагалі говорячи, воно відбувається дуже рідко, коли інтервали часового квантування двох потоків знаходяться у певному співвідношенні;

воно може включати більше двох потоків і синхронізованих об'єктів, тобто блокування може відбуватися через заплутану послідовність подій.

Для повного розуміння блокування корисно побачити його в дії. Наступний приклад створює два класи (A і B) з методами foo() і bar(), відповідно, які роблять коротку паузу перед спробою викликати метод іншого класу. Головний клас (з ім'ям Deadlock) створює екземпляри типу A й B, і потім стартує другий потік для установки стану блокування. Обидва методи використовують sleep() для виклику стану блокування.

```
// приклад блокування,

class A {

synchronized void foo(B b) {

String name = Thread.currentThread().getName();

System.out.println(name + " entered A.foo");

try {

Thread.sleep(1000);

} catch(Exception e)

{

System.out.println("A interrupted");

}


System.out.println(name + "tries to call B.last()");

b.last();

}

synchronized void last() {

System.out.println("Inside A.last");

}
```

```
}

class B {
synchronized void bar(A a) {
String name=Thread.currentThread().getName();
System.out.println(name + " entered B.bar");
try {
Thread.sleep(1000);
} catch(Exception e)
{
System.out.println("B interrupted");
}
System.out.println (name + " tries to call A.last()");
a.last();
}
synchronized void last(){
System.out.println("inside A.last");
}
}
class Deadlock implements Runnable  {
A a = new A();
B b = new B();
Deadlock()  {
Thread.currentThread().setName("MainThread");
Thread t = new Thread(this, "RacingThread");
t.start();
a.foo(b);                    // отримати блокування на цьому потоці
```

```
System.out.println("return to main thread");
}
public void run()  {
b.bar(a);              // отримати блокування на b в іншому потоці
System.out.println("Return to other thread");
}
public static void main(String args[])
{
new Deadlock();
}

}
```

При виконанні цієї програми виведуться наступні повідомлення:

MainThread entered A.foo
RacingThread entered B.bar
MainThread tries to call B.last()
RacingThread tries to call A.last()

Оскільки програму було блоковано, для виходу з неї треба натиснути **<Ctrl>+<C>**. Можна побачити повний кеш-дамп потоку й монітору, якщо натиснути **<Ctrl>+<Break>** на PC (або **<Ctrl>+<\>** на Solaris).
Видно, що RacingThread має монітор на **b**, у той час як він очікує монітор на **a**. У той же час MainThread має **a** та чекає на отримання **b**. Ця програма ніколи не буде завершена.

Зупинка, відновлення та закриття потоку в Java 2

Нижче наведена програма, що дозволяє правильно управляти потоками в Java 2, уникаючи стану блокування.
Класс NewThread містить boolean-змінну экземпляра з іменем suspendFlag, що використовується для управління виконанням потоку. Вона ініціалізується значенням false. Метод run() містить синхронізований блочний оператор, що перевіряє suspendFlag. Якщо ця змінна — true, то викликається метод wait(), щоб зупинити виконання потока. Метод mysuspend() встановлює у suspendFlag значення true. Метод myresume () встановлює в suspendFlag значення false та викликає метод notify(),

щоб відновити потік. Нарешті, метод main() було модифіковано для виклику методів mysuspend() та myresume ().

```
// Зупиняє та відновлює потік для Java 2

class NewThread implements Runnable {

String name;                    // ім’я потока

Thread t;

boolean suspendFlag;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

suspendFlag = false;

t.start();                      // старт потока

};

// точка входу для потока

public void run()
```

```
{
try {
for(int i=15; i>0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(200);
        }
synchronized (this){
        while(suspendFlag) {
        wait();}
  System.out.println(name + " ended.");

}catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
}
}
```

```
synchronized void mysuspend() {
suspendFlag = true;
}
synchronized void myresume()
{
  suspendFlag = false;
  notify();
};
}

  class SuspendResume {
public static void main(String args[]) {
NewThread ob1 = new NewThread("First");
NewThread ob2 = new NewThread("Second");
  try {
  Thread.sleep(1000);
```

```
ob1.mysuspend();

System.out.println("First thread sleeps");

Thread.sleep (1000);

ob1.myresume();

System.out.println ("First thread wakes up");

ob2.mysuspend();

System.out.println("Second thread sleeps");

Thread.sleep(1000);

ob2.myresume();

  System.out.println("Second thread wakes up");

  } catch (InterruptedException e) {

System.out.println("Main thread interrupted");

}

// чекати завершення потоку

try {

System.out.println("Wait for threads to end.");
```

```
ob1.t.join();

  ob2.t.join();

  } catch (InterruptedException e) {

  System.out.println("Main thread interrupted.");

  }

System.out.println("Main thread ended.");

}

}
```

2.17. Тупиковий стан у розподіленій системі (Distributed Deadlock).

Централізоване виявлення тупика

Перший алгоритм (див. Рис.2.33).

Центральний координатор підтримує системний глобальний граф очікування (wait-for graph – WFG). Коли доцільно, він перевіряє WFG на наявність циклів. WFG – це RAG мінус ресурси; він показує, що потік чекає на ресурс, захоплений іншим потоком.

Всі сайти (sites) захоплюють та звільняють ресурси шляхом відправлення відповідних повідомлень до координатора. Коли координатор отримує запит, він оновлює WFG, робить перевірку на тупики і дозволяє запит, якщо немає тупиків. Коли координатор отримує повідомлення про звільнення, він оновлює WFG

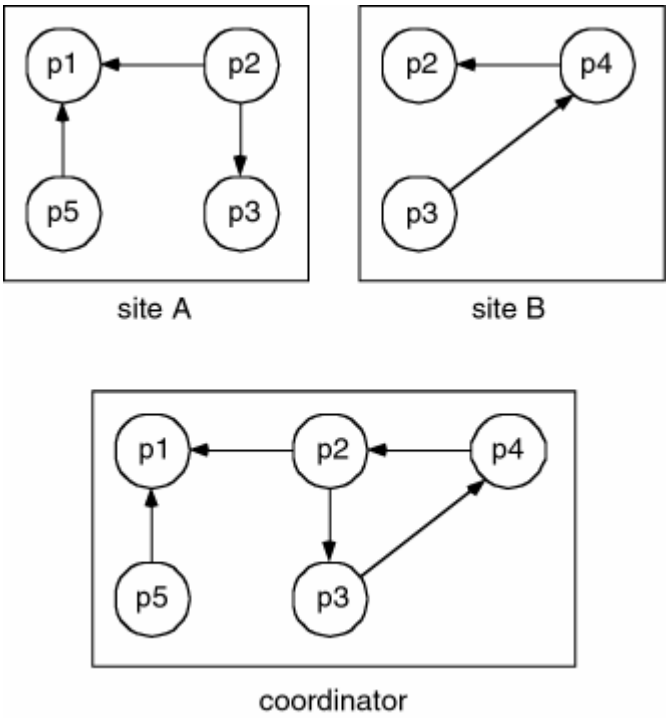


Рис. 2.33. Приклад використання першого алгоритму.

Цикл у глобальному WFG $\Rightarrow$ тупик

Немає циклу у глобальному WFG $\Rightarrow$ немає тупика

Другий алгоритм

Центральний координатор підтримує глобальний WFG системи. Індивідуальні сайти також підтримують локальні WFGs для локальних процесів та ресурсів. Глобальний WFG є наближенням загального стану системи

Коли координатор має оновлювати WFG та намагатися виявити тупики?

Завжди, коли нове ребро вставляється або видаляється у локальному WFG. Сайт інформує координатор, посилаючи повідомлення. Глобальний WFG може бути трохи застарілим.

Періодично, коли з WFG відбулися певна кількість змін. Сайт посилає кілька змін за раз. Глобальний WFG може бути більш застарілим

Завжди, коли потрібно виявити тупик.

Після виявлення тупика, координатор вибирає “жертву”, та каже всім сайтам, які застосувати дії.

Розподілене виявлення тупиків.

Індивідуальні сайти підтримують локальні WFGs. Кожний .WFG містить звичайні вузли для локальних процесів та вузли “Pex”, які представляють зовнішні процеси.

Виявлення тупиків.

Якщо сайт S_i знаходить цикл, що не включає Pex, то він знайшов тупик.

Якщо сайт S_i знаходить цикл, що включає Pex, існує можливість тупика.

Він посилає повідомлення, що містить виявлений ним цикл, будь-якому сайту, включеному у Pex.

Якщо сайт S_j отримує таке повідомлення, він оновлює свій локальний WFG та перевіряє його на цикли. Якщо S_j знаходить цикл, що не включає його Pex, він знайшов тупик.

Якщо S_j знаходить цикл, що включає його Pex, він висилає попередження про можливий тупик. Можна помилитися і повідомити про неіснуючий тупик.

Приклад:

Початковий стан (Рис. 2.34):

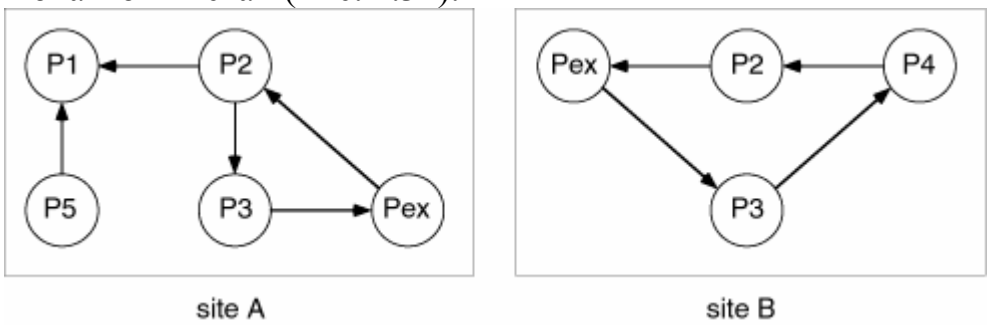
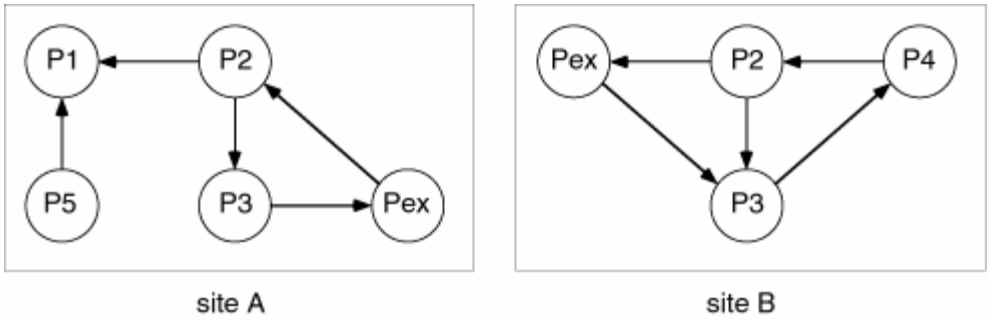


Рис Сайт А виявляє цикл, посилає повідомлення, що описує цей цикл, до сайту В.

Виявлення тупика (Рис. 2.35):



Сайт B оновлює свій WFG, знаходить цикл, що не включає Pex $\Rightarrow$ тупик

Ієрархічне виявлення тупиків.

Сайти організовані ієрархічно. Сайт відповідає лише за виявлення тупика, що включає його "дітей". Сайти (контролери) організовані у вигляді дерева. Контролери-листки керують ресурсами. Кожен підтримує локальний WFG, що стосується лише його ресурсів. Внутрішні контролери відповідають за виявлення тупиків. Кожен підтримує глобальний WFG, що об'єднує WFGs його дітей і виявляє тупик серед своїх дітей. Завжди, коли контролер змінює свій WFG, він передає зміни до свого батька. Батько оновлює свій WFG, та шукає цикли, передає зміни далі вгору.

Після виявлення тупика: відновлення

У залежності від умов використання ОС треба приймати рішення по відновленню функціонування системи на основі аналізу наступних питань.

Як часто виявляти тупики, після кожного запиту на ресурс або менш часто, кожну годину або коли використання ресурсів менше?

Якщо ОС виявить тупик, що робити? Закінчити процес, всі тупикові процеси, або один процес за раз, поки немає тупика?
Який процес закінчити, процес з більшим ресурсом, з меншою вартістю?

Чи це взагалі прийнятно, закінчити процес? Чи є якась краща альтернатива? Чи можна *відкатити* – повернути процес до певного безпечного стану, та розпочати знову?

2.18. Віртуальна пам'ять.

Структура фізичної пам'яті.

Головна пам'ять.

Розглянемо структуру головної пам'яті, способи підвищення її ефективності і надійності та способи захисту.

Розшарування адрес.

Звичайно адресація головної пам'яті виконується з точністю до байта, а звернення до неї здіснюється самим процесором. При зверненні до головної пам'яті з метою прискорення цього процесу запис та зчитування кількох байтів може здіснюватись за один раз. Для цього головна пам'ять розділяється на кілька незалежних блоків (банків). У системі адресації застосовується процедура розшарування.

Суть цієї процедури полягає в тому, що при кількості блоків n адреса a, що надходить із процесора, відноситься до блоку (a) mod n. Наприклад, при чотирьох блоках пам'яті адресація виконується у відповідності зі структурою, показаною на Рис. 2.36.

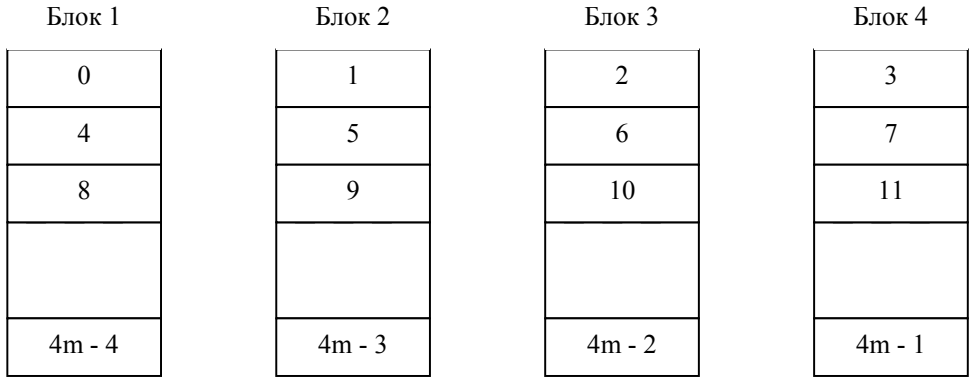


Рис. 2.36. Розшарування головної пам'яті.

При звертанні по послідовним сусіднім адресам можна досягти n-кратного збільшення швидкості звертання за рахунок паралельної дії всіх блоків. Така структура розподілення адрес по блоках називається n-розшаруванням звертань до пам'яті.

Звичайно n дорівнює 2÷16, але досягає у деяких випадках 64.

Заходи до підвищенню надійності.

Для підвищення надійності звертання до головної пам’яті запис та зчитування даних виконується з використанням надлишкових кодів. Якщо достатнім вважається тільки визначення наявності помилок, то можливо обійтися тільки перевіркою на парність. Але можна використовувати коди з виправленням помилок. Помилки бувають у більшості випадків поодинокими, і тому надлишкових

бітів для їх виправлення потрібно не так уже й багато. Наприклад, для корекції всіх поодиноких помилок у 4-бітній послідовності достатньо збільшити ііі довжину до 7 біт. Це один з прикладів так званого кода Хемінга, що має властивість виправлення поодиноких і визначення наявності подвійних помилок.

Захист пам'яті.

При мультипрограмній роботі комп'ютера необхідно забезпечити таке розміщення кожної програми у пам'яті, щоб вона не руйнувалась іншими програмами. Крім того, для забезпечення збереження інформації її треба захистити від несанкціонованого доступу. Звичайно захист пам'яті забезпечується за рахунок взаємодії спеціальних засобів операційної системи та апаратури.

Одним з механізмів подібного захисту є система ключів. Суть цієї системи у тому, що головна пам'ять розділяється на блоки певної довжини і для кожного блока встановлюється ключ. Ця система обмежує зчитування та запис шляхом порівняння ключів виконуваних програм з ключами захисту. У випадку, коли ключі не співпадають, звертання вважається недопустимим і наступає переривання виконання програми.

Кеш-пам'ять.

Буферна пам'ять (кеш-пам'ять) призначена для підвищення швидкодії процесу звертання до головної пам'яті.

У її структуру входить Масив даних та Довідник. У Масив даних копіюються відповідні блоки головної пам'яті, а їхні адреси заносяться у Довідник. Блок обробки команд процесора звичайно звертається до кеш-пам'яті. При відсутності у кеш-пам'яті потрібних байтів вони переписуються до неї з головної пам'яті.

