Навчально-методична література

Міністерство освіти і науки України
Тернопільський національний технічний університет
ім. Івана Пулюя
Кафедра комп'ютерно-інтегрованих технологій

Конспект лекцій

з дисципліни

«Програмування систем реального часу»

напряму підготовки 6.050202 «Автоматизація та комп'ютерно-інтегровані технології»

Тернопіль – 2016

УДК 681.3
ББК 32.973

Укладачі:
Чихіра І.В., канд. техн. наук, доцент,
Микитишин А.Г., канд. техн. наук, доцент

Рецензент
Коноваленко І.В., канд. техн. наук, доцент

Схвалено та рекомендовано до друку на засіданні кафедри протокол №1 від 29.08.2016р.

Засідання факультету протокол №1 від 29.08.2016р.

Відповідальний за випуск *Чихіра І.В*., канд. техн. наук, доцент,

Лекція 1.

Вступ.

1. Поняття про операційні системи реального часу.

Операційні системи (ОС) загального призначення, особливо багатокористувацькі, такі как UNIX, Windows XP, орієнтовані на оптимальний розподіл ресурсів комп'ютера між користувачами і задачами (системи поділу часу). В операційних системах реального часу подібна задача відходить на другий план - усе відступає перед головною задачею - встигнути зреагувати на події, що відбуваються на об'єкті.

Інша відмінність - застосування операційної системи реального часу завжди зв'язано з апаратурою, з подіями, що відбуваються на об'єкті. Система реального часу, як апаратно-програмний комплекс, містить у собі датчики, що реєструють події на об'єкті, модулі вводу-виводу, що перетворюють покази датчиків в цифровий код, придатний для обробки цих показів на комп'ютері, і, нарешті, комп'ютер із програмою, що реагує на події, які відбуваються на об'єкті. Операційна система реального часу орієнтована на обробку зовнішніх подій. Саме це призводить до корінних відмінностей (в порівнянні з ОС загального призначення) у структурі системи, у функціях ядра, у побудові системи вводу-виводу. Операційна система реального часу може бути схожа по інтерфейсу користувача на ОС загального призначення (до цього, до речі, прагнуть майже усі виробники операційних системах реального часу).

Застосування операційних систем реального часу завжди конкретно. Якщо ОС загального призначення зазвичай сприймається користувачами (не розробниками) як вже готовий набір додатків, то операційна система реального часу служить тільки іструментом для створення конкретного апаратно-програмного комплексу реального часу. І тому найбільш широкий клас користувачів операційних систем реального часу - розробники комплексів реального часу, люди що проектують системи керування і збору даних. Проектуючи і розробляючи конкретну систему

реального часу, програміст завжди знає точно, які події можуть відбутися на об'єкті, знає критичні терміни обслуговування кожного з цих подій.

Назвемо системою реального часу (СРЧ) апаратно-програмний комплекс, що реагує в конкретний передбачуваний час на непередбачений потік зовнішніх подій. Це визначення означає, що:

- Система повинна встигнути відреагувати на подію, що відбулася на об'єкті, протягом часу, критичного для цієї події (meet deadline). Величина критичного часу для кожної події визначається об'єктом і самою подією, і, звичайно, може бути різною, але час реакції системи повинен бути передбачуваним (обчисленим) при створенні системи. Відсутність реакції в передбачений час вважається помилкою для систем реального часу.
- Система повинна встигати реагувати на події, що відбуваються одночасно. Навіть якщо дві чи більше зовнішніх подій відбуваються одночасно, система повинна встигнути зреагувати на кожну з них протягом інтервалів часу, критичних для цих подій.

Отже, основні вимоги до операційних систем реального часу (ОСРЧ):

- вимоги за часом;
- можливість паралельного виконання декількох задач;
- передбачуваність;
- особливі вимоги в питаннях безпеки;
- можливість безвідмовної роботи на протязі тривалого періоду часу.

2. Характеристики та класифікація ОСРЧ.

Загальні характеристики ОСРЧ:

- великі і складні системи;
- розподілені системи;
- жорстка взаємодія з апаратурою;
- виконання задач залежить від часу;
- складність у тестуванні.

ОСРЧ повинні реагувати на різні типи внутрішніх і зовнішніх подій (періодичних і неперіодичних). Необхідно відзначити, що приналежність системи

до класу ОСРЧ ніяк не пов'язана з її швидкодією. Вихідні вимоги до часу реакції системи й інших тимчасових параметрів визначаються технічним завданням на систему, чи просто логікою її функціонування. Зрозуміло, що швидкодія ОСРЧ повинна бути тим більше, чим більше швидкість протікання процесів на об'єкті контролю і керування. При роботі із широкосмуговими по своїй природі перехідними процесами часто застосовують швидкодіючі АЦП із буферною пам'яттю, куди з необхідною швидкістю записуються значення сигналу, які потім аналізуються і/або реєструється обчислювальною системою. При цьому потрібно закінчити всю необхідну обробку до наступного перехідного процесу, інакше інформація буде загублена.

Розрізняють системи реального часу двох типів - системи жорсткого і м'якого реального часу.

1. Системою жорсткого реального часу називається система, де нездатність забезпечити реакцію на будь-які події в заданий час є відказом і веде до неможливості рішення поставленої задачі. Багато теоретиків ставлять тут крапку, з чого випливає, що час реакції у жорстких системах може складати і секунди, і години, і тижні. Однак більшість практиків вважають, что час реакції в системах жорсткого реального часу повинен бути все-таки мінімальним.

Системи жорсткого реального часу не допускають ніяких затримок реакції системи ні при яких умовах, тому що:

- результати можуть виявитися марні у випадку запізнення,
- може відбутися катастрофа у випадку затримки реакції,
- вартість запізнення може виявитися нескінченно велика.

Більшість систем жорсткого реального часу є системами контролю і керування. Такі СРЧ складні в реалізації, тому що для них пред'являються особливі вимоги в питаннях безпеки. Приклади систем жорсткого реального часу - бортові системи керування, системи аварійного захисту, реєстратори аварійних подій.

2. Системи м'якого реального часу характеризуються тим, що затримка реакції не критична, хоча і може привести до збільшення вартості результатів і зниження продуктивності системи в цілому. Так як система м'якого реального часу

може не встигати все робити завжди в заданий час, виникає проблема визначення критеріїв успішності (нормальності) її функціювання.

Основну відмінність між системами жорсткого і м'якого реального часу можна виразити так: система жорсткого реального часу ніколи не спізниться з реакцією на подію, система м'якого реального часу - не повинна спізнюватися з реакцією на подію.