При цьому час звертання процесора до пам'яті дорівнює:

$$t_a = t_b + \alpha t_m,$$

де

t_b – час звертання до кеш-пам'яті,

t_m – час звертання до головної пам'яті,

α – ймовірність відсутності у кеш-пам'яті потрібної інформації (ймовірність кеш-промаху).

Звичайно tb менше tm на порядок і при достатньому зменшенні α можна досягти $t_a \approx t_b$.

У міру вдосконалення технології збільшується швидкість дії процесора. При цьому скорочується і t_b –час звертання до кеш-пам'яті. Однак, у зв'язку з постійно зростаючим об'ємом головної пам'яті, час звертання до головної пам'яті зменшити важко, і тому різниця між t_b і t_m збільшується, що веде до зниження ефективності

кеш-пам’яті. Цю тенденцію можна подолати за рахунок використання двохрівневої кеш-пам’яті.

Система управління кеш-пам’яттю.

Головна та кеш-пам’яті розбиваються на блоки, у ході звертання процесору до пам’яті виконується порівняння потрібних адрес з адресами, які знаходяться у довіднику кеш-пам’яті. При наявності у довіднику потрібного блоку (кеш-попадання) виконується вибірка інформації з кеш-пам’яті.

З метою спрощення механізму порівняння адрес запропоновані різні способи організації взаємодії головної та кеш-пам’яті. З них найбільш типовим є спосіб асоціативної обробки адрес.

У цій системі головна та кеш пам’яті розділяються на кілька наборів блоків; розміщення блоків відбувається у межах відповідних наборів.

Значення поля, що визначає адресу набору, відправляється в довідник і вміст довідника по цій адресі порівнюється зі старшими розрядами адреси. Їх збіг означає наявність указаних блоків у відповідній частині масива даних кеш-пам’яті.

Вибір у кеш-пам’яті блоків, які можуть бути замінені на ті, що поступають, виконується за допомогою стратегій LRU (Least Recently Used – той, що довше всіх не використовувався), FIFO (First In First Out) та інших.

Найбільш раціональною є стратегія LRU, у відповідності з якою з кеш-пам’яті видаляється блок, останнє звертання до якого мало місце раніше, ніж до інших блоків.

Запис даних у кеш-пам’ять та головну пам’ять виконується одним з наступних способів:

- наскрізний запис (write through);
- запис з перекачкою (Swap або Store in, write back).

У відповідності з першим способом блок, що записується у кеш-пам’ять одночасно записується у головну пам’ять, у результаті чого вміст одноіменних блоків на обох рівнях пам’яті звичайно є однаковим. При використанні другого способу блок, що записується у кеш, у головну пам’ять не заноситься. При підкачці кеша, коли об’єктом заміни стає блок, вміст якого у кеші оновлювався, дані цього блоку передаються у головну пам’ять.

Сторінкова організація пам’яті (Paging).

Paging є принциповою складовою архітектури сучасного ком’ютера. Коли була розроблена архітектура IBM-360 здавалось, що вона вже вічна і що подальший процесс неможливий. Але

незабаром в Англії з Paging є принциповою складовою архітектури сучасного комп'ютера. Коли була розроблена архітектура IBM-360 здавалось, що вона вже вічна і що подальший прогрес неможливий. Але незабаром в Англії з"явився Paging; через 15 років була прийнята архітектура IBM-370, що відрізнялася від IBM-360 саме Paging-ом. Paging був реалізований і у комп'ютері БЭСМ-6.

Концепції управління пам'яттю.

Прикладні програми бачать пам'ять, як свій віртуальний адресний простір (Рис. 2.37). ОС бачить пам'ять як фізичний адресний простір. Пристрій управління пам'яттю (MMU-hardware) використовується для імплементації сегментації, сторінкової організації або їх комбінації, виконуючи трансляцію адрес.

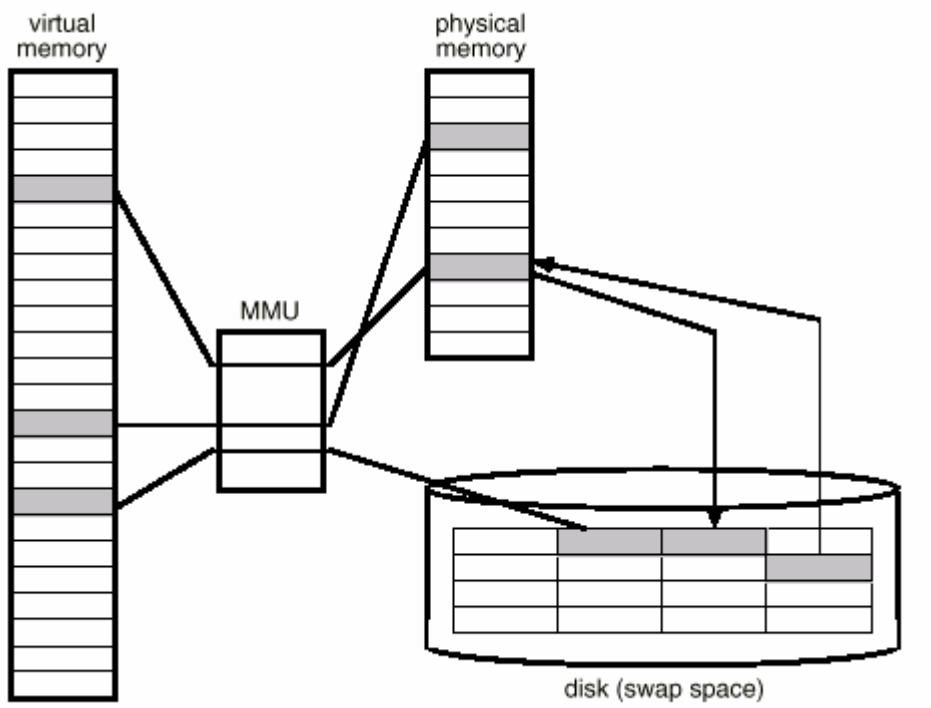


Рис. 2.37 Організація віртуальної пам'яті.

Підкачка (Paging) по запиту (Virtual Memory).

Сторінка віртуальної пам'яті може бути збережена одним з двох способів: у фреймі або у фізичній пам'яті на диску (backing store, or swap space).

Процес може виконуватись тільки з частиною свого віртуального адресного простору у головній пам’яті. Створюється ілюзія майже необмеженої пам’яті.

Запуск нового процесу

Процеси стартують з 0 або більше своїх віртуальних сторінок у фізичній пам’яті, і “відпочивають” на диску.

Вибір сторінки – коли нові сторінки вводяться у фізичну пам’ять?

Випереджуюча підкачка сторінок (Prepaging) – попереднє завантаження при старті: code, static data, one stack page (DEC ULTRIX)

Підкачка по запиту – старт з 0 сторінки, завантаження кожної сторінки по запиту, коли трапляється відмова сторінки (найбільш загальний метод). Недолік: багато відмов сторінок, коли програма починає виконуватись.

Підкачка по запиту працює відповідно до принципу локалізації посилань (locality of reference). Кнут оцінив, що протягом 90% часу програма використовує 10% команд.

Виняток у зв’язку з відсутністю сторінки в оперативній пам’яті (Page Faults).

Спроби звенутись до сторінки, якої немає у фізичній пам’яті, спричиняють внутрішнє переривання процесора (в архітектурі Intel x86 воно називається “винятком”) – “відмову сторінки” - Page Faults.

Таблиця сторінок повинна включати present біт (інакше називається valid біт) для кожної сторінки. Спроба звернутись до сторінки без present біту дає у результаті “відмову сторінки”, що спричиняє перехід до ОС.

Коли трапляється відмова сторінки,ОС повинен:

підкачати (page in) сторінку – повернути її з диску на вільний фрейм у фізичній пам’яті;

оновити таблицю сторінок і present біт.

забезпечити, щоб процес, у якому була відмова, продовжував виконуватись.

На відміну від переривиння, відмова сторінки може виникнути у будь-який час при звертанні до пам’яті. Навіть у середині команди!

Однак, обробка відмови сторінки повинна бути непомітною для процесу, що її викликав. Обробник відмови сторінки мусить бути здатним відновити попередній стан машини (на момент відмови),

щоб продовжити виконання програми (це потребує hardware – підтримки).

Заміщення сторінок.

Коли ОС потребує фрейм для розміщення процесу,але всі фрейми зайняті, він повинен виселити (скопіювати до допоміжного запоминаючого пристрою) сторінку з якогось фрейму.

Як ми обираємо сторінку для заміщення?

Деякі політики заміщення сторінок:

Random – для виселення обирається яка – небудь випадкова сторінка;

FIFO - виселяється сторінка, що перебуває у пам’ять найдовше,
використовується черга для відслідкування.

Ідеєю є дати всім сторінкам “справедливе” (рівне) використання пам’яті.

Розміщення фреймів

Як багато фреймів використовує кожний процес (M фреймів, N процесів)?

Хоча б 2 фрейма (один для команд, один для операндів у пам’яті), можливо більше ...

Максимумом є кількість у фреймів у фізичній пам’яті.

Алгоритми розміщення:

при розміщенні порівну кожному процесу виділяється M/N фреймів;

при пропорціональному розміщенні кількість виділених фреймів залежить від розміру і приоритету процесу.

Який пул фреймів використовується для заміщення?

Локальне заміщення – процес може повторно використати тільки свій власний фрейм.

Глобальне заміщення - процес може повторно використати який завгодно фрейм (навіть використаний іншим процесом).

Пробуксовка (Thrashing).

Розглянемо, що відбувається, коли пам’ять бере надмірні зобов’язання (gets over committed).

Після кожного виконання процесу, до того, як він обирається для виконання знову, всі його сторінки можуть бути відкачані (paged out).

Наступного разу коли процес виконується, ОС буде витрачати деякий час на відмову сторінки (page faulting), і повертати сторінки назад.

Може виникнути ситуація, при якій весь час витрачається на підкачку і не дається часу на корисну роботу. З підкачкою по запиту ми бажаємо дуже великої віртуальної пам'яті, яка була б такою швидкою, як фізична пам'ять яле замість цього отримуємо таку повільну, як диск!

Ця втрата активності, обумовлена частим Paging-ом називається **пробуксовкою (thrashing).**

Пробуксовка виникає, коли сума всіх потреб процесів більша, ніж фізична пам'ять.

Робочі комплекти.

Робочий комплект (Working set), це колекція сторінок, з якими процес працює, і які мусять бути резидент ними у головній пам'яті, щоб запобігти пробуксовці. Робочий комплект завжди зберігається у пам'яті. Інші сторінки можуть бути відкинутими, якщо це потрібно.

2.19. Сегментація та сторінкова організація в архітектурі Intel x86.

Сегментація.

Основна ідея – використовуючи програмістський погляд на програму, поділити процес на окремі сегменти в пам'яті.

Кожний сегмент має окрему ціль. Приклади: код, статичні дані, купа, стек (code, static data, heap, stack).

Можливо, код та стек будуть мати окремий сегмент для кожної функції. Тоді сегменти можуть бути різними за розміром, а стек та купа не будуть конфліктувати. Процес все ще завантажується в пам'ять цілком, але сегменти, які складають, процес не обов'язково повинні завантажуватися суміжно у пам'ять. Простір в межах сегментів безперервний.

Кожний сегмент має біти захисту:

сегмент Read-only - тільки для читання (код);

сегмент Read-write - для читання та запису (дані, купа, стек);

дозвіл процесам сумісно використовувати код та дані.

Адресація сегментів.

Віртуальна (логічна) адреса складається з номеру сегмента та зсуву (Offset) з початку цього сегмента. Обидві складові генеруються ассемблером.

Що вказується в адресній частині команди?

Простий метод -вищі біти адреси визначають сегмент, нижні біти адреси визначають зсув.

Неявна специфікація сегмента - сегмент вибирається неявно виконуваною командою. Приклади: PDP-11, Intel x86

Явна специфікація сегменту - префікс команди може вимагати щоб використовувався вказаний сегмент. Приклади: PDP-11, Intel x86

Імплементація сегментів.

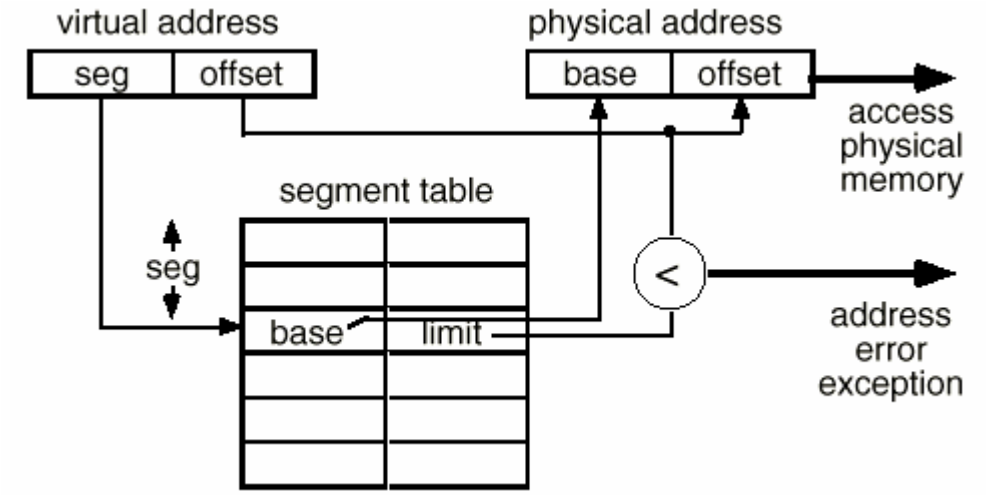


Рис.2.38. Сегментація.

Таблиця сегментів (Рис. 2.38) процесу містить строку для кожного сегменту. Кожний запис таблиці сегментів містить base та limit а також інформацію захисту (дозволи на розділення - sharing, читання – read, читання/запис - read/write).

Необхідна додаткова апаратна підтримка: кілька base та limit регістрів або вказівник бази таблиці сегментів (вказує на таблицю в пам’яті).

Приклад сегментації.

Відводиться 2 біта для номеру сегменту, 12 біт для зсуву.
Частина таблиці сегментів має такий вигляд:

segment	base	limit	R W
0	0x4000	0x6FF	1 0
1	0x0000	0x4FF	1 1
2	0x3000	0xFFF	1 1
4			0 0

Відображення адресного простору для цієї частини таблиці сегментів подано на Рис. 2.39.

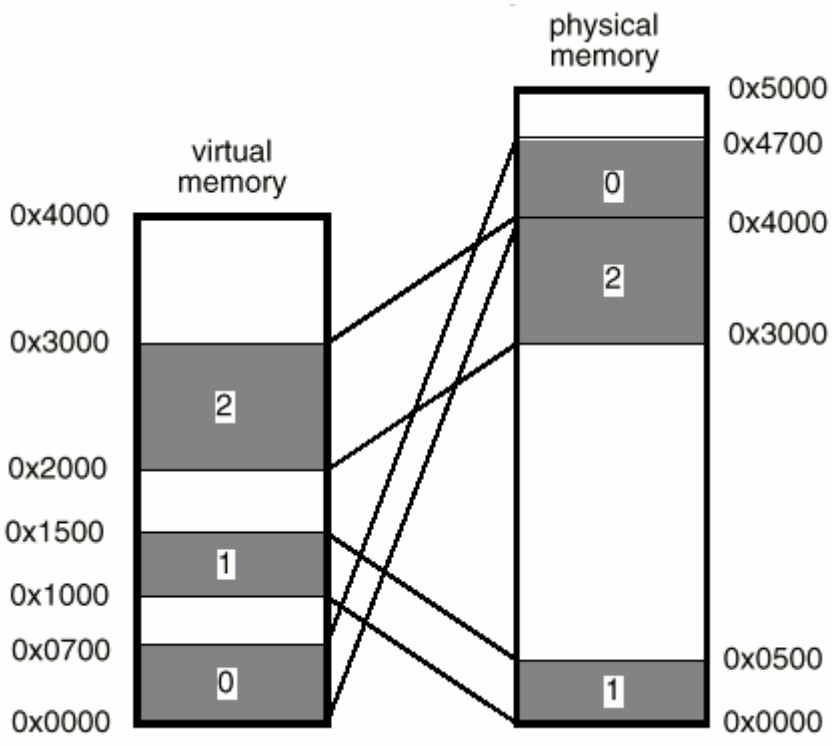


Рис.2.39 Відображення адресного простору.

Управління сегментами.

Коли процес створюється, розподіляється простір віртуальної пам'яті для всіх сегментів процесу, складається та розміщується в PCB процесу таблиця сегментів, головним чином пуста.

Коли відбувається перемикання контексту:

зберігається таблиця сегментів ОС в старому PCB процесу;

завантажується таблиця сегментів ОС з нового PCB процесу, та розподіляється місце в фізичній пам'яті, якщо спочатку процес завантажується вперше.

Якщо не має місця в фізичній пам'яті:

потрібно стиснути пам’ять (перерозмістити сегменти обновити base) зробити неперервним адресний простір;

перекачати (Swap) один або більше сегментів з пам’яті на диск.

Щоб виконувати цей процес знову, треба перекачати всі його сегменти назад у пам’ять.

Сторінкова організація (Paging).

Порівняння сегментації та сторінкової організації:

- paging реалізує розподіл пам’яті та підкачку більш легко;
- ніякої зовнішньої фрагментації;
- кожен процес поділяється на множину маленьких, фіксованого розміру частин, так званих сторінок (pages);
- фізична пам’ять поділяється на велику кількість маленьких, розміру частин, так званих фреймів (frames).

Сторінки не мають ніякого відношення до сегментів. Розмір сторінки = розміру фрейму і складає звичайно від 512 bytes до16K bytes. Процес все ще цілком завантажується в пам’ять, але сторінки процесу не повинні бути завантажені в неперервний набір фреймів. Віртуальна адреса складається з номера сторінки та зміщення (offset) від початку цієї сторінки.

Імплементація сторінкової організації.

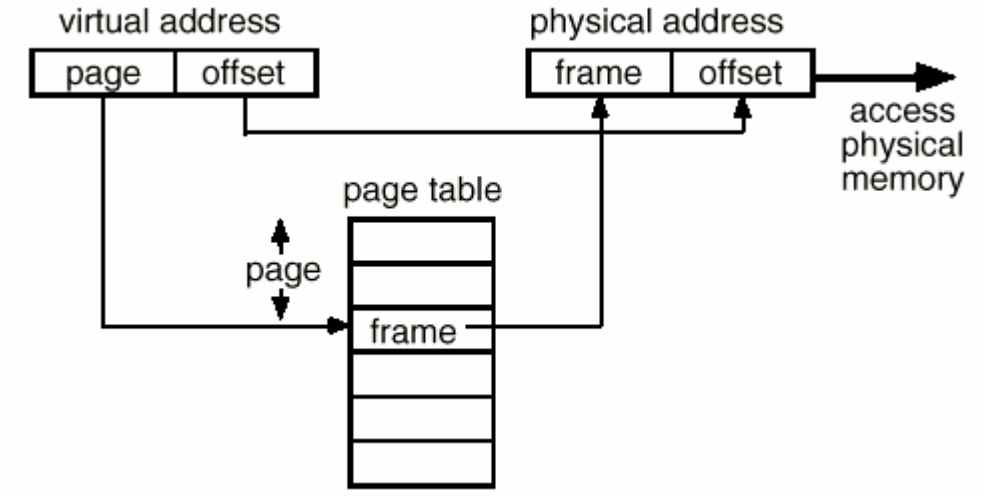


Рис. 2.40. Сторінкова організація

Таблиця сторінок (Рис. 2.40) процесу (page table) зберігає запис для кожної сторінки, що містить посилання на фрейм у головній

(фізичній) пам’яті. Можна додавати біти захисту, але не вони такі потрібні.

Потрібна додаткова апаратна підтримка злегка меншої складності, ніж для сегментації. Ніякої потреби зберігати запис та порівнювати з лімітом, Чому?

Управління сторінками і фреймами.

ОС звичайно зберігає трек вільних фреймів в пам’яті, використовуючи бітову карту. Бітова карта відображає тільки масиви бітів:

1 означає, що фрейм вільний;

0 означає, фрейм розподілений сторінці.

Щоб відшукати вільний фрейм треба знайти одиничний біт в бітовій карті.

Найбільш сучасні набори команд містять команди які повертають в регістрі зміщення першого одиничного біта.

Page table base pointer (special register) (спеціальний регістр) вказує на page table активного процесу

Saved/restored як частина контекстного перемикання.

Page table також містить valid біт який вказує, що сторінка (page) знаходиться у віртуальному адресному просторі, а не на диску.

2.20. Розподілена загальнодоступна пам’ять (Distributed Shared Memory).

Базисна ідея (Kai Li, 1986)

Група робочих станцій, сполучених локальною мережею, спільно використовує сторінки віртуального адресного простору, об’єднуючись таким чином у систему DSM (Distributed Shared Memory).

Кожна сторінка є присутньою точно на одній машині, і посилання до локальної сторінки виконується у звичайному режимі

Посилання до віддаленої сторінки викликає апаратне внутрішнє переривання – “відмова сторінки”, обробляючи яке OS посилає повідомлення віддаленій машині, щоб одержати сторінку.

Команда, що викликала переривання, може потім бути повторена і виконана.

Для програміста DSM - система виглядає, як стандартна машина з загальнодоступною пам'яттю

Просто пристосувати старі програми.

Недостатня ефективність - велика кількість сторінок посилається через мережу вперед і назад

Моделі Несуперечливості

У розподіленій системі неможливо встановити для кожної операції унікальну помітку часу, погоджену з «Реальним часом». Але як тоді забезпечити несуперечливість у роботі DSM ?

Строга несуперечливість (найдужча модель)

Значення, повернуте операцією читання завжди таке саме, як значення, записане самою останньою операцією запису

Працює подібно централізованій системі пам'яті, записи становляться негайно доступними усім. Зберігається порядок «Реального часу» . Неможливо імплементувати.

Несуперечливість послідовності (Lamport 1979)

Всі процеси бачать усі записи пам'яті в тому ж самому порядку (але не обов'язково « Програмному порядку »)

Перестановка порядку записів між процесами дозволяється, оскільки усі процеси бачать те ж саме упорядкування. Порядок «Реального часу» не витримується.

Операція читання не може повертати результат самої останньої операції запису! Якщо це суттєво, використовуйте семафори!

Причинна несуперечливість.

Всі процеси бачать всю причинно – зв’язану пам'ять записаною в тому ж самому порядку. Перестановка порядку записів між процесами дозволяється, оскільки усі процеси бачать те ж саме упорядкування причинно – зв’язаних записів.

Два посилання до пам'яті потенційно причинно – зв’язані, якщо перша потенційно впливає на другу, наприклад:

записати змінну, потім читати цю змінну;

записати змінну, потім записати її знову.

Розділені (що не перетинаються) записи можуть бути конвеєризовані для підвищення ефективності системи пам'яті.

Потрібно підтримувати граф залежності, стежити, котрі операції залежить від інших операцій

Несуперечливість процесора.

Всі процеси бачать усі записи з індивідуального процесора у програмному порядку, і всі записи до тієї ж самої пам'яті розташовані в тому ж самому порядку.

Розділені(що не перетинаються) запису, і записи одного процесора, можуть бути конвеєризовані.

PRAM несуперечливість.

Всі процеси бачать усі записи з індивідуального процесора у програмному порядку Запис з різних процесорів може бути видимим в будь-якому порядку.

PRAM = pipelined (конвеєризована) RAM

Записи одним процесором можуть бути конвеєризовані, щоб підвищити ефективність; процесор не повинен чекати закінчення однієї операції до початку іншої.

Слабка несуперечливість.

Послідовна несуперечливість застосовується тільки до синхронізованого доступу, а не до індивідуального доступу до пам'яті. Синхронізація доступів діє як бар'єр. Всі попередні записи повинні бути завершені до початку синхронізованого доступу.

Весь синхронізований доступ повинен бути завершений перед новим доступом для читання / запису. Нормальний доступ до пам'яті може бути конвеєризованим.

Несуперечливість звільнення.

Послідовна несуперечливість застосовується тільки до блокування операціями acquire (оволодівання) та release (звільнення).

Блокування оволодіванням – acquire – повинно виконуватись перед початком нового читання/запису, але не треба чекати, щоб завершились попередні записи

Перед звільненням – release - повинні завершитися попередні записи, але не треба чекати початку нового читання/ запису.

Порівняння моделей несуперечливості

Наведені вище моделі несуперечливості відрізняються по ступені обмеженості, складності реалізації, легкості у використанні, і ефективності.

Строга несуперечливість - найбільше обмежувальна, але неможлива для виконання.

Послідовна несуперечливість – інтуїтивна семантично, але не дозволяє значного паралелізму. Використовується у DSM системах. Різновидність, що називається *relaxed memory* (ослаблена пам'ять), використовується в комерційних системах пам'яті (дозволяють, щоб

читання і записи бути переупорядкованими, якщо вони звертаються до різних розташувань пам'яті).

Каузальна, процесорна і PRAM моделі несуперечливості – дозволяють більшу кількість паралелізму, але мають не інтуїтивну семантику, і покладають особливий тягар на програміста.

Слабка і "звільнення" несуперечливості – мають інтуїтивну семантику, але покладають додатковий тягар на програміста.

Імплементація послідовної несуперечливості у основаній на сторінковій організації DSM

Чи сторінка може рухатися? …, скопіюватися?

Сторінки не копіюються, не переміщаються.

Всі запити до сторінки повинні посилатись власнику сторінки.

Просто забезпечувати послідовну несуперечливість – власник впорядковує всі запити доступу.

Ніякого паралелізму.

Сторінки не копіюються, але переміщаються

Всі запити до сторінки повинні посилатись власнику сторінки

Щоразу, коли запитується віддалена сторінка, вона переміщається на процесор, що звернувся до неї.

Просто забезпечувати послідовну несуперечливість - тільки процеси на цьому процесорі можуть звернутися до сторінки.

Ніякого паралелізму

Сторінки копіюються та переміщаються

Всі запити до сторінки повинні посилатись власнику сторінки.

Щоразу, коли запитується віддалена сторінка, вона переміщається на процесор, що звернувся до неї.

Кілька операцій читання можуть бути виконані одночасно.

Щоб забезпечити послідовну несуперечливість – треба оголосити невірними (найбільш загальний підхід) або протягом операції запису модифікувати інші копії сторінки.

Сторінки копіюються, але не переміщаються

Сторінки копіюються у фіксовані розташування.

Всі запити до сторінки повинні бути послані одному з власників сторінки.

Щоб забезпечити послідовну несуперечливість – треба модифікувати інші копії сторінки протягом операції запису.

2.21. Ввод-вивід

Керування вводом-виводом

Однією з головних функцій ОС є керування всіма пристроями вводу-виводу комп'ютера. ОС повинна передавати пристроям команди, перехоплювати переривання й обробляти помилки та підтримувати інтерфейс між пристроями й іншою частиною системи.

Для забезпечення можливості розвитку ОС інтерфейс повинен бути однаковим для всіх типів пристроїв (незалежність від пристроїв).

Щоб виконати ввод/вивід, програми користувача повинні робити системний виклик до ОС. Коли процес користувача робить системний виклик, відбувається згенероване програмним забезпеченням переривання (trap), що викликає встановлення привілейованого режиму (Kernel mode) та передачу управління (з використанням вектора переривання) відповідному оброблювачу переривання.

Оброблювач переривання зберігає стан процесу, виконує запитаний ввод/вивід (якщо запит відповідний), відновлює збережений ним стан, установлює непривілегійований режим, і повертається до викликаної програми.

Фізична організація пристроїв вводу-виводу

Пристрою вводу-виводу поділяються на два типи: блок-орієнтовані пристрої і байт-орієнтовані пристрої.

Блок-орієнтовані пристрої зберігають інформацію в блоках фіксованого розміру, кожний з який має свою власну адресу. Найпоширеніший блок-орієнтований пристрій - диск.

Байт-орієнтовані пристрої не адресовані і не дозволяють здійснювати операцію пошуку. Вони генерують чи використовують послідовність байтів. Прикладами є термінали, рядкові принтери, мережні адаптери.

Однак деякі зовнішні пристрої не належать до жодного класу, наприклад, годинник, що, з одного боку, не адресований, а з іншого боку, не породжує потік байтів. Цей пристрій тільки видає сигнал переривання в деякі моменти часу.

Зовнішній пристрій звичайно складається з механічного й електронного компонента**.** Електронний компонент називається контролером пристрою чи адаптером. Механічний компонент власне є пристроєм. Деякі контролери можуть керувати декількома пристроями.

Якщо інтерфейс між контролером і пристроєм стандартизований, то незалежні виробники можуть випускати сумісні як контролери, так і пристрої.

Операційна система зазвичай має справу не з пристроєм, а з контролером. Контролер, як правило**,** виконує прості функції, наприклад, перетворює потік біт у блоки, що складаються з байт, і здійснює контроль і виправлення помилок. Кожен контролер має кілька регістрів, що використовуються для взаємодії з центральним процесором. У деяких комп'ютерах ці регістри є частиною фізичного адресного простору. У таких комп'ютерах немає спеціальних операцій вводу-виводу. В інших комп'ютерах адреси регістрів вводу-виводу, що називаються портами, утворюють власний адресний простір через введення спеціальних операцій вводу-виводу (наприклад, команд IN і OUT у процесорах Intel x86).

ОС виконує ввід-вивід, записуючи команди в регістри контролера. Наприклад, контролер гнучкого диска IBM PC приймає 15 команд, таких як READ, WRITE, SEEK, FORMAT і т.д. Коли команда передана, процесор залишає контролер її виконувати в асинхронному режимі і займається іншою роботою. При завершенні команди контролер організує переривання для того, щоб передати керування процесором операційній системі, що повинна перевірити і використати результати операції. Процесор отримує результати операції і статус пристрою, читаючи інформацію з регістрів контролера.

Програмне забезпечення вводу-виводу

Основна ідея організації програмного забезпечення вводу-виводу полягає в розбивці його на кілька рівнів, причому нижні рівні забезпечують екранування особливостей апаратури від верхніх, а ті, у свою чергу, забезпечують зручний інтерфейс для користувачів.

Ключовим принципом є незалежність від пристроїв. Вигляд програми не повинен залежати від того, чи читає вона дані з гнучкого чи диска з твердого диска.

Дуже близької до ідеї незалежності від пристроїв є ідея однакового іменування, тобто для іменування пристроїв повинні бути прийняті єдині правила.

Іншим важливим питанням для програмного забезпечення вводу-виводу є обробка помилок. Узагалі, помилки варто обробляти якнайближче до апаратури. Якщо контролер виявляє помилку читання, то він повинен спробувати її скорегувати. Якщо ж це йому не вдається, то виправленням помилок повинен зайнятися драйвер пристрою.

Багато помилок можуть зникати у повторних спробах виконання операцій вводу-виводу, наприклад, помилки, викликані наявністю порошинок на голівках читання чи на диску. І тільки якщо нижній рівень не може впоратися з помилкою, він повідомляє про помилку верхньому рівню.

Ще одне ключове питання – це використання блокувальних (синхронних) і неблокувальних (асинхронних) передач.

Більшість операцій фізичного вводу-виводу виконується асинхронно - процесор починає передачу і переходить на іншу роботу, поки не настає переривання. Користувачам набагато легше писати програми, якщо операції вводу-виводу блокувальні – після команди READ програма автоматично припиняється доти, поки дані не потраплять у буфер програми. ОС виконує операції вводу-виводу асинхронно, але представляє їх для програм користувачів у синхронній формі.

Остання проблема полягає в тому, що одні пристрої є поділюваними, а інші – виділеними**.** Диски – це поділювані пристрої, тому що одночасний доступ декількох користувачів до диска не є проблемою**.** Принтери – це виділені пристрої, тому що не можна змішувати рядки, що друкуються різними користувачами. Наявність виділених пристроїв створює для операційної системи деякі складнощі.

Для рішення поставлених проблем доцільно розділити програмне забезпечення вводу-виводу на чотири шари (див. Рис.2.41):

обробка переривань;

драйвери пристроїв;

незалежний від пристроїв шар операційної системи;

користувальницький шар програмного забезпечення.