Також СРЧ можна розділити на системи спеціалізовані й універсальні**:**

1. Спеціалізованою СРЧ називається система де конкретні тимчасові вимоги зазвичай визначені. Така система повинна бути спеціально спроектована для задоволення цих вимог.

2. Універсальна СРЧ повинна уміти виконувати довільні (заздалегідь не визначені) тимчасові задачі без застосування спеціальної техніки. Розробка таких систем безумовно є самою складною задачею, хоч зазвичай, вимоги, пропоновані до таких систем, м'якші ніж вимоги для спеціалізованих систем.

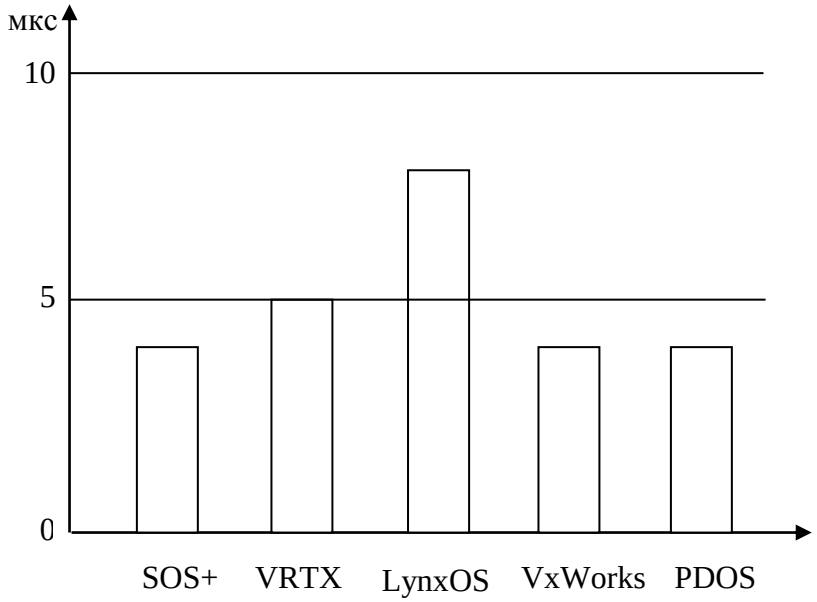

Рис.1 Час реакції різних систем на переривання

3. Основні параметри ОСРЧ

3.1 Час реакції системи. Майже усі виробники систем реального часу приводять такий параметр, як час реакції системи на переривання (interrupt latency). Справді, якщо головним для системи реального часу є її властивість вчасно відреагувати на зовнішні події, то такий параметр, як час реакції системи є

ключовим. Однак, в даний момент, на жаль, немає загальноприйнятих методик виміру цього параметра, тому через це ведуться суперечки між маркетинговими службами виробників систем реального часу. Є надія, что незабаром положення зміниться, тому що вже розпочався проект порівняння операційних системах реального часу, який містять у собі, в тому числі, і розробку методик тестування.

Події, що відбуваються на об'єкті, реєструються датчиками, дані з датчиків передаються в модулі вводу-виводу (інтерфейси) системи. Модулі вводу-виводу, одержавши інформацію від датчиків і перетворивши її, генерують запит на переривання в керуючому комп'ютері, подаючи йому тим самим сигнал про те, що на об'єкті відбулася подія. Одержавши сигнал від модуля вводу-виводу, система повинна запустити програму обробки цієї події. Інтервал часу - від події на об'єкті і до виконання першої інструкції в програмі обробки цієї події і є часом реакції системи на події, і, проектуючи систему реального часу, розроблювачі повинні вміти обчислювати цей інтервал.

З чого від складається? Час виконання ланцюжка дій - від події на об'єкті до генерації переривання - ніяк не залежить від операційних систем реального часу і цілком визначається апаратурою, а от інтервал часу - від виникнення запиту на переривання і до виконання першої інструкції оброблювача визначається цілком властивостями операційної системи й архітектурою комп'ютера. Причому цей час потрібно вміти оцінювати в гіршій для системи ситуації, тобто в припущенні, що процесор завантажений, що в цей час можуть відбуватися інші переривання, що система може виконувати якісь дії, що блокують переривання. Непоганою підставою для оцінки часів реакції системи можуть служити результати тестування з докладним описом архітектури цільової системи, у якій проводилися виміри, засобів виміру і точною вказівкою, які проміжки часу вимірялися. Деякі виробники операційних систем реального часу надають результати такого тестування.

3.2 Час перемикання між контекстами. В операційні системи реального часу закладений паралелізм, можливість одночасної обробки декількох подій, тому всі ОСРЧ являються багатозадачними (багатопроцесорними, багато нитковими). Для того, щоб вміти оцінювати накладні витрати системи при обробці паралельних подій, необхідно знати час, який система витрачає на передачу управління від

процесу до процесу (від задачі до задачі, від потоку до потоку), тобто час перемикання контексту (рис.2).

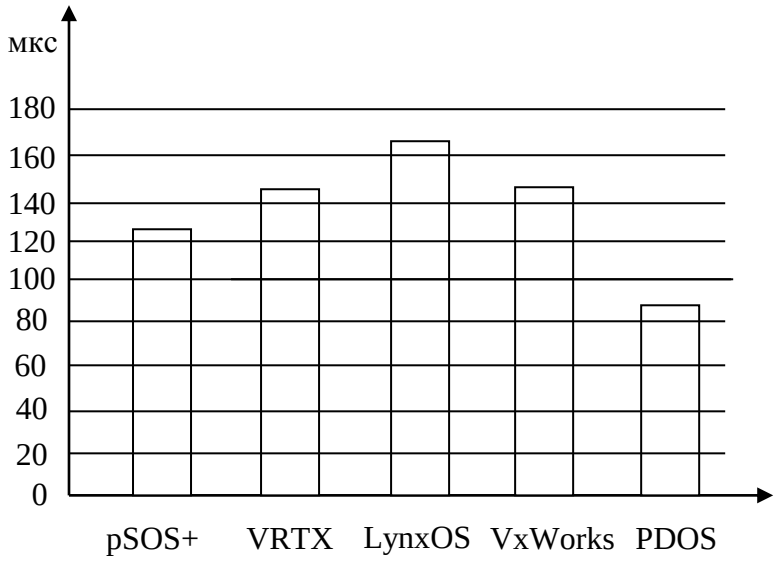

Рис.2 Час переключення контексту

Лекція 2

Базове забезпечення ОСРЧ

1. Класифікація ОСРЧ в залежності від програмного середовища.

Процес проектування конкретної системи реального часу починається з ретельного вивчення об'єкта. Розроблювачі проекту досліджують об'єкт, вивчають можливі події на ньому, визначають критичні терміни реакції системи на кожну подію і розробляють алгоритми обробки цих подій. Потім виконують процес проектування і розробки програмних додатків. Бажаною для кожного розроблювача є ідеальна операційна система реального часу, у якій додатки реального часу розробляються мовою подій об'єкта. Така система має свою назву, хоча й існує тільки в теорії. Називається вона: "система, керована критичними термінами". Розробка додатків реального часу в цій системі зводиться до опису можливих подій на об'єкті. В кожному описувачі подій вказується два параметри: часовий інтервал - критичний час обслуговування даної події й адреса підпрограми його обробки. Усю подальшу турботу про те, щоб підпрограма обробки події стартувала до витікання критичного інтервалу часу бере на себе операційна система. Але це - мрія. У реальності ж розроблювач повинний перевести мову подій об'єкта в сценарій багатозадачної роботи додатків операційних систем реального часу, намагаючись оптимально використовувати надані йому спеціальні механізми й оцінити часи реакцій системи на зовнішні події при цьому сценарії.

Будь-яка ОС зобов'язана забезпечити повний цикл життя програмного забезпечення: створення тексту програми, її компіляція, компонування, налагодження, виконання, супровід. Задачі реального часу висувають свої вимоги до обчислювально-керуючих систем, у тому числі до ОС, у яких реалізоване програмне забезпечення реального часу. Ці вимоги викладені в стандарті POSIX 1003.4. Стандарт визначає ОС як систему реального часу, якщо вона забезпечує необхідний рівень сервісу за цілком визначений, обмежений час. Тобто ОС реального часу повинна бути передбачуваною. Правильна, але спізніла реакція системи на зовнішню подію може бути згубною в системах безпеки атомних станцій, системах керування повітряними транспортними потоками і т.д. При цьому важливий не

тільки абсолютний час реакції системи, але і те, що він визначений заздалегідь. У системі керування прокатним станом час реакції системи повинний бути в межах декількох мілісекунд, а в системі контролю за навколишнім середовищем - кілька хвилин. Проте обоє ці приклади - з області задач реального часу. Виникає питання, чи можна задачі реального часу вирішувати за допомогою систем загального призначення (Windows, UNIX і т.д.)? Головна вимога, пропонована до систем загального призначення, полягає в тім, що вони повинні забезпечити оптимальний поділ усіх ресурсів між усіма процесами. Відповідно, не повинно бути високопріоритетних задач, що використовували б який-небудь ресурс системи стільки, скільки їм необхідно. Треба усе-таки врахувати, що так чи інакше, розроблювачі ОС досягають компромісу між механізмом пріоритетності і згаданою вимогою.

UNIX став de-facto стандартом ОС загального призначення. Він реалізований і на мікро-, і на суперкомп'ютерах. Багато міжнародних програмних стандартів і угоди засновані на UNIX: POSIX, SVID (UNIX System V Interface Definitions), BSD 43 UNIX Socket і т.д. Однак ОС UNIX, розроблена як система загального призначення, не має ефективного механізму пріоритетності задач і тому мало придатна для задач реального часу. У той же час багато ОС реального часу можна охарактеризувати як UNIX-подібні. Стає очевидним те, що задачі реального часу необхідно реалізовувати в рамках специфічного системного програмного середовища. Системи реального часу можна розділити на 4 класи.

1-й клас: програмування на рівні мікропроцесорів. При цьому програми для програмованих мікропроцесорів, що вбудовуються в різні пристрої, дуже невеликі і зазвичай написані мовою низького рівня типу Асемблера. Внутрісхемні емулятори придатні тільки для налагодження, високорівневі засоби розробки і налагодження програм не застосовні. Операційне середовище зазвичай недоступне.

2-й клас: мінімальне ядро системи реального часу. На більш високому рівні знаходяться системи реального часу, що забезпечують мінімальне середовище виконання. Передбачені лише основні функції, а керування пам'яттю і диспетчер часто недоступні. Ядро являє собою набір програм, що виконують типові, необхідні для вбудованих систем низького рівня функції, такі, як операції з плаваючою

крапкою, і мінімальний сервіс вводу-виводу. Прикладна програма розробляється в інструментальному середовищі, а виконується, як правило, на вбудованих системах.

3-й клас: ядро системи реального часу й інструментальне середовище. Цей клас систем має багато рис ОС з повним сервісом. Розробка ведеться в інструментальному середовищі, а виконання - на цільових системах. Цей тип систем забезпечує набагато більш високий рівень сервісу для розроблювача прикладної програми. Сюди включені такі засоби, як дистанційний символьний відладчик, протокол помилок і інші засоби відлагодження. Часто доступно рівнобіжне виконання програм.

4-й клас: ОС з повним сервісом. Такі ОС можуть бути застосовані для будь-яких додатків реального часу. Розробка і виконання прикладних програм ведуться в рамках однієї і тієї ж системи.

Системи 2 і 3 класів відносяться до систем "жорсткого" реального часу, а 4 клас - "м'якого". Очевидно, це можна пояснити тим, що в першому випадку до системи пред'являються більш жорсткі вимоги за часом реакції і необхідному обсягу пам'яті, ніж у другому. Як ми бачимо, середовище розробки і середовище виконання в системах реального часу можуть бути розділені, а вимоги, пропоновані до них, дуже різні.

2. Середовище виконання ОСРЧ.

Вибираючи ОС загального призначення, звертаємо увагу на її поширеність, на зручність і комфортабельність систем розробки, на інструментарій системи: мережа, бази даних, офісні пакети й ін. А що важливо для операційних систем реального часу? Яка структура цих продуктів?

Одне з корінних зовнішніх відмінностей систем реального часу від систем загального призначення - чітке розмежування систем розробки і систем виконання. Система виконання операційних систем реального часу – це набір інструментів (ядро, драйвери, що виконуються, модулі), що забезпечують функціонування додатка реального часу. Більшість сучасних ведучих операційних систем реального часу підтримують цілий спектр апаратних архітектур, на яких працюють системи виконання (Intel, Motorola, RISC, MIPS, PowerPC, і інші). Це пояснюється тим, що

набір апаратних засобів - частина комплексу реального часу й апаратура повинні бути також адекватні поставленій задачі, тому ведучі операційні системи реального часу перекривають цілий ряд найбільш популярних архітектур, щоб задовільнити самим різним вимогам по частині апаратури. Система виконання операційної системи реального часу і комп'ютер, на якому вона виконується називають "цільовою" (target) системою.