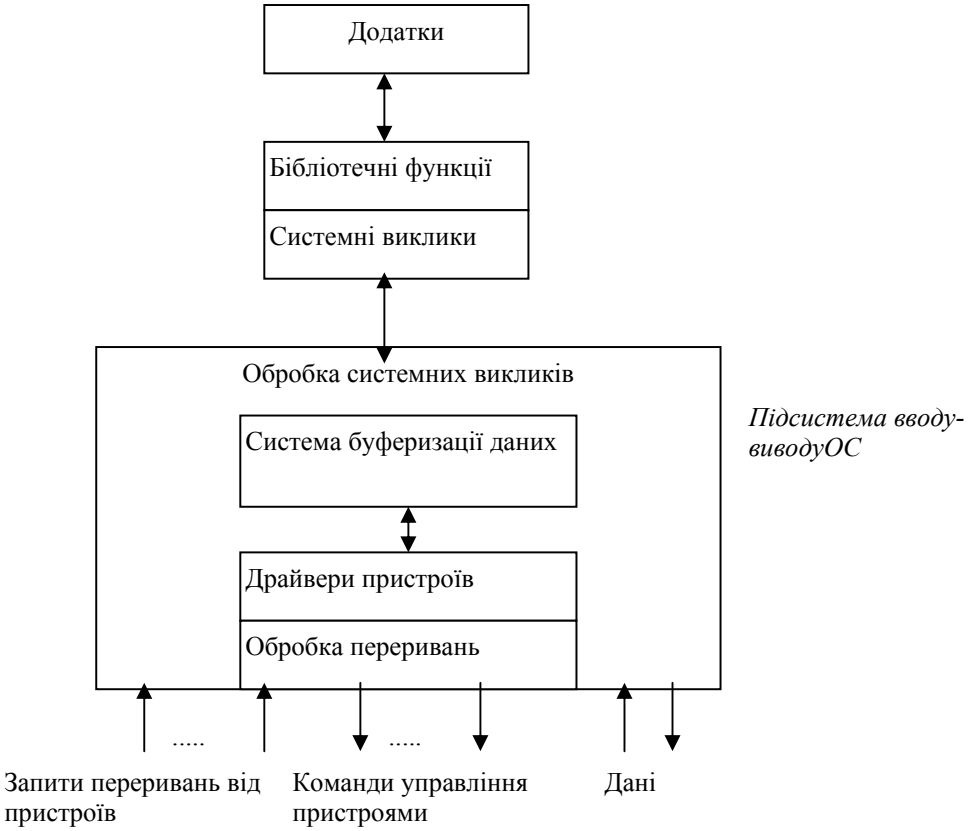


Рис. 2.41 Багаторівнева організація підсистеми вводу-виводу

Обробка переривань

Переривання повинні бути сховані якнайглибше в надрах операційної системи, щоб як мога менша частина ОС мала з ними справу. Найкращий спосіб складається в дозволі процесу, що ініціював операцію вводу-виводу, блокувати себе до завершення операції і настання переривання. Процес може блокувати себе, використовуючи, наприклад, виклик WAIT для семафора чи для умовної змінної, або виклик RECEIVE для очікування повідомлення. При настанні переривання процедура обробки переривання виконує розблокування процесу, що ініціював операцію вводу-виводу, використовуючи виклики UP, SIGNAL або посилаючи процесу повідомлення. У будь-якому випадку ефект від переривання буде полягати в тому, що раніше заблокований процес тепер продовжить своє виконання.

Система переривань процесорів сімейства Intel - x86.

Нормальний хід виконання програми процесором – це коли після виконання кожної команди лічильник команд інкрементується, або у нього завантажується адреса передачі управління.

Переривання та винятки порушують нормальний хід виконання програми для обробки зовнішніх подій або для повідомення про виникнення особливих умов чи помилок.

Переривання поділяються на апаратні, що маскуються (тобто можуть бути замаскованими) та що не маскуються, які викликаються електричними сигналами, та програмні, які виконуються по команді INT XX. Програмні переривання обробляються процессором як різновид винятків.

Процессор може сприймати переривання після виконання кожної команди.

Переривання, що маскуються, виконуються переходом на високий рівень сигнала на вході INTR (Interrupt Request) при піднятому прапорці дозволу Interrupt Flag (IF=1). У цьому випадку процессор зберігає у стеку регістр прапорців, опускає прапорець IF і виробляє два послідовних цикла підтвердження переривання, під час яких генеруються управляючі сигнали INTA# (Interrupt Acknowledge). Високий рівень сигналу INTR повинен зберігатися до підтвердження переривання.

Перший цикл підтвердження – холостий, (спадщина процессора 8080) по другому сигналу INTA# зовнішній контролер переривань передає по шині 8-розрядний номер переривання, що обслуговує даний тип апаратного переривання. Таким чином, процессор може виконувати 256 типів (номерів) переривань. Обробка поточного переривання може бути, у свою чергу перервана перериванням, що не маскується, а якщо програма обробки переривань підніме прапорець IF, то і іншим перериванням, що маскується.

Переривання, що не маскуються, виконуються незалежно від стану прапорця IF по сигналу NMI (Non Miscible Interrupt). Його обробка не може перериватись під дією нового сигнала на вході NMI.

Винятки (Exception) підрозділяються на відмови, ловушки та аварійні завершення.

Відмова (Fault) – це виняток, який визначається та обсуговується до виконання інструкції, що викликає помику. Після обслуговування цього винятку управління повертається знову на ту ж інструкцію, яка викликала відмову.

Відмови, що використовується у системі віртуальної пам”яті, дозволяють, наприклад, підкачати з диска в головну пам’ять потрібну сторінку або сегмент.

Ловушка (trap) – це виняток, який визначається та обслуговується після виконання інструкції, що його викликає. Після обслуговування цього винятку управління передається на інструкцію, наступну після інструкції, що викликає ловушку. До класу ловушок відносяться і програмні переривання.

Аварійне завершення (abort) – це виняток, що не дозволяє точно встановити інструкцію, яка його викликала. Він

використовується для повідомлення про серйозну помилку, таку як апаратна помилка або пошкодження системних таблиць.

Процедура, що обслуговує переривання або виняток, визначається по таблиці за допомогою номера – 8-бітного вказівника на вектор переривання.

Вказівник для програмних переривань задається командою. Для апаратних переривань, що маскуються, вказівник вводиться від зовнішнього контролера у другому циклі INTA#. Переривання, що не маскуються, мають фіксований вектор. Винятки генерують та передають вектор у самому процессорі.

Кожному номеру (0-255) переривання або винятку відповідає елемент у таблиці дескрипторів переривань IDT (Interrupt Descriptor Table). Таблиця містить 8-байтні дескриптори переривання і може бути розташована в якому завгодно місці фізичної пам”яті. Переривання та відмови процессора, наведені у наступній таблиці.

Номер	**Функція**	**Тип**
0	Ділення на 0	Fault
1	Виняток відладки	Fault/Trap
2	Немасковані переривання (NMI)	NMI
3	Переривання відладки (NMT 3)	Trap
4	Переповнення	Trap
5	Переривання по контролю	Fault
6	Недопустимий код операції	Fault
7	Співпроцессор недоступний	Fault
8*	Подвійна відмова	Abort
13	Загальне порушення захисту	Fault
14	Відмова сторінки	Fault
16	Виняток співпроцессора	Fault

Драйвери пристроїв

Весь залежний від пристрою код міститься в драйвері пристрою. Кожен драйвер керує пристроями одного типу чи, можливо, одного класу.

В операційній системі тільки драйвер пристрою знає про конкретні особливості якого-небудь пристрою. Наприклад, тільки драйвер диска має справу з доріжками, секторами, циліндрами, часом встановлення голівки й іншими факторами, що забезпечують правильну роботу диска.

Драйвер пристрою приймає запит від пристроїв програмного шару і вирішує, як його виконати. Типовим запитом є читання *n* блоків даних. Якщо драйвер був вільний під час надходження

запиту, то він починає виконувати запит негайно. Якщо ж він був зайнятий обслуговуванням іншого запиту, то новий запит приєднується до черги вже наявних запитів, і він буде виконаний, коли наступить його черга.

Після передачі команди контролеру драйвер повинен вирішити, чи блокувати себе до закінчення заданої чи операції ні. Якщо операція забирає значний час, як при друкуванні деякого блоку даних, то драйвер блокується доти, доки операція не закінчиться, і оброблювач переривання не розблокує його. Якщо команда вводу-виводу виконується швидко (наприклад, прокручування екрана), то драйвер очікує її завершення без блокування.

Незалежний від пристроїв шар операційної системи

Велика частина програмного забезпечення вводу-виводу є незалежною від пристроїв. Точна границя між драйверами і незалежними від пристроїв програмами визначається системою, тому що деякі функції, що могли б бути реалізовані у незалежний спосіб, в дійсності реалізуються у складі драйверів для підвищення ефективності чи з інших причин.

Типовими функціями для незалежного від пристроїв шару є забезпечення загального інтерфейсу до драйверів пристроїв, іменування пристроїв, захист пристроїв, забезпечення незалежного розміру блоку (тобто даний шар ПО не повинен безпосередньо залежати від розміру блоку в конкретному пристрої. Це досягається введенням т.зв. "логічного блоку"), буферизація, розподіл пам'яті на блок-орієнтованих пристроях, розподіл і звільнення виділених пристроїв, повідомлення про помилки.

Зупинимося на деяких функціях даного переліку. Верхнім шарам програмного забезпечення незручно працювати з блоками різної величини, тому даний шар забезпечує єдиний розмір блоку, наприклад, за рахунок об'єднання декількох різних блоків у єдиний логічний блок. У зв'язку з цим верхні рівні мають справу з абстрактними пристроями, що використовують єдиний розмір логічного блоку незалежно від розміру фізичного сектора.

Користувальницький шар програмного забезпечення

Хоча велика частина програмного забезпечення вводу-виводу знаходиться усередині ОС, деяка його частина міститься в бібліотеках, що використовуються програмами користувачів. Системні виклики, що містять виклики вводу-виводу, звичайно реалізуються бібліотечними процедурами.

Набір подібних процедур є частиною системи вводу-виводу. Зокрема, форматування вводу чи виводу виконується бібліотечними процедурами. Прикладом може служити функція `printf` мови С, що приймає рядок формату і, можливо, деякі перемінні як вхідну інформацію, потім будує рядок символів ASCII і робить системний виклик `write` для виводу цього рядка. Стандартна бібліотека вводу-виводу містить велике число процедур, що виконують ввід-вивід і працюють як частина користувацької програми.

Іншою категорією програмного забезпечення вводу-виводу є підсистема спулінга (spooling). Спулінг - це спосіб роботи з виділеними пристроями в мультипрограмній системі.

Розглянемо типовий пристрій, що вимагає спулінг – рядковий принтер. Хоча технічно легко дозволити кожному користувацькому процесу відкрити спеціальний файл, зв'язаний із принтером, такий спосіб небезпечний через те, що користувацький процес може монополізувати принтер на довільний час. Замість цього створюється спеціальний процес – монітор, що дістає виняткові права на використання цього пристрою. Також створюється спеціальний каталог, називаний каталогом спулінга. Для того, щоб надрукувати файл, користувальницький процес поміщує виведену інформацію в цей файл і поміщає його в каталог спулінга. Процес-монітор по черзі роздруковує усі файли, що містяться в каталозі спулінга.

2.22. Структури файлових систем.

Абстракція файлової системи.

У відповідності з викладеною вище загальною структурою системи вводу/виводу встановились і рівні абстракції у структурі файлової системи, показані на Рис. 2.42.

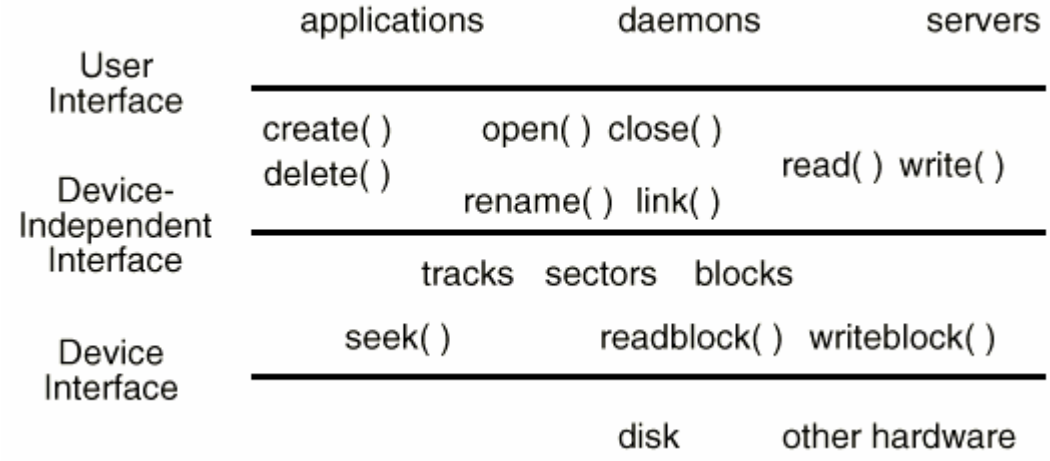


Рис. 2.42. Рівні абстракції у структурі файлової системи.

Цінність файлових систем.

Для користувача важливі такі якості файлових систем:

живучість (довге існування) - дані перебувають між навколишніми потужними коловоротами та аваріями;

зручність використання - дані можна зручно шукати, перевіряти, модифікувати, і т. п.;

ефективність – раціональне використання дискового простору;

швидкість – можна звернутись до даних швидко;

захист – інші користувачі не можуть зіпсувати (іноді навіть подивитись) мої дані.

ОС забезпечує:

файлову систему з каталогізацією та найменуванням - дозволяє користувачу вказувати директорії та імена замість місця знаходження даних на диску;

управління диском – збереження інформації (keeps track) про розміщення файлів на диску, швидкий доступ до цих файлів;

захист – неможливість неавторизованого доступу.

Інтерфейс користувача з файловою системою.

Файл є логічною одиницею запам’ятовування:

послідовності записів - a series of records (IBM mainframes);

послідовності байтів - a series of bytes (UNIX, most PCs).

Що запам’ятовується у файлі?

C++ source code, object files, executable files, shell scripts, …

Macintosh ОС явно підтримує типи файлів - TEXT, PICT, і т. п.

Windows використовує угоди про найменування файлів - “.exe” та “.com” для виконуваних (Executables), і т. п.

UNIX дивиться на зміст, щоб визначити тип:

Shell scripts — start with “#”

PostScript — starts with “%!PS-Adobe…”

Executables — starts with magic number (системного коду)

Операції з файлами.

Create(name):

створює дескриптор файлу на диску для репрезентації новосворюваного файлу;

додає елемент до директорії, щоб асоціювати ім’я з цим дескриптором файлу;

відводить простір на диску для файлу;

додає іформацію про розміщення на диску до дескриптора файлу.

fileId = **Open**(name, mode):

визначає унікальний ідентифікатор, що називається file ID (повертається до користувача);

встановлює mode (r, w, rw) для управління спільним доступом до файлу;

Close(fileId) – операція, протилежна до **Open.**

Delete(fileId) – операція, протилежна до **Create.**

Загальні моделі доступу до файлів.

Послідовний доступ (Sequential access).

Дані обробляються по порядку, байт за байтом, завжди просуваючись вперед. Більшість випадків доступу мають саме цю форму.

Приклад: компілятор читає вихідний (source) файл.

Прямий або випадковий доступ (Direct / random access). Кожний байт у файлі може бути доступним безпосередньо без звертання до яких – небудь його попередників. Приклад: accessing database record 12

Доступ по ключу (Keyed access).

Доступ до байта можливий на основі значення ключа. ОС не підтримує доступа по ключу. Програма користувача повинна визначити адресу ключа, після цього використати випадковий (random) доступ до файлу, що підтримується ОС.

Приклади: database search, dictionary

Операції з файлами (продовження).

Read(fileId, from, size, bufAddress):

Операція читання прямого доступу. Читання у буфер заданої кількості байт з файлу, починаючи з заданої позиції:

```
for (pos=from, i=0 ; i < size ; i++)
*bufAddress[i] = file[pos++];
```

Read(fileId, size, bufAddress):

Операція читання послідовного доступу. Читання у буфер заданої кількості байт з файлу, починаючи з поточної позиції:

```
for (pos=fp, i=0 ; i < size ; i++)
*bufAddress[i] = file[pos++];
fp += size;
```

Write — подібно до **Read.**

Каталогізація і наіменування.

Користувач і ОС повинні мати деякий спосіб звертання до файлів, що зберігаються на диску. ОС бажає використовувати номера (індекс у масиві дескрипторів файлів (ефективність і т. п.). Користувач бажає використовувати текстові імена (читабельність, мнемонічність і т. п.). ОС використовує директорії для збереження інформації про імена і відповідні індекси файлів.

Проста система іменування.

Один простір імен для цілого диску. Кожне ім'я повинно бути унікальним.

Імплементація: Запам'ятовується директорія на диску. Директорія містить пари <name, index>.

Використовувалась на ранніх mainfram-ах, ранніх Macintosh OS, та в MS DOS.

User-based система іменування.

Один простір імен для кожного user-а. Кожне ім'я у цій директорії користувача повинно бути унікальним, але два різні користувача можуть вживати однакове ім'я для файлу у своїх директоріях.

Multilevel система іменування.

Простір імен має структуру дерева. Імплементація:

директорії запам'ятовуються на диску, подібно до файлів;

кожна директорія містить пару <name, index> без певного порядку;

файл, розміщений у директорії, може являти собою іншу директорію;

система іменування має "/" для відокремлення рівнів.

Структура, що створюється у результаті, є деревом директорій. Використовується у UNIX.

Реалізація файлових систем

Апаратні засоби диску (Disk Hardware).