Вимоги, до середовища виконання систем реального часу, наступні:

- невелика пам'ять системи - для можливості її вбудовування;
- система повинна бути цілком резидентна в пам'яті, щоб уникнути заміщення сторінок пам'яті чи підкачування;
- система повинна бути багатозадачною - для забезпечення максимально ефективного використання всіх ресурсів системи;
- ядро з пріоритетом на обслуговування переривання. Пріоритет на переривання означає, що готовий до запуску процес, що володіє деяким пріоритетом, обов'язково має перевагу в черзі стосовно процесу з нижчим пріоритетом, швидко заміняє останній і надходить на виконання. Ядро закінчує будь-яку сервісну роботу, як тільки надходить задача з вищим пріоритетом. Це гарантує передбачуваність системи;
- диспетчер із пріоритетом - дає можливість розроблювачу прикладної програми привласнити кожному завантажувальному модулю пріоритет, непідвласний системі. Присвоєння пріоритетів використовується для визначення черговості запуску програм, готових до виконання. Альтернативним такому типу диспетчеризації є диспетчеризація типу "карусель", при якій кожній готовій до продовження програмі дається рівний шанс запуску. При використанні цього методу немає контролю за тим, яка програма і коли буде виконуватися. У середовищі реального часу це неприпустимо. Диспетчеризація, в основу якої покладений принцип присвоєння пріоритету, і наявність ядра з пріоритетом на переривання дозволяють розроблювачу прикладної програми цілком контролювати систему. Якщо настає подія з вищим пріоритетом, система припиняє обробку задачі з нижчим пріоритетом і відповідає на запит, що надійшов з вищим пріоритетом.

Сполучення описаних вище властивостей створює могутнє й ефективне середовище виконання в реальному часі.

3. Середовище розробки ОСРЧ.

Система розробки – це набір засобів, які забезпечують створення і налагодження додатку реального часу. Системи розробки (компілятори, відладчики і всілякі інструменти) працюють, як правило, у популярних і розповсюджених ОС, таких, як UNIX і Windows. Крім того, багато операційних систем реального часу мають і так звані резидентні засобу розробки, що виконуються в середовищі самої операційної системи реального часу - особливо це відноситься до операційних систем реального часу класу "ядра". Функціонально, засоби розробки операційних систем реального часу відрізняються від звичних систем розробки, таких, наприклад, як Developers Studio, TaskBuilder, тому що часто вони містять засоби віддаленого налагодження, засоби профілювання (вимір часів виконання окремих ділянок коду), засоби емуляції цільового процесора, спеціальні засоби налагодження взаємодіючих задач, а іноді і засоби моделювання.

Для розроблювача прикладних програм вкрай важлива відкритість системи, у якій він працює, і стандарти, якими він користується. Відкрита система означає незалежність розроблювача. Стандарти роблять прикладну програму взаємопогоджуваною з іншими системами. У цьому аспекті варто приділити увагу стандарту на операційні системи - POSIX. Він покликаний забезпечити переносимість додатків між різними платформами. Одна з найважливіших частин цього стандарту присвячена забезпеченню мобільності додатків реального часу. Для цього стандартизуються необхідні програмні інтерфейси (диспетчеризація і синхронізація процесів, забезпечення вводу/виводу і ін.), підтримка "потоків" і процесів, таймаути, керування перериваннями, профілі прикладних контекстів реального часу. Цей стандарт одержує усе більше поширення і підвищує гарантії живучості операційної системи.

Вартість проекту будь-якої системи реального часу визначається вартістю програмного забезпечення. Тому усе більше уваги приділяється середовищу розробки програм. Більш досконалі методи і засоби розробки, налагодження і

підтримки прикладних програм дозволяють їхнім розроблювачам робити більш якісний програмний продукт за менший час. Для розробки прикладних програм необхідні визначені засоби: редактори, компілятори, компоновщики й відладчики. Розроблювачі хочуть мати інтегровані програмні засоби, що охоплюють весь цикл розробки: аналіз, проектування, кодування, тестування, документування і підтримку задачі.

4. POSIX.

4.1 Призначення стандарту POSIX.

Як випливає з назви, POSIX (Portable Operating System Interface) - це стандарт на сполучення (інтерфейс) між операційною системою і прикладною програмою. У тексті POSIX неодноразово підкреслюється, що стандарт не висуває ніяких вимог до деталей реалізації операційної системи; його можна розглядати як сукупність домовленостей між прикладними програмістами і розроблювачами операційних систем. Таким чином, POSIX становить інтерес не тільки для розроблювачів операційних систем, але насамперед для більш численної категорії програмістів - прикладних.

Потреба в стандарті такого роду була усвідомлена ще в 80-і роки, коли одержали широке поширення операційні системи UNIX. Виявилося, що хоча ця система і задумувалася як уніфікована, розходження між конкретними її реалізаціями приводили до того, що прикладні програми, написані для однієї системи, далеко не завжди могли виконуватися в іншій. На рішення саме цієї проблеми, відомої як проблема мобільності програмного забезпечення, націлений POSIX. Перша редакція стандарту була випущена в 1988 році у якій усе різноманіття питань, зв'язаних з мобільністю програм, було розбито на дві частини: 1- інтерфейс прикладних програм, 2- командний інтерпретатор і утиліти (інтерфейс користувача). Ці частини одержали назву POSIX.1 і POSIX.2 відповідно.

Друга редакція стандарту на інтерфейс прикладних програм POSIX.1 була затверджена 12 липня 1996 року. В інформативній частині стандарту підкреслюється, що POSIX являє собою не опис інтерфейсу деякої "ідеальної" операційної системи, а результат узагальнення і систематизації досвіду,

накопиченого при розробці операційних систем UNIX. Крім того, POSIX не може служити навчальним посібником по операційних системах, хоча в інформативній частині містяться рекомендації програмістам і фрагменти програм.

У стандарті прямо говориться про те, що неможливо створити повноцінну операційну систему, орієнтуючись винятково на описані в ньому інтерфейсні функції. (Зокрема, у POSIX.1 не відбиті такі питання, як робота в мережі і відповідні інтерфейсні функції чи графічний інтерфейс.) Тим не менше, фінансові витрати, викликані немобільністю програм і потреба, що звідси виникає, в уніфікації інтерфейсу настільки великі, що більшість виробників прагне мати хоч який-небудь стандарт, ніж не мати ніякого. З цієї причини на POSIX орієнтується багато розроблювачів програмного забезпечення. Це дозволяє якщо не ліквідувати немобільність програм цілком, то принаймні істотно зменшити немобільну частину програми.

4.2 Процеси і потоки керування в стандарті POSIX.