Механізм доступу (Arm) може переміщатись в середину і назовні. Головки читання/запису мають доступ до кільця даних за час, коли диск робить один оберт. Диск містить одну або більше пластин. Кожна пластина розподіляється на кільця даних, що називаються треками (tracks), і кожний трек ділиться на сектори. Інформація, розміщена на одній пластині, треку і секторі, називається блоком.

Тенденції у дисковій технології:

диски стають меншими за розміром, зменшується вага;

при збереженні ємності збільшується швидкість передачі даних;

диски зберігають дані з більшою плотністю

плотність покращується швидше ніж механічні обмеження (час пошуку - seek time, затримка обертання - rotational delay);

диски стають дешевшими.

Структури даних для файлів.

Кожний файл описується дескриптором файлу, який може містити (змінюється з ОС):

тип;

права доступу - read, write;

лічильник зв'язків - кількість каталогів, що містять даний файл;

власник, група, розмір;

час доступу - коли створено, останній доступ, остання модифікація;

блоки, де файл розміщено на диску.

Не включено:

ім’я файлу

Структури даних ОС для файлів.

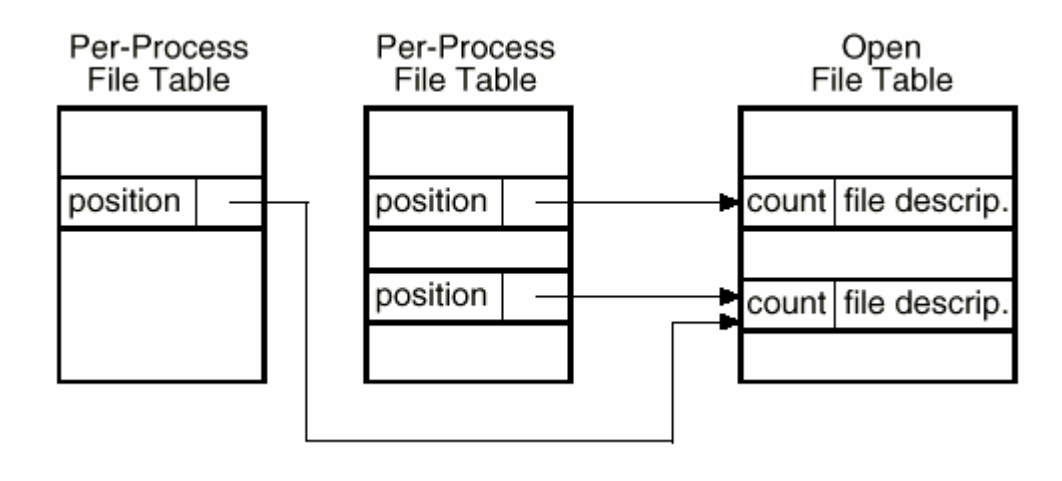


Рис.2.43. Таблиці файлів в ОС.

Таблиця відкритих файлів Рис.2.43 (одна, належить ОС) перелічує всі відкриті файли. Кожний вхід містить:

дескриптор файлу;

кількість відкривань - кількість процесів, що мають файл відкритим.

Таблиця файлів процесу (багато) перелічує всі відкриті файли для даного процесу. Кожний вхід містить:

покажчик (Pointer) на вхід у таблиці відкритих файлів;

поточну (offset) позицію у файлі.

Стуктури даних у UNIX для файлів.

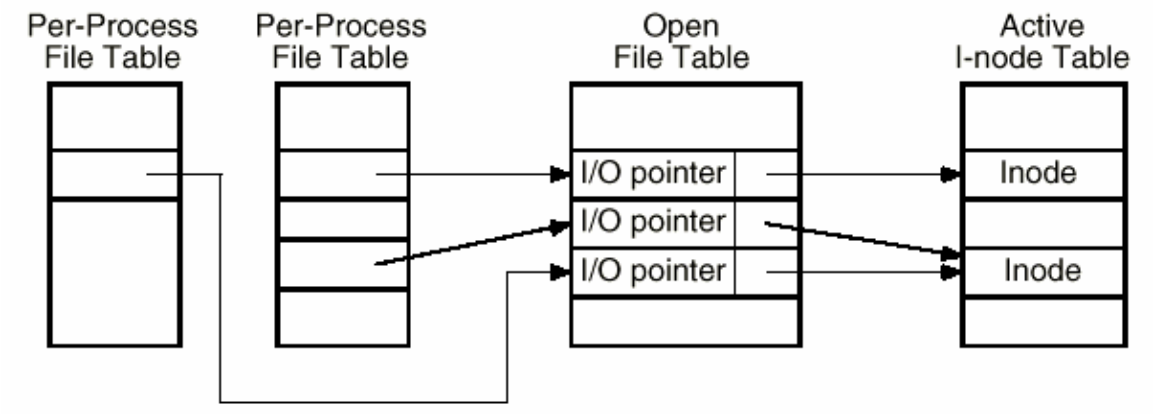


Рис.2.44. Таблиці файлів в ОС Unix.

Таблиця активних Inode - інформаційних вузлів Рис.2.44.(одна, належить ОС). Перелічує всі активні інформаційні вузли (file descriptors).

Таблиця відкритих файлів (одна, належить ОС)**; к**ожний вхід містить:

покажчик на вхід в таблицю активних інформаційних вузлів;

поточну (актуальну) позицію (зміщення) у файлі.

Таблиця файлів процесу (багато); **к**ожний вхід містить покажчик на вхід у таблицю відкритих файлів.

Структури даних на диску для файлів.

Інформація дескриптора файлу повинна бути збережена також і на диску, для довготривалого існування (for persistence). Вона включає всю базисну інформацію, перелічену вище.

Всі інформаційні вузли (file descriptors) запам'ятовуються у фіксованого розміру масиві на диску, що називається ilist.

Розмір масиву ilist визначається при ініціалізації диску. Індекс дескриптора файлу у цьому масиві називається номером інформаційного вузла – inumber.

Дескриптор файлу запам'ятовується спочатку на внутрішньому (або зовнішньому) треку, після цього на середньому треку (чому?).

У результаті маленькі дескриптори файлів розкидані по диску так, щоб покрити файли даних.

Приклади

Файлові системи у UNIX.

Дескриптор файлу (інформаційний вузол - inode) репрезентує файл. Всі Інформаційні вузли - inodes запам'ятовуються на диску у масиві фіксованого розміру, що називається ilist. Розмір масиву ilist визначається при ініціалізації диску. Індекс дескриптора файлу у цьому масиві називається номером інформаційного вузла – inumber.

Інформаційні вузли - inodes для активних файлів розміщуються також і у пам'яті, у таблиці активних вузлів (active inode table).

Диск UNIX-у може бути розділених на частини (partitions), кожна з яких містить блоки для запам'ятовування директорій і файлів, а також блоки для запам'ятовування масиву ilist.

Інформаційні вузли - Inodes корреспондуються з файлами, але є деякі спеціяльні інформаційні вузли:

– Boot block — code for booting the system;

– Super block — size of disk, number of free blocks, list of free blocks, size of ilist, number of free inodes in ilist, etc.

Високорівневий погляд (Рис.2.45.):

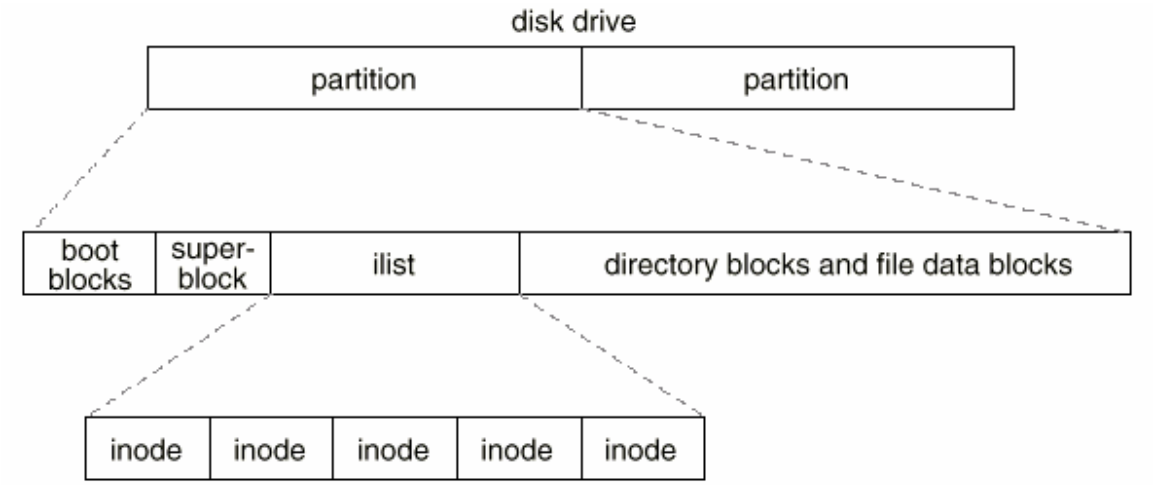


Рис.2.45 Файлові системи у UNIX. Високорівневий погляд

Низькорівневий погляд (Рис. 2.46):

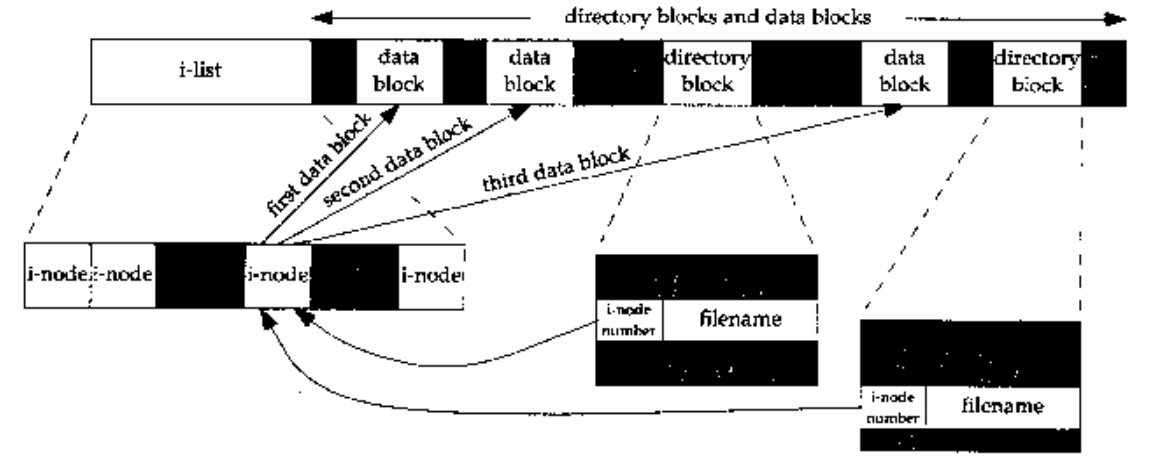


Diagram from Advanced Programming in the UNIX Environment, W. Richard Stevens, Addison Wesley, 1992.

Рис.2.46. Файлові системи у UNIX. Низькорівневий погляд

2.23. Розподілені файлові системи (Distributed File Systems).

Розподілена файлова система – розподілена реалізація файлової системи

Файл сервіс – специфікація інтерфейсу файлової системи, як її бачать користувачі

Файл сервер – процес, що діє на деякій машині, та допомагає впроваджувати файл сервіс

Цілі розподіленої файлової системи

Прозорість мережі

Передбачають однакові операції для доступу до віддалених та локальних файлів

В ідеалі, клієнт не має знати місцезнаходження файлів, щоб дістатися до них

Доступність / міцність – файл сервіс повинен підтримуватися навіть коли система частково пошкоджена

Продуктивність – потрібно подолати вузьке місце централізованої файлової системи

В принципі, файли у розподіленій файловій системі можуть зберігатися на будь-якій машині

Однак, типове розподілене середовище має кілька визначених машин, які називаються файл серверами та зберігають усі файли

Сервери розподіленої файлової системи – файл сервіс інтерфейс

Потрібні операції для створення та видалення, відкриття та закриття та читання та запису файлів

Модель запису / завантаження (upload / download)

Файл сервіс забезпечує:

Читати – передача цілого файлу до клієнта

Записати – передача цілого файлу до сервера

Клієнт працює з файлом локально (у пам’яті або на диску)

Просто, ефективно, якщо працювати над цілим файлом

Необхідно пересувати цілий файл

Потрібне місце на локальному диску

Модель віддаленого доступу

Файл сервіс забезпечує звичайні операції над файлами

Файл залишається на сервері

Розподілена структура імен (Distributed Naming Structure)

Потрібні операції для перетворення імен, підтримки багаторівневих директорій та зв’язків

Прозорість розміщення – ім’я файлу не вказує на фізичне розміщення

Правда для багатьох схем іменування

Незалежність розміщення – ім’я файлу не повинно змінюватися, якщо змінюється його місце розташування

Неправда для багатьох схем іменування

Абсолютні імена:

Імена форми: машина : шлях

Використовувалися у:

Старих розподілених системах UNIX

Поточних веб броузерах (наприклад, Netscape)

Користувач може використовувати однакові засоби та операції для локального та віддаленого доступу

Немає прозорого та незалежного розміщення

Перетворювати віддалені директорії у локальні директорії (можливо за вимогою)

Підтримана клієнтом інформація про перетворення:

Використовувалася UNIX та NFS – мережна файлова система Sun

Клієнт підтримує:

Множину локальних імен для віддалених розміщень

Таблиця перетворення (/etc/fstab), що визначає:
<ім’я віддаленої машини : шлях>
та <локальний шлях>

При завантаженні, локальне ім’я пов’язане з віддаленим іменем

Інформація про перетворення, що підтримується сервером:

Якщо файли переміщаються на інший сервер, інформація про перетворення потрібно лише оновлювати на серверах

Однаковий простір імен для віддалених та локальних директорій

Імена форми: /…/машина/fs/шлях

Використовується у:

OSF розподіленому обчислювальному середовищі

Імена файлів завжди однакові, незважаючи на те чи локальні, чи віддалені ці імена

Тоді як клієнт отримує доступ до файлу, сервер посилає копію до робочої станції клієнта

Розміщення незалежне, але не прозоре

Мережна файлова система Sun (Sun’s Network File System)

Розроблена фірмою Sun Microsystems

Перший розподілений файл сервіс розроблено як проект, представлений у 1985

Щоб впровадити його прийняття як стандарт

Визначення ключових інтерфейсів були поміщені у загальнодоступний домен у 1989

Код був зроблений доступним до інший ліцензованих комп’ютерних виробників

Зараз фактичний стандарт для LANs

Забезпечує прозорий доступ до віддалених файлів на LAN, для клієнтів на UNIX та інших операційних систем

UNIX комп’ютери типово мають NFS клієнт та серверний модуль у своєму ядрі ОС

Доступний для майже будь-якого UNIX та MACH

Модулі клієнтів доступні для Макінтошів та ПК

Перетворення віддалених файлових систем

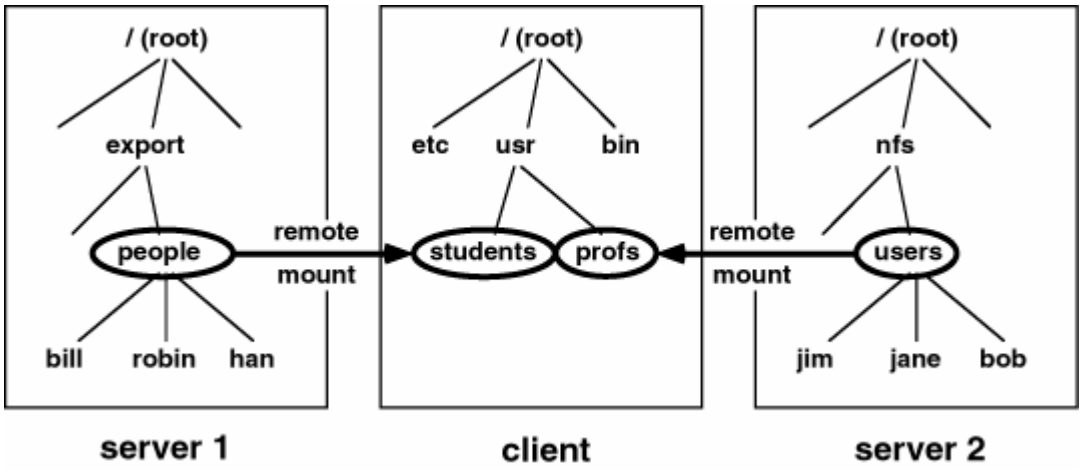


Рис.2.48. Перетворення віддалених файлових систем в NFS.