Два цих терміна виражають ідею паралельності виконання. Операційна система UNIX була споконвічно задумана як багатокористувацька, і програми, що запускаються різними користувачами, повинні бути надійно ізольований одна від одної , щоб випадково не спотворити "чужі" дані. Ця ізоляція забезпечена тим, що програма користувача виконується в рамках деякого процесу, що розвивається у власному віртуальному адресному просторі. Навіть якщо в програмі є глобальні дані, при запуску її в різних процесах вони будуть автоматично "розведені" по різних адресних просторах.

Однак механізм процесів не цілком задовільний при програмуванні задач реального часу. Прикладна програма реального часу (що виконується від імені того самого користувача) часто може бути природним чином представлена у виді паралельно виконуючих частин, що називають "потоками керування". Найбільш істотна їхня відмінність від процесів полягає в тому, що всі потоки керування розвиваються в єдиному адресному просторі. Цим забезпечується швидкий доступ до глобальних даних, але одночасно виникає ризик ненавмисного їхнього перекручування, і щоб цього не відбувалося, необхідно дотримуватись деякої

дисципліни програмування. Відповідні рекомендації містяться в інформативній частині стандарту.

Потрібно підкреслити, що ідея багатопоточності реалізована в багатьох операційних системах реального часу, і реалізована по-різному в тому розумінні, що кожному потоку керування відповідають різні множини атрибутів і інтерфейсних функцій; іноді замість терміна "потік керування" використовується термін "задача". Для того щоб уникнути непорозумінь, у стандарті підкреслюється, що мова в ньому йде винятково про потоки керування POSIX.

Лекція 3

Архітектура ОСРЧ

1. Ядра і ОСРЧ.

Всі ОСРЧ сьогодні є багатозадачними системами. Задачі поділяють між собою ресурси обчислювальної системи, у тому числі і процесорний час.

Чіткої границі між ядром і операційною системою немає. Розрізняють їх, як правило, по наборі функціональних можливостей. Ядра надають користувачу такі базові функції, як планування, синхронізація задач, міжзадачна комунікація, керування пам'яттю і т.д. Операційні системи на додаток до цього мають файлову систему, підтримку мережі, інтерфейс з оператором і інші засоби високого рівня.

По своїй внутрішній архітектурі ОСРЧ можна умовно розділити на:

- монолітні ОС;
- ОС на основі мікроядра;
- об’єктно-орієнтовні ОС.

Важливою частиною будь-якої ОСРЧ є планувальник задач, функція якого - визначити, яка з задач повинна виконуватися в системі в кожен конкретний момент часу. До основних методів планування звичайно відносять: циклічний алгоритм, поділ часу з рівнодоступністю, кооперативну багатозадачність**.** Найбільш часто використовуваний в ОСРЧ принцип планування - пріоритетна багатозадачність з витісненням**.** Основна ідея полягає в тому, что високо пріоритетна задача, як тільки для неї з'являється робота, негайно перериває (витісняє) низько пріоритетну. Однак діапазон систем реального часу дуже широкий, починаючи від цілком статичних систем, де всі задачі і їх пріоритети заздалегідь визначені, до динамічних систем, де набір виконуваних задач, їх пріоритети і навіть алгоритми планування можуть мінятися в процесі функціонування. Існують, наприклад, системи, де кожна окрема задача може брати участь в кожному із трьох алгоритмів планування чи їх комбінації (витиснення**,** поділ часу, кооперативність). Крім того, пріоритети теж можна призначати по-різному. У загальному випадку алгоритми планування повинні відповідати критеріям оптимальності функціонування системи. Однак, якщо для систем жорсткого реального часу такий критерій очевидний “завжди й усе

робити вчасно”, то для систем м'якого реального часу це може бути, наприклад, мінімальне (максимальне) запізнення чи середньозважена своєчасність завершення операцій. У залежності від критеріїв оптимальності можуть застосовуватися алгоритми планування задач відмінні від розглянутих. Наприклад, може виявитися що планувальник повинен аналізувати момент видачі критичних за часом керуючих впливів і запускати на виконання ту задачу, що відповідає за найближчий з них.

2. Монолітні ОСРЧ.

У загальному випадку "структура" монолітної системи являє собою відсутність структури (рис. 1). ОС написана як набір процедур, кожна з який може викликати інші, коли їй це потрібно. При використанні цієї техніки кожна процедура системи має добре визначений інтерфейс у термінах параметрів і результатів, і кожна вільна викликати будь-яку іншу для виконання деякої потрібної для неї корисної роботи.

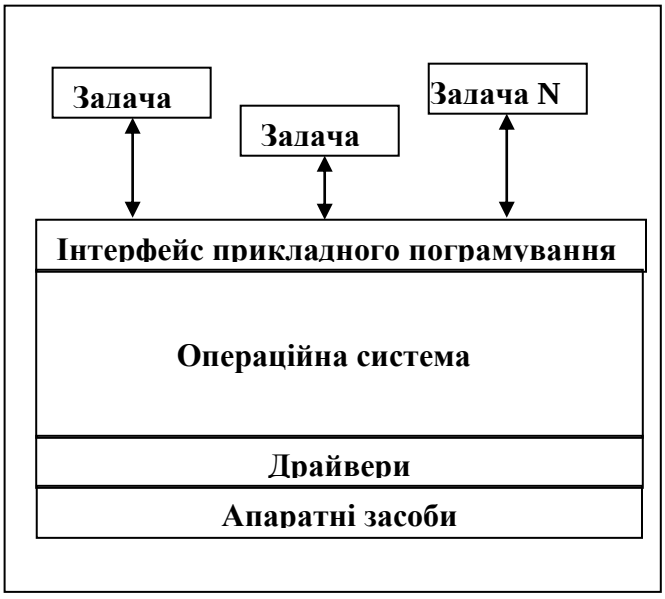

рис.1 ОСРЧ з монолітною структурою

Для побудови монолітної системи необхідно скомпілювати всі окремі процедури, а потім зв'язати їх разом у єдиний об'єктний файл за допомогою компоновщика (прикладами можуть служити ранні версії ядра UNIX чи Novell NetWare). Кожна процедура бачить будь-яку іншу процедуру (на відміну від структури, що містить модулі, у якій велика частина інформації є локальною для модуля, і процедури модуля можна викликати тільки через спеціально визначені точки входу).