NFS підтримує перетворення віддалених файлових систем машинами-клієнтами (Рис.2.48)

Простір імен, що бачить кожен клієнт, може бути різним

Один файл на сервері може мати різні імена для різних клієнтів

NFS не примушує до використання одного на всю мережу простору імен, але універсальний простір імен (та прозорість розміщення) може бути оснований, якщо бажано

На кожному сервері

Є файл (звичайно /etc/exports), що містить імена локальний файлових систем, які доступні для віддаленого перетворення

Список доступу асоціюється з кожним іменем та показує яким хостам дозволено перетворювати цю файлову систему

На кожному клієнті

Модифікована версія команда перетворення UNIX перетворює віддалену файлову систему

Базоване на RPC – визначає ім’я віддаленого хоста, шлях до директорії у віддаленій файловій системі та локальне ім’я, коду він має бути перетворений

Запити на перетворення звичайно представлені, коли система ініціалізується

Користувач може також перетворювати інші віддалені файлові системи

2.24. Архітектура програмного забезпечення NFS

Віртуальна файлова система:

відділяє характерні системні файлові операції від їх реалізації (може мати різні типи локальних файлових систем);

базується на дескрипторі файлу, що називається vnode та є унікальним у всій мережі.

Протокол NFS

NFS протокол впроваджує множину віддалених викликів процедур для віддалених файлових операцій

Шукає файл у директорії

Керує зв'язками та директоріями

Створення, перейменування та видалення файлів

Отримання та встановлення атрибутів файлів

Читання файлів та запис у файли

NFS протокол не змінює свій стан

Сервери не підтримують інформацію про своїх клієнтів від одного доступу до наступного

На сервері немає відкритих файлових таблиць

Немає операцій відкриття та закриття

Кожен запит повинен впроваджувати унікальний ідентифікатор файлу та зміщення (offset) у файлі

Легко вийти з аварійної ситуації, але файлові операції мають бути ідемпотентними

Через те, що NFS не змінює свій стан, всі модифіковані дані повинні записуватися на диск серверу до того, як результати повернуться до клієнта

Аварійні ситуації на сервері та вихід з них мають бути невидимими клієнту – дані мають бути непошкодженими

Втрачаються переваги кешування

Рішення – диски RAM з резервними батареями, що періодично записуються на диск

Гарантовано, що одиночний запис NFS буде атомарним, та не змішується з іншими записами до того ж файлу

Однак, NFS не забезпечує керування паралелізмом

Системний виклик запису (write system call) може бути розкладеним у кілька записів NFS, які можуть бути вкладеними

Так як NFS не змінює стан, це не вважається проблемою NFS

Кешування у NFS

Традиційний UNIX

Кешуються частини файлів, директорії та атрибути файлів

Використовуються випереджаюче читання (prefetching) та затриманий запис

Сервери NFS

Так же, як в UNIX, за виключенням того, що серверні операції запису використовують наскрізний запис (write-through)

Інакше, поломки серверу можуть виявитися у невиявлених втратах даних клієнтами

Клієнти NFS

Кешуються результати читання, запису, пошуку та операції читання директорій.

Можливі проблеми несумісності: записи одного клієнта не спричиняють негайне оновлення кешів інших клієнтів

Читання файлу

Коли клієнт записує у кеш один або більше блоків файлу, він також записує часову помітку, що вказує час, коли файл було востаннє модифіковано на сервері

Будь-коли, коли відкривається файл, та відбувається контакт з сервером, щоб вибрати новий блок з файлу, відбувається перевірка достовірності

Клієнт посилає запит на сервер про останню модифікацію та порівнює цей час зі своєю часовою поміткою з кешу

Якщо час модифікації більш пізнє, всі записані у кеш блоки з цього файлу визнаються недійсними

Блоки вважаються вірними наступні 3 секунди (30 секунд для директорій)

Записи файлів

Коли кешована сторінка модифікується, вона помічається як зіпсована, та очищується (flushes), коли файл закривається.

Частина 3.
Системне та мережеве адміністрування.

3.1. Розвиток засобів системного та мережевого адміністрування.

Функціональні області управління, що відносяться до цієї сфери, чітко визначені [30]:

розв'язання проблемних ситуацій – діагностика, локалізація та ліквідація несправностей, регістрація помилок, тестування;

керування ресурсами – облік, контроль використання ресурсів, обмеження доступу до них;

керування кофігурацією, що спрямоване на забезпечення надійного і ефективного функціонування всіх компонент інформаційної системи;

контроль продуктивності – збів та аналіз інформації про роботу окремих ресурсів, прогнозування ступеню забезпечення потреб користувачів/додатків;

захист даних – керування доступом користувачів до ресурсів, забезпечення цілістності даних та керування їх шифруванням.

На даний час існують сотні продуктів, що дозволяють розв'язувати ті або інші задачі системного адміністрування. В основі своєрідної піраміди знаходяться базові платформи управління - CA-Unicenter TNG компанії Computer Associates, HP OpenView та Tivoli Enterprise виробництва IBM. Названі системи покривають всі перелічені вище функціональні області і внаслідок тривалого розвитку стали доволі схожим одна на одну. Так, OpenView розроблялась спочатку як система мережного адміністрування, а засоби системного адміністрування були додані пізніше; напроти, Tivoli Enterprise спочатку була націлена на системне адміністрування, а засоби мережного адміністрування додані пізніше.

Що стосується архітектурної реалізації, то тут можна спостерігати як варіанти єдиних інтегрованих систем, так і модульних структур (OpenView).

Аналіз розвитку базових концепцій системного та мережного адміністрування приводить до висновку, що комплексний підхід став домінуючим, системне та мережеве адміністрування перестало сприйматись як два антогоністичні світи, а це розчистило дорогу для інтеграційних процесів. Одна з сучасних задач полягає у максимально можливому використанні сучасних Web – технологій, від екранних інтерфейсів до застосування активних Java-компонентів і мови XML. Самостійну цінність являють собою

засоби керування Web-ресурсами, причому не тільки стосовно мереж intranet та extranet, але ще у більшій мірі у зв’язку з розвитком електронної комерції.

Система моніторингу Інтернет сервісів.

Моніторинг сервісів Інтернет

Для компаній, що мають сайти в Інтернеті, постачальників послуг Інтернет і компаній, що здійснюють свою діяльність через Інтернет, контроль за працездатністю мережі в тому вигляді, як це представляється відвідувачам, а також за непомітними для них компонентами, став найважливішою умовою успішного бізнесу. Недоступність або низька продуктивність Интернет-сайтів негативно позначається на прибутковості, підриває репутацію торгової марки і спричиняє відтік клієнтури. Підприємства очікують від своїх постачальників послуг та внутрішніх відділів IT надання чітких гарантій якості послуг у в плані доступності Интернет-сайта, часу відгуку, а також своєчасного повідомлення про випадки перебоїв чи уповільнення роботи.

У даному випадку корисними можуть виявитися сервери кешування, вбудовані системи резервування і програми розподілу навантаження. Але що робити при виникненні проблем в інфраструктурі IT? Які мережні додатки зачіпаються в цьому випадку? Де знаходиться проблемний елемент? У сервері доменних імен? В одній з магістральних ліній зв'язку глобальної мережі? У бездротовому сервері? У сервері web-додатків? А може вся справа в прикладних програмах електронної комерції? У випадку виникнення проблемних ситуацій, персоналу технічного обслуговування web-систем і мережних систем необхідна можливість оперативно локалізувати й усунути несправності, а відділу по роботі з клієнтами варто вчасно повідомляти про такі проблеми всіх клієнтів, щоб зберегти їхню довіру і лояльність.

Система моніторингу Інтернет сервісів дає комплексне представлення про інфраструктуру Web. Призначення цього продукту – допомогати працівникам відділу IT в ефективному прогнозуванні, локалізації, діагностуванні і виявленні проблемних елементів, попередженні випадків недостатньої пропускної здатності, а також керуванні і складанні звітів про рівні обслуговування. Вона має характеристики, що роблять її привабливою для компаній, які мають сайти в Інтернеті, телекомунікаційних компаній і операторів зв'язку, що забезпечують доступ в Інтернет, постачальників послуг Інтернет, котрі надають послуги хостингу і спільного розміщення в Web, а також постачальників послуг додатків, що пропонують хостинг для прикладних програм і систем електронної комерції. Серед цих

характеристик:

- Контроль за складними транзакціями в області електронного бізнесу;
- Всебічна підтримка стандартних Інтернет-протоколів, серверів і додатків;
- Надання клієнтам безпечного доступу до власної інформації, а постачальнику послуг Інтернет – можливості розбиття даних по клієнтах чи сферам діяльності.

У нових умовах Інтернет-економіки успіху досягають лише ті компанії, що готові до постійних змін. Технології керування повинні бути гнучкими й автоматично адаптуватися до інфраструктури, що доволі часто змінюється. Ключова функція системи моніторингу Інтернет сервісів - моментальний збір інформації. Відразу після автоматичного запуску програми установки, "майстер" конфигурирования крок за кроком допомагає адміністратору сконфігурувати цільові об’єкти, служби і зонди. Встановлені за замовчуванням значення значно полегшують процедуру для тих, хто займається цим вперше, а для зміни граничних значень сигналів тривоги і рівнів обслуговування можуть використовуватися прості регулятори. Для установки системи моніторингу Інтернет сервісів потрібно всього кілька хвилин, після чого програма відразу приступає до моніторингу усієї інфраструктури Web.

Моделювання і моніторинг діяльності

Система моніторингу Інтернет сервісів поєднує в собі два методи оцінки ефективності функціонування Интернет-сайта: "активний" моніторинг (синтетичні транзакции) і "пасивний" моніторинг (моніторинг на рівні сервера). У результаті з'являється система показників, що безпосередньо відображає операції ваших клієнтів, а також забезпечується оперативне повідомлення про виникнення проблем на сервері. Додайте до цього можливість контролю фактичної працездатності сервера в тому вигляді, як це представляється кінцевому користувачу з Web-браузером, - і Ви забезпечені повною інформацією про час відгуку системи для кінцевих користувачів з розбивкою по таким важливим складовим, як сервери обробки даних, "хмара" Internet, "остання миля" і робоче місце клієнта.

Система моніторингу Інтернет сервісів здійснює активний мониторинг комплексних процесів "компанія-компанія", таких як логістика, постачання, структура постачань і виконання, а також транзакцій "компанія-клієнт", таких як страхові вимоги, операції з акціями і покупки з використанням "кошика". Ви можете відслідковувати нові мережні додатки відразу ж, як тільки вони

стають доступними в режимі он-лайн. Інтуїтивно-зрозумілий "майстер" дозволяє адміністратору відслідковувати кожен етап типової транзакции кінцевого користувача, такий як вхід користувача в систему, пошук по каталогу, перегляд "кошика", а також здійснює автоматичний запис всіх URL адрес і всіх даних, що вводяться. Потім на регулярній основі відтворюється записана транзакцію, моделюються типові дії кінцевого користувача, і здійснюється збір найважливіших даних про готовність і час відгуку. Для забезпечення оптимальної гнучкості передбачена інтелектуальна обробка динамічного інформаційного вмісту, зокрема, ідентифікаторів сеансу, які генеруються автоматично, даних про попередні звернення (cookie), що генеруються сервером URL, активних серверних сторінок (Active Server Pages), скриптов CGI і серверних скриптів JavaScript. Крім того, система моніторингу Інтернет сервісів допомагає в локалізації неполадок, здійснюючи активний моніторинг дискретних Інтернет-послуг і протоколів за допомогою моделювання запитів користувачів до базових служб, таких як дозвіл імен і доступ до каталогів, поштові служби для відправлення й одержання електронної пошти, такі web-послуги, як протокол захищеної передачі гіпертекстових повідомлень і протокол передачі файлів, а також послуги віддаленого доступу до мережі.

Збір даних здійснюється за допомогою активного моніторингу за допомогою програмних зондів, встановлених у місцях присутності постачальника послуг Інтернет, у віддалених офісах компанії або на сайті партнера за межами міжмережевого екрана (firewall). Використовуючи протокол HTTP, зазначені програмні зонди забезпечують захищену передачу даних про продуктивність у централізовану реляційну базу даних Microsoft Access, Microsoft QL-Server чи Oracle. Потім база даних обновляє інструментальну панель з індикаторами рівня обслуговування, готовності і часу відгуку в режимі, близькому до реального часу, і генерує докладні web-звіти. Аварійні сигнали можуть пересилатися з використанням механізму гарантованої доставки повідомлень, а також з використанням протоколу NMP у будь-яку універсальну програму керування SNMP.

Моніторинг критично важливих серверів

Як доповнення до активного моніторингу зондів, а також для забезпечення оперативного й обґрунтованого повідомлення про наявність неполадок на критично важливому сервері в системі моніторингу Інтернет сервісів передбачені також різні стратегії (політики) пасивного моніторингу. За допомогою цих стратегій (політик) забезпечується відстеження файлів журналу і процесів ведучих web-серверів і міжмережевих екранів.

Керування на основі Service Level Agreement (SLA)

Модель даних для системи моніторингу Інтернет сервісів дозволяє використання угод про рівень обслуговування (SLA) по клієнтам чи сферам діяльності. Модель дозволяє вибирати в режимі, близькому до реального часу, різні види та звіти по певним клієнтах чи комбінаціям клієнт/послуга. Різні порушення обслуговування можуть піддаватися різнобічному аналізу з метою визначення клієнтів, на яких будуть впливати збої на сервері, і локалізації компонента, що є причиною порушення часу відгуку.

Звіти по обслуговуванню генеруються щодня і доступні для перегляду в Web. Зазначені звіти містять докладне розбиття по транзакциям і візуальне порівняння фактичного рівня активності і встановленого "базового" рівня для даної послуги. У системі моніторингу Інтернет сервісів використовується динамічне визначення граничних значень (щодо базового рівня) у залежності від часу. При застосуванні базового значення показники повинні не тільки перевищувати встановлений поріг, але й відповідати "нормальному" рівню активності для даної пори дня. Така технологія генерування сигналів тривоги допомагає запобігти "сплескам" сигналів тривоги і спрощує процедуру установки граничних значень.

Масштабованість

Архітектура системи моніторингу Інтернет сервісів дозволяє використовувати цей продукт як для моніторингу невеликої кількості серверів, так і для моніторингу величезного серверного господарства, що включає в себе тисячі серверів. Управлінська інформація з розподілених програмних зондів на високій швидкості передається на сервер оцінки продуктивності. Дані буферируются в синхронізованому інформаційному кеші й узагальнюються, що забезпечує відновлення інструментальної панелі і генерацію звітів з високою продуктивністю. Додатково удосконалити систему можна за рахунок збереження управлінських даних у базі даних SQL Server чи Oracle, орієнтованих на високу пропускну здатність.

Моніторинг від клієнта до магістралі мережі

У світі електронних послуг границі компанії стираються. Рішення системного керування повинні, минаючи firewall, добиратися до провайдерів послуг і навіть до web-браузера клієнта, забезпечуючи точну оцінку фактичної активності клієнта і виявлення дефектних ланок у ланцюзі обслуговування. Система моніторингу Інтернет сервісів дає повне представлення про функціонування внутрішньої web-інфраструктури компанії. Локалізуються проблеми, пов'язані із середовищем системи клієнта чи постачальником послуг Інтернет.

3.2. Система керування вузлами мережі.

Керування вузлами мережі.

Проактивное керування – це усунення причини потенційної несправності до того, як збій реально відбудеться. Одна лише реакція на зовнішні впливи не дозволить забезпечити цілодобову доступність мережі. Одночасно з цим необхідно безупинно керувати мінливими конфгураціями мережі, підтримувати швидкий ріст мережі, поєднувати різні мережні середовища. Система керування вузлами мережі вирішує ці проблеми, володіючи набором засобів мережного керування, дозволяючи здійснювати випереджальне (проактивне) керування мережним обчислювальним середовищем і підтримувати її інфраструктуру, що постійно розширюється. Система керування вузлами мережі є основною частиною рішень на в області інформаційного бізнесу (i-business) і керування інтегрованими послугами (Integrated Service Management).

Можливості системи керування вузлами мережі.

Система керування вузлами мережі є комплексною, інтелектуальною і простою у застосуванні платформою мережного керування. Він може застосовуватися як без попереднього налаштування, так і легко налаштовуватись під конкретні специфічні потреби компанії. Система керування вузлами мережі володіє наступними можливостями:

- Автоматично виявляє мережні пристрої і надає інформацію про обчислювальне середовище;
- Створює графічні карти і підкарти мережі, що дозволяють візуально відображати мережні об'єкти і події, що відбуваються в мережі;
- Здійснює постійний моніторинг мережі, відслідковуючи всі події, що відбуваються в мережі, який допомагає швидко визначати основну причину неполадок;
- Має вбудовані засоби усунення збоїв, що дає можливість швидко вирішувати мережні проблеми;
- Збирає ключову інформацію про мережу, що допомагає локалізувати проблеми і здійснювати випереджальне (проактивне) керування;
- Надає готові до застосування звіти, що дозволяють здійснювати планування росту мережі;
- Допускає віддалений доступ через Веб, що дозволяє обслуговуючому персоналу мережі, адміністраторам і замовникам здійснювати доступ з будь-якого місця;
- Розподілена архітектура системи дозволяє керувати великими

розподіленими мережами.

Ці й інші можливості системи керування вузлами мережі дозволяють здійснювати точний моніторинг мережі і швидко виявляти і вирішувати мережні проблеми перш, ніж вони перейдуть у критичну стадію. Крім того, Система керування вузлами мережі володіє вбудованими інтелектуальними засобами збору інформації про мережу, що може застосовуватися для створення звітів і планування модернізації мережі.

Наочне представлення мережі

Система керування вузлами мережі автоматично виявляє і заносить на карту всі пристрої, що підтримують протоколи TCP/IP, IPX, і пристрої рівня 2 (Bridge, Repeater, HUB, Switch), які наявні в мережі. На карті мережі графічно відображаються всі мережні об'єкти і топологія мережі. Причому кожен об'єкт представлений кольоровим графічним символом, що відображає його статус працездатності, на підставі якого можна визначити, де відбулися збої, перш ніж вони стануть критичними.

Расоби керування комутаторами в системі керування вузлами мережі забезпечують адміністраторів мережі докладною інформацією про обчислювальні середовища, що комутуються. Ось ці засоби:

- Відображення картини з'єднань, що комутуються, включаючи магістральні з'єднання і мережі складних топологій;
- Надання інформації про VLAN для кожного комутатора;
- Непідключений порт не розпізнається; тому відображається справжній стан комутатора;
- Відображення номерів портів, що дозволяє довідатися, який пристрій до якого порту комутатора під’єднано.

Керування мережами – від малих до великих

На основі системи керування вузлами мережі може бути створена розподілена система керування великими мережами. Станції збору даних з операційними системами Windows чи UNIX розподіляються в обчислювальному середовищі, реєструють дані локально і передають найбільш важливу інформацію на одну чи більше станцій керування.

За допомогою віддалених консолей, підключених до станцій керування і станцій збору даних, доступ до Windows чи UNIX може бути наданий декільком операторам. Веб-интерфейс користувача збільшує число операторів і дозволяє підключатися через зв'язки глобальних мереж. Система керування вузлами мережі дозволяє замовнику обмежитися купівлею тільки тих елементів, які йому

необхідні, і розширювати систему керування в міру росту мережі.

Обробка подій

Події пересилаються в оглядач подій, що входить до складу системи керування вузлами мережі, доступ до якого здійснюється зі станцій керування чи станцій збору даних, через веб-интерфейс користувача на основі мови Java. Тут наявний простий спосіб налаштування подій, які потрібно відображати; можна також відфільтровати ті події, які менш важливі для роботи.

При відмові в роботі одного пристрою можуть бути згенеровані сотні подій. Замість того, щоб приголомшувати оператора безліччю подій, які йому треба відстежити, служба кореляції подій, що входить до складу системи керування вузлами мережі, сортує події і подає операторам один сигнал тривоги на верхньому рівні. Іншими словами, консолідує всі події із загальною кореневою причиною і видає на консоль одну результуючу подію. До складу системи входять чотири готові кореляційні схеми. Вони охоплюють більшість розповсюджених подій, зв'язки між якими доводиться встановлювати користувачам:

- Відмова у роботі з'єднувача (Connector Down). Зв'язує недоступність пристроїв, що лежать нижче по потоку інформації, з відмовою пристрою, розташованого вище по потоку;
- Планове технічне обслуговування (Scheduled Maintenance). “Придушує” події, що відбуваються при плановому технічному обслуговуванні;
- Події, що повторюються (Repeated Event). Мінімізує дублювання тривог у заданих часових межах, наприклад відмови ідентифікації протоколу SNMP;
- Двоточкові події (Pair-Wise Events). “Придушує” події, що з'являються парами, наприклад відмова і наступне відновлення роботи вузла (node-up/node-down).

Проактивне керування за допомогою генерації звітів і сховища даних

Готові до застосування звіти дозволяють проводити проактивний аналіз тенденцій стану мережі. Система керування вузлами мережі створює звіти про швидкодію мережі, її доступність і т. д. Дані звіти дозволяють виявити вузькі місця в роботі мережі і запобігти потенційній несправності до її виникнення. Аналіз цих архівних даних надає чітку картину стану мережних пристроїв і дозволяє адміністраторам мережі здійснювати різні заходи, перш, ніж у мережі з'являться неполадки. Крім того, дані системи керування вузлами мережі про топологію, події і дані, зібрані по

протоколу SNMP, експортуються у спеціальне сховище даних системи. Потім дані піддаються інтелектуальній обрабці і сортуванню. До складу сховища даних входить відкрита схема, що забезпечує доступ для засобів складання звіту й інструментів аналізу, а також експорт в інші бази даних..

Прискорений інтелектуальний збір даних

Інформація про роботу мережі забезпечує операторів мережі даними для ефективного усунення неполадок і планування з урахуванням майбутніх вимог. Збирач даних, що входить до складу системи керування вузлами мережі, швидко збирає багато типів даних. Адміністратор має у своєму розпорядженні спосіб налаштування типів даних, що збираються, і частоти збору. Потім дані відображаються в графічному виді, що дозволяє визначити продуктивність мережі і планувати її зміни в майбутньому. Нова функція автоматичної установки базової лінії дозволяє автоматично запускати збір даних і встановлювати пороги, ґрунтуючись на відхиленні даних від норми. Якщо пороги перевищені, генерується подія, що повідомляє про появу неполадки.

Диспетчер системи керування вузлами мережі збирає інформацію у виді нечислових строкових змінних і генерує тривоги при наявності змін у мікропрограмному забезпеченні, стані зв'язків чи інших змінних. Наприклад, збираються дані про версію системи IOS у кожному комутаторі мережі і при їхній зміні генерується тривога.

Генерація звітів для ключових даних

Система керування вузлами мережі надає різноманітні звіти, що дозволяють адміністраторам мережі, а також внутрішнім і зовнішнім клієнтам бачити повну картину стану мережі. Ці звіти застосовуються для демонстрації того, що угоди про рівень обслуговування виконані. До складу продукту включені такі звіти, як: продуктивність, включаючи звіти про ступінь завантаження, про пристрої, які більше всього працюють на передачу та прийом, а також помилки інтерфейсу для вхідної і вихідної інформації.

У число звітів по продуктивності входять наступні звіти:

- Час відгуку на запит (Ping Response Time) і повтор запиту (Ping Retry). Показує час відгуку на запит і число повторних спроб визначення затримки всередині мережі та ідентифікує пристрої, що знаходяться на грані відмови в роботі;
- Ступінь завантаження сегмента RMON (RMON Segment Utilization) по відтікам. Показує відсоток використовуваної пропускної здатності мережі. Цей звіт допомагає в усуненні

вузьких місць і проблем маршрутизації мережі;

- Ретрансляція кадрів (Frame Relay). Відслідковує швидкості утворення заторів при прямій і зворотній передачі та виявляє вузькі місця;
- Доступність, включаючи зведені і докладні звіти по доступність пристроїв у відсотках;
- Інвентаризація устаткування (Inventory), включаючи зведені і докладні звіти по устаткуванню. Це дозволяє адміністраторам мережі стежити за її ростом;
- Виняткові події (Exeptions), у тому числі звіти про число і ступінь критичності випадків перевищення порогів.

Ці звіти дозволяють адміністраторам негайно побачити неполадки в мережі.

Цілодобовий доступ до мережі з будь-якого місця

Підтримка цілодобової роботи мережі в життєво важливою для підприємства. Система керування вузлами мережі надає можливості для вирішення цієї задачі. Вона дозволяє за розкладом виконувати архівування критично важливої інформації, пов'язаної з керуванням мережею, продовжуючи при цьому моніторинг і керування критичними подіями в мережі. Крім того, станції збору даних системи можуть налаштовуватися на відновлення після збою на станціях керування системи керування вузлами мережі і забезпечувати безупинний моніторинг мережі. У випадку збою на станції збору даних моніторинг мережі не переривається. Система керування вузлами мережі навіть веде моніторинг самої себе, щоб гарантувати, що вона працює правильно; у такий спосіб можна упевнитися, що мережа доступна і працює.

Система керування вузлами мережі забезпечує веб-доступ з будь-якого місця до засобів керування мережею. Надається доступ до карт мережі, даних і подій. Це дозволяє операторам і адміністраторам керувати мережею з будь-якого місця, включаючи віддалений офіс, будинок чи номер в готелі, і підтримувати доступність і високу швидкодію мережі. Карти і події обновлюються динамічно, без будь-якого втручання оператора. Топологія мережі представляється в графічному чи табличному форматі. Крім того, система дозволяє, з метою діагностики мережі і планування, робити запити про дані з мережі по протоколу SNMP (наприклад, інформацію про трафік через інтерфейс, завантаження центрального процесора або маршрутизації трафика).

Реєстрація з ідентифікацією за допомогою пароля при вході забезпечує безпеку даних керування. Веб-інтерфейс користувача дозволяє визначати ролі користувача і здійснювати фільтрацію інформації, спираючись на те, за яку частину мережі відповідає

даний користувач. Компактний веб-інтерфейс користувача дозволяє системі підтримувати роботу великої кількості операторів мережі.

3.3. Система керування середовищем широкомовної передачі.

Моніторинг і керування трафіком широкомовної передачі.

Можливість широкомовної (багатоадресної) передачі даних, звичайно, вбудована в інфраструктуру IP-мережі. Широкомовна передача має на увазі передачу даних з одного чи декількох джерел у кілька пунктів призначення. Вона ідеально підходить для додатків, у яких один великий набір даних необхідно раціонально передати у багато пунктів призначення одночасно.

Приклади додатків, у яких може використовуватися широкомовна передача:

- Доставка відео- та аудіоінформації через мережу;
- Дистанційне навчання, Web-мовлення чи відеоконференції;
- Одночасна доставка в багато пунктів призначення інформації про товарні запаси, ціни та біржову інформацію.

Особливість цих типів додатків полягає в необхідності мати достатню пропускну здатність або визначати пріоритети, щоб уникнути втрат чи порушення порядку проходження пакетів. Для цього потрібен моніторинг і керування трафіком широкомовної (багатоадресної) передачі.

Реалізація і керування середовищем широкомовної передачі

Маршрутизатори мережі виконують велику частину роботи, зв'язаної із широкомовною передачею. Наприклад, усі маршрутизатори Cisco, що працюють під керуванням системи IOS версії не нижче 12.0, підтримують широкомовну передачу. Реалізація широкомовної передачі по протоколі IP у мережі зводиться в основному до застосування вже наявних засобів.

Однак широкомовна передача може мати серйозний вплив на продуктивність мережі, а сама широкомовна передача часто буває чуттєва до затримок (latency).

Отже, важливо мати інструментальні засоби для ефективного моніторингу і керування сеансами широкомовної передачі. Система керування середовищем широкомовної передачі розробляється спеціально для цієї мети.

Функції системи керування середовищем широкомовної передачі.

Система керування середовищем широкомовної передачі дозволяє оператору переглядати топологію і стан середовища

широкомовної передачі. Оператор може одержувати попередження про зміни стану, наприклад, про збій маршрутизатора, і швидко приступати до усунення збою. Це інструментальний засіб також вимірює інтенсивність (flow rate) широкомовного трафіка по всій мережі.

Система керування середовищем широкомовної передачі має наступні можливості:

Автоматичне виявлення зв'язків у топології маршрутизації широкомовної передачі

Система керування середовищем широкомовної передачі виявляє маршрутизатори, що підтримують широкомовну передачу, і зв'язку піринга (peering, рівноправний обмін інформацією) між ними. Він застосовує такі функціональні можливості системи, як виявлення, графічне відображення (mapping), колірне кодування стану об'єктів і виявлення з обмеженнями за допомогою списку включень чи списку виключень.

Інтерактивне відображення топології багатоадресної передачі

Графічна карта системи керування середовищем широкомовної передачі показує прямі двоточкові (point-to-point) канали зв'язку (глобальна мережа) і канали зв'язку з мультидоступом (multi-access links) (наприклад, підмережа локальної мережі) між маршрутизаторами, що підтримують широкомовну передачу.

Можливий перегляд по запиту логічних зв'язків піринга з сусідами для широкомовних маршрутизаторів у підмережі, а також зв'язків піринга для всіх інтерфейсів одного маршрутизатора. Існує можливість накласти дерево розсилання на карту топології, при цьому будуть показані джерела і напрямки багатоадресної передачі по протоколу IP. Відображення дерева розсилання полегшує локалізацію збоїв.

Членство в групах може відображатися на картах за допомогою унікального колірного кодування. Членство в групах може також відображатися в табличній формі. По запиту на карті також можуть бути виділені відведені для широкомовної передачі маршрутизатори у всіх підмережах на карті чи в окремій підмережі.

Вимірювання інтенсивності широкомовного трафіка

Є можливість збору і моніторингу інформації про інтенсивність вхідного і вихідного трафіка через інтерфейси маршрутизаторів. Дані про продуктивність можуть проглядатися в табличному чи графічному виді. Частота опитування для збору інформації про поточну інтенсивність трафика для всіх груп широкомовної передачі може встановлюватися незалежно для кожного маршрутизатора.

По запиту може бути відображений у табличній формі "знімок" інтенсивності трафика для всіх груп, що відомий одному маршрутизатору. Може бути також відображена діаграма трафіка для однієї групи, побудована на основі інформації від одного чи більше маршрутизаторів. Для кожної групи чи маршрутизатора може проводитися збір, представлення й автоматичне відновлення статистичних даних. При відображенні дерева розсилання можна побудувати діаграму інтенсивності трафіка для всіх маршрутизаторів цього дерева. Ця інформація корисна для планування розширення реалізації багатоадресної передачі даних.

При будь-якій зміні стану маршрутизатора, зв'язків піринга із сусідніми пристроями або при перевищенні заданих користувачем граничних значень трафіка можуть генеруватися аварійні сигнали SNMP.

3.4. Система керування розподіленим обчислювальним середовищем.

Основні функції системи.

Система керування розподіленим обчислювальним середовищем забезпечує підхід до керування обчислювальним середовищем електронних служб підприємства з погляду бізнесу. Дане рішення являє собою управлінське середовище, котре здійснює моніторинг, контроль і складання звітів про стан IT ресурсів та IT служб компанії, збільшуючи, таким чином, час безвідмовної роботи всіх шарів, що складають обчислювальне середовище сучасного електронного підприємства: мережі, системи, бази даних, додатки, сервіси та Інтернет.

Програмне забезпечення системи керування розподіленим обчислювальним середовищем надає централізовану єдину "технологічну консоль", що дозволяє строго та ефективно контролювати події, що відбуваються у всіх системах, створюючи свого роду "центр керування польотом" для всього розподіленого обчислювального середовища. Система відслідковує, фільтрує, робить кореляцію і обробку тисяч подій, що відбуваються щодня в мережних пристроях, системах, базах даних і додатках.

Застосовуючи загальний користувальницький інтерфейс системи керування розподіленим обчислювальним середовищем, написаний мовою Java, для всіх керованих компонентів, адміністратори й оператори одержують швидкий і узгоджений доступ до стану найважливіших служб додатків і можливість швидко усувати проблеми. Вбудовані інтелектуальні механізми

кореляції подій, що мають просте налаштування, скорочують навантаження подій і збільшують кількість об'єктів, якими може керувати один оператор.

Використання комплексних концепцій керування

Комплексні концепції керування, властиві архітектурі системи керування розподіленим обчислювальним середовищем, дозволяють адаптуватися практично до будь-якої організаційної схеми. Ієрархічне керування забезпечує найкращу масштабованість:

- *Ролі користувачів (User roles),* організація розподіленої системи керування на основі різних ролей користувачів: операторів і адміністраторів. Оператори – відповідають тільки за визначений сегмент IT середовища, вирішують оперативні проблеми і можуть не бути технічними фахівцями. Адміністратори – розподіляють зони відповідальності операторів і приймають рішення в нестандартних випадках;
- *Керування "слідом за сонцем" (Follow-the-Sun),* при якому попередження розподіляються по різних центрах керування в залежності від пори дня. Це забезпечує безупинну роботу і цілодобовий доступ до системи керування;
- *Центри компетенції (Competence Centers).* Проблеми, що вимагають спеціальних знань, передаються в центри керування, де є відповідні фахівці. Таким чином, повністю використовується досвід IT-адміністраторів і досягається більш економічне рішення проблеми;
- *Зв'язки між центрами керування (Manager-to-Manager Communication)* дозволяють будувати ієрархію керування (наприклад, по географічних регіонах) і пересилати повідомлення, у тому числі наверх по ієрархії, у залежності від діючих правил, як автоматично, так і вручну, силами адміністраторів або операторів;
- *Резервування керуючого сервера (Backup Server Concept)* дозволяє передавати обов'язки по керуванню від одного центра керування до іншого у випадку збою системи. Таким чином, усуваються “місцеві” збої.