Однак навіть такі монолітні системи можуть бути структурованими. При звертанні до системних викликів, які підтримуються ОС, параметри поміщаються в строго визначені місця, такі, як регістри чи стек, а потім виконується спеціальна команда переривання, відома як виклик ядра чи виклик супервізора. Ця команда переключає машину з режиму користувача в режим ядра, названий також режимом супервізора, і передає керування ОС. Потім ОС перевіряє параметри виклику для того, щоб визначити, який системний виклик повинен бути виконаний. Після цього ОС індексує таблицю, що містить посилання на процедури, і викликає відповідну процедуру. Така організація ОС припускає наступну структуру:

1. Головна програма, що викликає необхідні сервісні процедури.
2. Набір сервісних процедур, що реалізують системні виклики.
3. Набір утиліт, що обслуговують сервісні процедури.

3. ОС на основі мікроядра (модель клієнт-сервер).

Модель клієнт-сервер - це ще один підхід до структурування ОСРЧ. У широкому змісті модель клієнт-сервер припускає наявність програмного компонента - споживача якого-небудь сервісу - клієнта, і програмного компонента - постачальника цього сервісу - сервера. Взаємодія між клієнтом і сервером стандартизується, так що сервер може обслуговувати клієнтів, реалізованих різними способами. При цьому головною вимогою є те, щоб вони запитували послуги сервера зрозумілим йому способом. Ініціатором обміну звичайно є клієнт, що надсилає запит на обслуговування серверу, що знаходиться в стані очікування запиту. Той самий програмний компонент може бути клієнтом стосовно одного виду послуг, і сервером для іншого виду послуг. Модель клієнт-сервер успішно застосовується не тільки при побудові ОС, але і на всіх рівнях програмного забезпечення, і має в деяких випадках більш вузький, специфічний зміст, зберігаючи, звичайно, при цьому усі свої загальні риси. Стосовно до структурування ОС ідея складається в розбивці її на кілька процесів - менеджерів, кожний з який виконує окремий набір сервісних функцій - наприклад, керування пам'яттю, чи створення планування процесів. Кожний менеджер (сервіс) виконується в користувальницькому режимі.

рис.2 ОСРЧ на основі мікроядра

Клієнт, яким може бути або інший компонент ОС, або прикладна програма, запитує сервіс, посилаючи повідомлення на сервер. Ядро ОС (назване тут мікроядром), працюючи в привілейованому режимі, доставляє повідомлення потрібному серверу, сервер виконує операцію, після чого ядро повертає результати клієнту за допомогою іншого повідомлення (рис. 2).

Підхід з використанням мікроядра замінив вертикальний розподіл функцій операційної системи на горизонтальний. Компоненти, що лежать вище мікроядра, хоча і використовують повідомлення, що пересилаються через мікроядро, взаємодіють один з одним безпосередньо. Мікроядро відіграє роль регулювальника. Воно перевіряє повідомлення, пересилає їх між серверами і клієнтами, і надає доступ до апаратури.

Дана теоретична модель є ідеалізованим описом системи клієнт-сервер, у якій ядро складається тільки з засобів передачі повідомлень. У дійсності різні варіанти реалізації моделі клієнт-сервер у структурі ОС можуть істотно розрізнятися по обсязі робіт, виконуваних у режимі ядра.

Мікроядро реалізує життєво важливі функції, що лежать в основі операційної системи. Це базис для менш істотних системних служб і додатків. Саме питання про те, які із системних функцій вважати несуттєвими, і, відповідно, не включати їх до складу ядра, є предметом суперечки серед прихильників ідеї мікроядра. У загальному випадку, підсистеми, що були традиційно невід'ємними частинами операційної системи - файлові системи, керування вікнами і забезпечення безпеки - стають периферійними модулями, взаємодіючими з ядром і один з одним.

Головний принцип поділу роботи між мікроядром і навколишніми його модулями - включати в мікроядро тільки ті функції, яким абсолютно необхідно виконуватись в режимі супервізора й у привілейованому просторі. Під цим звичайно мають на увазі машинозалежні програми (включаючи підтримку декількох процесорів), деякі функції керування процесами, обробка переривань, підтримка пересилання повідомлень, деякі функції керування пристроями вводу-виводу, зв'язані з завантаженням команд у регістри пристроїв. Ці функції операційної системи важко, якщо не неможливо, виконати програмам, що працюють у просторі користувача.

Є два шляхи рішення цієї проблеми. Один шлях - розмістити декілька таких, чуттєвих до режиму роботи процесора, серверів, у просторі ядра, що забезпечить їм повний доступ до апаратури і у той же час, зв'язок з іншими процесами за допомогою звичайного механізму повідомлень. Такий підхід був використаний, наприклад, при розробці Windows: крім мікроядра, у привілейованому режимі працює частина Windows, названа executive керуючою програмою. Вона включає ряд компонентів, що керують віртуальною пам'яттю, об'єктами, вводом-виводом і файловою системою, взаємодією процесів, і частково системою безпеки.

Інший шлях, полягає в тому, щоб залишити в ядрі тільки невелику частину сервера, що представляє собою механізм реалізації рішення, а частина, що відповідає за ухвалення рішення, перемістити в користувальницьку область. Відповідно до цього підходу, наприклад, у мікроядрі Mach, на базі якого розроблена Workplace OS, розміщається тільки частина системи керування процесами (і нитками), що реалізує диспетчеризацію (тобто безпосереднє переключення з процесу на процес), а усі функції, зв'язані з аналізом пріоритетів, вибором чергового

процесу для активізації, ухваленням рішення про переключення на новий процес і інші аналогічні функції виконуються поза мікроядром. Цей підхід вимагає тісної взаємодії між зовнішнім планувальником і резидентним диспетчером.

Важливо зробити розходження - запуск процесу чи нитки вимагає доступу до апаратури, так що по логіці - це функція ядра. Але ядру все рівно, яку з ниток запускати, тому рішення про пріоритети ниток і дисципліні постановки в чергу може приймати працюючий поза ядром планувальник.

Крім вже представлених міркувань, переміщення планувальника на користувальницький рівень може знадобитися для чисто комерційних цілей. Деякі виробники ОС (наприклад, IBM і OSF зі своїми варіантами мікроядра Mach) планують ліцензувати своє мікроядро іншим постачальникам, яким може знадобитися замінити вихідний планувальник на інший, підтримуючий, наприклад, планування в задачах реального часу чи реалізуючи якийсь спеціальний алгоритм планування. А от інша ОС - Windows, що також використовує мікроядерну концепцію - втілила поняття пріоритетів реального часу у своєму планувальнику, резидентно розташованому в ядрі, і це не дає можливості замінити її планувальник на інший.

Драйвери пристроїв також можуть розташовуватися як Як і керування процесами, керування пам'яттю може розподілятися між мікроядром і сервером, що працює в користувальницькому режимі.

усередині ядра, так і поза ним. При розміщенні драйверів пристроїв поза мікроядром для забезпечення можливості дозволу і заборони переривань, частина програми драйвера повинна виконуватись в просторі ядра. Відділення драйверів пристроїв від ядра уможливлює динамічну конфігурацію ОС. Крім динамічної конфігурації, є й інші причини розглядати драйвери пристроїв як процеси користувальницького режиму. СУБД, наприклад, може мати свій драйвер, оптимізований під конкретний вид доступу до диска, але його не можна буде підключити, якщо драйвери будуть розташовані в ядрі.

В даний час саме операційні системи, побудовані з використанням моделі клієнт-сервер і концепції мікроядра, найбільшою мірою задовольняють вимогам, пропонованим до сучасних ОСРЧ. Високий ступінь переносимості обумовлений

тим, що весь машино-залежний код ізольований у мікроядрі, тому для переносу системи на новий процесор потрібно менше змін і усі вони логічно згруповані разом.

Технологія мікроядер є основою побудови множинних прикладних середовищ, що забезпечують сполучність програм, написаних для різних ОС. Абстрагуючи інтерфейси прикладних програм від розташованих нижче операційних систем, мікроядра дозволяють гарантувати, що вкладення в прикладні програми не пропадуть протягом декількох років, навіть якщо будуть змінюватися операційні системи і процесори.

Звичайно операційна система виконується тільки в режимі ядра, а прикладні програми - тільки в режимі користувача, за винятком тих випадків, коли вони звертаються до ядра за виконанням системних функцій. На відміну від звичайних систем, система побудована на мікроядрі, виконує свої серверні підсистеми в режимі користувача, як звичайні прикладні програми. Така структура дозволяє змінювати і додавати сервери, не впливаючи на цілісність мікроядра.

Використання моделі клієнт-сервер підвищує надійність. Кожен сервер виконується у виді окремого процесу у своїй власній області пам'яті, і в такий спосіб захищений від інших процесів. Більш того, оскільки сервери виконуються в просторі користувача, вони не мають безпосереднього доступу до апаратури і не можуть модифікувати пам'ять, у якій зберігається керуюча програма. І якщо окремий сервер може потерпіти крах, то він може бути перезапущений без зупинки чи ушкодження іншої частини ОС.

Існує тенденція руху від монолітних систем в бік підходу з використанням невеликих ядер. Саме такий підхід використовується компанією QNX Software, яка протягом декількох років поставляє операційні системи на основі мікроядра QNX на ринок систем реального часу.

4. Обєктно-орієнтовні ОСРЧ

Хоча технологія мікроядер і заклала основи модульних систем, здатних розвиватися регулярним чином, вона не змогла повною мірою забезпечити можливості розширення систем. В даний час цій меті найбільшою мірою відповідає

об'єктно-орієнтований підхід, при якому кожен програмний компонент є функціонально ізольованим від інших.

Основним поняттям цього підходу є "об'єкт". Об'єкт - це одиниця програм і даних, яка взаємодіє з іншими об'єктам за допомогою прийому і передачі повідомлень. Об'єкт може бути представленням як деяких конкретних речей - прикладної програми чи документа, так і деяких абстракцій - процесу, події.

Програми (функції) об'єкта визначають перелік дій, що можуть бути виконані над даними цього об'єкта. Об'єкт-клієнт може звернутися до іншого об'єкту, пославши повідомлення з запитом на виконання якої-небудь функції об'єкта-сервера.

Об'єкти можуть описувати сутності, які вони представляють, з різним ступенем деталізації. Для забезпечення наслідуваності при переході до більш детального опису розроблювачам пропонується механізм успадкування властивостей вже існуючих об'єктів, тобто механізм, що дозволяє породжувати більш конкретні об'єкти з більш загальних. Механізм успадкування дозволяє створити ієрархію об'єктів, у якій кожен об'єкт більш низького рівня здобуває усі властивості свого предка.

Внутрішня структура даних об'єкта схована від спостереження. Не можна довільно змінювати дані об'єкта. Для того, щоб одержати дані з об'єкта чи помістити дані в об'єкт, необхідно викликати відповідні об'єктні функції. Це ізолює об'єкт від того коду, який використовує його. Розроблювач може звертатися до функцій інших об'єктів, чи будувати нові об'єкти шляхом успадкування властивостей інших об'єктів, нічого не знаючи про те, як вони сконструйовані. Ця властивість називається інкапсуляцією. Таким чином, об'єкт являється для зовнішнього світу у виді "чорного ящика" з добре визначеним інтерфейсом. З погляду розроблювача, що використовує об'єкт, поки зовнішня реакція об'єкта залишається без змін, не мають значення ніякі зміни у внутрішній реалізації.

Це дає можливість легко заміняти одну реалізацію об'єкта іншою, наприклад, у випадку зміни апаратних засобів; при цьому складне програмне оточення, у якому знаходяться змінні об'єкти, не зазнає ніяких змін. З іншого боку, здатність об'єктів

з'являтися у виді "чорного ящика" дозволяє пакувати в них і представляти у виді об'єктів вже існуючі додатки, нічого в них не змінюючи.