Не існує двох подібних обчислювальних середовищ. Але у системі існує вичерпний набір відкритих інтерфейсів для підтримки будь-якого додатка замовника і для тісної інтеграції з існуючими розробками в сфері керування. Спеціальні графічні інтерфейси (API) і інтерфейси командного рядка також дозволяють замовникам, що мають великі і змінні обчислювальні середовища, виконувати зовнішнє і динамічне налаштування, що значно полегшує технічне обслуговування.

Зв'язок зі службою підтримки і звіти про якість IT-послуг

Планово-попереджувальне обслуговування особливо важливе для підтримки послуг, оскільки будь-які порушення в електронній комерції і бізнесах-процесах негайно виявляються замовником. Система керування розподіленим обчислювальним середовищем тримає службу підтримки в курсі стану всіх бізнесів-додатків компанії і дозволяє її фахівцям без труднощів реагувати на виклики бізнесів-користувачів, що стосуються раніше виявлених несправностей. Поєднуються такі процеси, як керування викликами, керування інцидентами і керування проблемами, а також процеси керування якістю надання послуг. Таким чином, система дозволяє IT - організаціям із самого початку надати замовнику інтегроване, повномасштабне рішення для керування IT на рівні послуг. Це дозволяє службам підтримки користувача та IT-організаціям співробітничати і спільно володіти інформацією для того, щоб забезпечити роботу головних бізнес-служб.

Звіти про якість послуг є ключовим елементом відносин між IT-організаціями і бізнес- підрозділами підприємства. Життєво важливо, щоб IT - відділ міг точно відзвітуватися в якості послуг, що поставляються ним. Система керування розподіленим обчислювальним середовищем дозволяє IT-відділам надавати регулярні і точні звіти, що свідчать про рівень якості послуг як для IT-відділів, так і для менеджерів компанії.

Додаткова безпека в системі.

У середовищі електронного бізнесу розробки для керування повинні бути безпечними і не створювати додаткових проблем під час роботи. Забезпечення безпечних механізмів зв'язку для керування критично важливими і чуттєвими IT-засобами в потенційно небезпечній мережній інфраструктурі є ключовою вимогою для успіху в проведенні корпоративної стратегії безпеки. Система керування розподіленим обчислювальним середовищем постачається зі стандартним захистом проти пасивних атак, що забезпечується шляхом захисту всього мережного трафіка між центральною консоллю керування і розподіленими інтелектуальними агентами. Інфраструктура зв'язку системи доповнюється підтримкою засобів ідентифікації, шифрування і цілісності даних; ці засоби застосовуються до даних керування. Забезпечується безпека даних у каналах зв'язку між центральними керуючими серверами системи, розподіленими інтелектуальними агентами і користувальницькими інтерфейсами, написаними мовою Java.

Засоби керування і контролю якості роботи додатків.

Керування додатками

Засоби керування і контролю якості роботи додатків являють собою сукупність заздалегідь набудованих, легко встановлюваних модулів, інтегрованих з консоллю системи керування розподіленим обчислювальним середовищем і мають розширені можливості для керування основними базами даних, інфраструктурою Інтернету, службами обміну повідомленнями, бізнес-додатками і платформами електронної комерції.

Засоби керування і контролю якості роботи додатків вносять інтелектуальні можливості збору даних та моніторингу і рятують користувача від необхідності витрачати сили на придбання спеціальних навичок по керуванню конкретними додатками. Вони включають інструкції та керуючі дії для операторів, дозволяють виявляти кореляцію між різними значеннями параметрів продуктивності по всіх рівнях вибраної послуги, для того щоб розібратися у взаємозалежностях і швидко знайти основну причину збою. Засоби керування і контролю якості роботи додатків дозволяють керувати додатками на підставі заздалегідь підготовлених звітів і візуальних представлень послуг, що дозволяє оператору встановити, чи була порушена подією критично важлива послуга чи ні. Таким чином, IT-спеціалисти можуть встановити пріоритети робіт і в першу чергу займатися самими терміновими проблемами.

3.5. Megavision Web

Загальні характеристики

MegaVision Web — це повнофункціональна система мережного керування (NMS) на основі протоколу SNMP, що забезпечує керування і моніторинг всіма пристроями Optical Access сімейства OptiSwitch.

Система мережного керування (NMS) для керування і моніторингу всіма пристроями Optical Access.

Система мережного керування MegaVision Web призначена для оптимізації керування всіма пристроями мережі. Вона забезпечує швидке конфігурування налаштувань мережі, виявлення проблем, що виникають, керування портами мережних пристроїв, а також надання графічної і статистичної інформації про роботу мережі в масштабі реального часу. На додаток, MegaVision Web може виявляти в мережі практично будь-які SNMP чи TCP/IP-сумісні пристрої будь-яких виробників, якщо вини підтримують

стандартні функції SNMP MIB, і здійснювати для них функції керування. Усі функції конфігурування і моніторингу доступні через розширений графічний інтерфейс на базі Web. Серверний додаток MegaVision Web працює на платформі Windows 95/98/NT/2000. При використанні будь-якого Web-браузера з підтримкою Java, доступ до віддаленої консолі керування можливий з будь-якої точки, з любої комп'ютерної платформи, що працює з будь-якою операційною системою. MegaVision Web також може працювати як повнофункціональний компонент любої відомої платформи мережного керування, наприклад з HP OpenView, що дає можливість його легкої інтеграції в існуючу інфраструктуру керування. MegaVision Web має розширений графічний інтерфейс з механізмами для керування пристроями, збору статистики по пристроям, портам і потокам даних у реальному часі.

Система MegaVision Web дає можливість конфігурувати і відслідковувати роботу беспровідних каналів зв'язку, віртуальних мереж (VLAN), систем маршрутизації, мереж на базі DiffServ та інших механізмів забезпечення якості послуг (QoS). Адміністратор має можливість вибирати спосіб оповіщення про події в мережі, що можуть автоматично відсилатися по електронній пошті.

Розширений графічний інтерфейс на базі Web дозволяє будувати різні види графіків і діаграм для одного чи декількох пристроїв на одному екрані одночасно. Схема кольорового кодування стану пристроїв дозволяє легко відслідковувати появу будь-яких неполадок у мережі. Пов'язуючи різні стани пристрою з певної колірною схемою, адміністратор мережі може відразу помітити будь-яку зміну і відреагувати на неї.

Усі мережні пристрої є цілком керованими через графічний інтерфейс MegaVision Web. При натисканні на піктограму будь-якого модуля мережного пристрою відкривається спеціальне вікно керування даним модулем з відповідними параметрами і функціями. Стан індикаторів кожного пристрою і всіх його модулів обновлюється через кожен часовий інтервал огляду мережі. Адміністратор може логічно об’єднувати пристрої в групи і створювати зв'язки між картами і групами пристроїв. Для різних карт можуть бути встановлені різні індивідуальні фонові малюнки, що полегшує їхню ідентифікацію та навігацію між ними.

При використанні спеціального редактора конфігурацій, вся інформація про пристрої мережі і їх конфігурацію може бути збережена в окремих файлах.

Основні можливості

Загальні:

- Повне керування всіма продуктами Optical Access;
- Можливість здійснювати моніторинг любих SNMP- або TCP/IP-сумісних пристроїв;
- Керування з будь-якої платформи;
- Зручний інтерфейс на базі Web;
- Керування і моніторинг декількох пристроїв одночасно;
- Клієнт та сервер TFTP/Boot;
- Підтримка RMON (Групи 1,2,3,9) через графічний інтерфейс;
- Повна підтримка стандартних і власних MIB;
- Керування маршрутизацією, QoS, VLAN, DiffServ та іншими функціями через графічний інтерфейс;
- Вбудована можливість конфігурування не SNMP-сумісних пристроїв через Telnet;

Обробка подій у мережі:

- Автоматичне виявлення SNMP- чи TCP/IP-сумісних пристроїв;
- Повідомлення про події SNMP у мережі, які можна налаштовувати;
- Повідомлення про події через електронну пошту;
- Збереження архівної інформації про події;

Моніторинг продуктивності:

- Інформація про продуктивність пристрою/порту/інтерфейсу відображається у вигляді таблиць, графіків чи діаграм;
- Моніторинг продуктивності мережі і трафіка за будь-який період часу;
- Моніторинг подій з встановленням порогів і типів оповіщення;
- Моніторинг продуктивності для потоків DiffServ;
- Моніторинг мережних сервісів. Забезпечує моніторинг продуктивності таких служб 7-го рівня, як електронна пошта, ftp, електронна комерція (http) і т.д.

Інтерфейси:

- Можливість роботи в якості агента SNMP для інших додатків керування;
- Запити NMS до бази даних SQL;
- Пересилання повідомлень додаткам OSS.

Конфігурування пристроїв:

- Повне графічне керування пристроями;
- Керування Qo і DiffServ;
- Графічне налаштування і відображення VLAN;
- Графічна конфігурація систем маршрутизації;

- Пороги оповіщення, що задаються користувачем;
- Конфігурування інтервалів опитування і граничних значень;
- Редактор конфігурацій для збереження інформації про пристрої мережі та їх конфігурації в окремих файлах.

Захист інформації:

- Захист доступу за допомогою пароля;
- Різні рівні авторизації в системі, що дозволяють обмежити або заборонити можливості перегляду і зміни конфігурації мережі;
- Підтримка SNMPv3.

Система обробки неполадок Event Horizon для MegaVision

Event Horizon для MegaVision – це система обробки відмовлень (Fault Management System application, FMS), що має розширені можливості, недоступні при використанні звичайних систем оповіщення на базі SNMP, які лише відображають повідомлення і сигнали, що надходять. Event Horizon надає адміністраторам засоби для аналізу проблем, що виникають, і різні способи реагування на них. Наприклад, при прямому підключенні, потрібний пристрій або група пристроїв можуть бути активовані простим кліком миші. Є також вбудовані засоби віддаленого доступу до устаткування і керування резервними системами.

Віддалений моніторинг і журнал подій

Event Horizon являє собою повнофункціональну систему FMS, що забезпечує відображення текстової інформації, яка описує всі подробиці виникаючих збоїв і відмов у роботі. Усі сигнали зводяться в єдину таблицю, у якій можна знайти детальний опис кожного з них. Опис включає інформацію про ступінь важливості сигналу, його опис, тип устаткування, місцезнаходження, часі події та причини її виникнення. Усі сигнали, оповіщення та дії зберігаються в спеціальному файлі, по якому можна здійснювати пошук інформації та її фільтрацію. Це дозволяє робити аналіз подій для знаходження закономірностей, які призводять до виникнення проблем, що дає можливість уникнути їх у майбутньому.

Моніторинг спрацьовування датчиків

Event Horizon оповіщає оператора про зміну стану датчиків типу «сухий контакт», підключених до портів серверів In-Reach. Таким датчикам можна привласнювати власні імена для кращого розуміння інформації, що відображається на екрані. До опису цих подій може бути також прив'язана інформація про місцезнаходження датчиків. При спрацьовуванні датчика або контакту, інформація посилається системі Event Horizon, що відображає цю інформацію в зрозумілому для адміністратора вигляді, а не у вигляді коду SNMP. Це зменшує час реакції персоналу на виникаючі події, дозволяючи швидше вживати заходів по усуненню можливих проблем.

Інтегрований віддалений доступ

Ще однією перевагою Event Horizon перед стандартними системами на основі SNMP, є можливість двохнаправленого моніторингу і керування. Наприклад, якщо система моніторингу спрацьовування датчика пристрою In-Reach виявляє несправність, користувач може відразу активізувати сеанс Telnet з устаткуванням. Це дає можливість негайно вжити заходів по усуненню несправності. Користувач сам задає інформацію про те, як здійснювати доступ до віддалених компонентів, і вона зберігається в

базі даних. Вміст цієї бази даних налаштовується користувачем і може бути як незалежним, так і прив'язаним до певних подій. Можливість зв'язку сеансів Telnet з окремими подіями, дозволяє адміністраторам не тільки відслідковувати стан пристроїв, але і легко керувати ними в разі потреби.

Віддалене керування

Додатковою функцією систем Event Horizon і продуктів In-Reach є можливість налаштування «дій», що починаються у відповідь на те чи інше повідомлення або сигнал. Дії можуть складатися в активації резервних систем (наприклад електрогенераторів) чи систем охолодження перш, ніж ситуація стане критичною. Дії здійснюються за допомогою команди "set" SNMP. Наприклад, якщо надходить сигнал від віддаленого датчика температури, то можливе автоматичне включення кондиціонера в цьому приміщенні або вимикання устаткування, що перегрівається.

Віддалений моніторинг стану навколишнього середовища

Можливість продуктів серії In-Reach відслідковувати параметри навколишнього середовища, такі як чи температура або вологість, стає ще більш зручним при використанні системи Event Horizon. Датчики температури і вологості можуть бути налаштовані на посилання сигналів SNMP при перевищенні заданих порогів значень.

Відправка оповіщень по електронній пошті

На додаток до сигналів SNMP, система може також посилати повідомлення про події по електронній поштою Таким чином адміністратори будуть одержувати оперативну інформацію неза залежно від свого місцезнаходження.

Література

1. Ф.Н. Энслоу. Мультипроцессоры и параллельная обработка.–М.: "Мир", 1976.
2. Дорошенко А.Ю. Лекції з паралельних обчислювальних систем. Київ: Видавничий дім “КМ Академія”, 2003. – 42 с.
3. Открытые системы, 2000, №4.
4. Компьютер пресс, 2002, № 8, с. 99-100.
5. Р. Хокни, К. Джесхоуп. Параллельные ЭВМ.– М.: "Радио и связь", 1986.
6. Системы параллельной обработки / под ред. Д. Ивенса // М.: Мир, 1985.- 415с.

7. Дорошенко А.Е. Математические модели и методы организации высоко-производительных параллельных вычислений. - К.,"Наукова думка", 2000.- 177 с.
8. В. Корнеев, Параллельные вычислительные системы.–М. Ноулидж. – 1999.
9. http://parallel.ru/
10. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления.– СПб.: БХВ-Петербург, 2002.–608 с.
11. Н. Вирт, Программы = Алгоритмы + Структуры данных, М.: Мир, 1976.
12. Программирование на параллельных вычислительных системах / под ред. Р Бэбба II // М.: Мир. 1991.- 372 с.
13. J. M. Crichlow. An introduction to Distributed and Parallel Computing.– Prentice Hall, 1997.– 238 p.
14. Dijkstra E.W. Cooperating seqential processes, Technological University, Eindhoven, 1965.
15. Элементы параллельного программирования / В.А. Вальковский, В.Е. Котов, А.Г. Марчук, Н.Н. Миренков // М,: Радио и связь, 1983.- 240 с.
16. Programming languages, F. Genuys (ed.), Academic Press, New York, 1968.
17. Hoare C.A.R. Monitors: an operating system structuring concept // Commun. ACM. - 1974. - vol.17, N 10. - p. 549 -557.
18. Hoare C.A.R. Communicating sequential processes // Commun. ACM. - 1978. - vol.21, N 8. - p. 666 -677.
19. Вегнер П. Программирование на языке Ада / Пер. с англ. под ред. В.Ш. Кауфмана. - М.: Мир, 1982. - 240 с.
20. А. Ахо, Дж. Хопкрофт, Дж. Ульман, Построение и анализ вычислительных алгоритмов.-М.: Мир, 1979.- 536 с.
21. Дж.Вебер, Технология Java в подлиннике, BHV,СПб, 2000, 1104 с.
22. MPI: A Message-Passing Interface Standard. – Int. J. Of Supercomputer Applications and High Performance Computing. – 1994. – Vol. 8, No. 3/4.– P. 159 –416.
23. Немнюгин С.А. Стесик О.Л. Параллельноге программирование для многопроцессорных вычислительных систем.–СПб:БХВ-Петербург, 2002.–400 с.
24. Шеховцов В.А. Операційні системи. Видавнича група BHV, Київ, 2005. - 575 с.
25. Э. Таненбаум. Архитектура комп’ютера. Питер, 2003. - 698 с.
26. Э. Таненбаум. Современные операционные системы. Питер, 2002. - 1037 с.
27. G. Nutt. Operating Systems, Addison Wesley, 2004. - 994 p.

28. Д. Бэкон, Т. Харрис. Операционные системы. Паралельные и распределенные системы, Питер, 2004. - 799 с.
29. Э. Таненбаум, М. Ван Стеен. Распределенные системы. Принципы и парадигмы. Питер, 2003. - 876 с.
30. Д. Хорвиц. Unix-системы. От проектирования до сопровождения, DiaSoft, Москва, Спб., Киев, 2004. - 589 с.