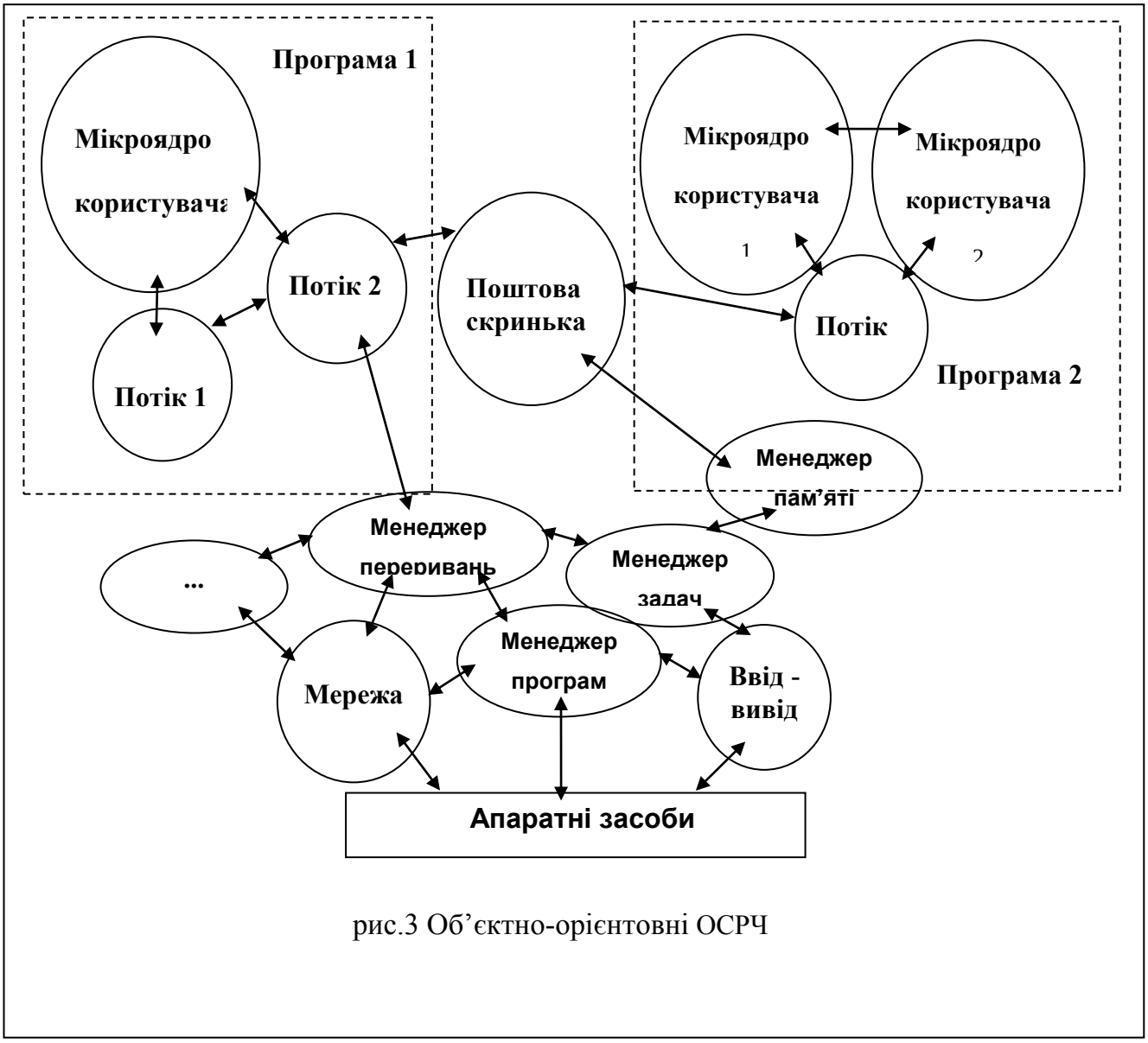

рис.3 Об'єктно-орієнтовні ОСРЧ

Використання обєктно-орієнтовного підходу особливо ефективно при створенні активно розвиваючого програмного забезпечення, наприклад, при розробці додатків, призначених для виконання на різних апаратних платформах.

Цілком обєктно-орієнтовні операційні системи дуже привабливі для системних програмістів, тому що, використовуючи об'єкти системного рівня, програмісти зможуть "входити" всередину операційних систем для пристосування їх до своїх потреб, не порушуючи цілісність системи.

Але особливо великі перспективи має цей підхід у реалізації розподілених обчислювальних середовищ. У той час, як різні пакети, що працюють у даний

момент в мережі, являють собою статично зв'язані набори програм, у майбутньому, з використанням обєктно-орієнтовного підходу, вони можуть перетворитися в єдину сукупність об'єктів, що динамічно зв'язуються, де кожен об'єкт оперативно встановлює і розриває зв'язки з іншими об'єктами для виконання актуальних у даний момент задач. Додатки, створені для такого сіткового середовища, що базується на об'єктах, можуть виконуватися, динамічно звертаючись до безлічі об'єктів, незалежно від їхнього місцезнаходження в мережі і незалежно від їхнього операційного середовища.

Оскільки будь-який обєктно-орієнтовний додаток являє собою набір об'єктів, розроблювачу бажано мати стандартні засоби для керування об'єктами й організації їхньої взаємодії. При використанні і розробці обєктно-орієнтовних додатків у неоднорідних розподілених середовищах, потрібні також засоби, що спрощують доступ до об'єктів мережі. При виникненні запиту до будь-якого об'єкта розподіленого середовища, незалежно від того, знаходиться необхідний об'єкт на тому ж комп'ютері чи на одному з віддалених, прозорим чином повинен бути виконаний пошук об'єкта, передача йому повідомлення, і повернення відповіді. Для забезпечення прозорого виявлення об'єктів, усі вони повинні бути забезпечені посиланнями, що зберігаються в каталогах. Звідси випливає дуже складна проблема організації служби каталогів, що дозволяє програмістам іменувати і шукати об'єкти в мережі, яка може бути розкидана по усьому світі.

Однак, незважаючи на згадані складності і проблеми, обєктно-орієнтовний підхід є однією із самих перспективних тенденцій у конструюванні програмного забезпечення.

Лекція 4

Процеси в ОСРЧ

1. Поняття про процеси.

Розглянемо ОС як безліч дій, кожна з який являє собою одну із функцій, таких як планування або входи/виходи. Кожна дія складається з виконання однієї чи декількох програм і буде викликатися тільки тоді, коли відповідна функція буде потрібна. Слово ”процес” використовується для опису такого роду дій. Поняття "процес" є, ймовірно, найбільш часто вживаним і найменш точно визначеним терміном з області операційних систем. Під процесом розуміють послідовність дій, представлених виконанням послідовності інструкцій (програм), набір результатів яких представляє деяку системну функцію або набір функцій користувача.

Говорячи нестрого, процес можна представити як групу комірок пам'яті, вміст яких змінюється за визначеними правилами. У комп'ютері ці правила зазвичай описуються програмою. Процес може включати виконання більш, ніж однієї програми; і навпаки, програма чи підпрограми можуть бути включені в більш ніж один процес. Наприклад, процедура додавання елементу до списку могла б бути використана в будь-якому процесі, зайнятим маніпулюванням над чергами. Отже, знання того, що якась програма виконується в даний момент, не говорить нам багато про дії чи реалізовану функцію. Значним, з цього погляду, є те, що концепція процесу є більш корисна, чим концепція програми, коли ми говоримо про ОС.

Хорнінг і Ренделл запропонували формальну модель процесу. Нехай

$$X=\{x_0, x_1,\ldots,x_n,\ldots\}$$

-це набір (може бути нескінченний) змінних, що характеризують стан, який називається набором змінних стану. Стан описується заданням значень всіх елементів, що входять у набір змінних стану. Простір станів для даного набору змінних стану – це безліч значень, які може приймати набір змінних. Дія – це присвоєння значень деяких із перемінних цього набору. Послідовність станів, що належать простору станів, називається роботою. Один зі способів виконати роботу полягає в послідовному застосуванні різних дій. Функція дії – це функція відображення станів у дії. Функція дії може також породжувати роботу по заданому

початкового стану. Вона переробляє нові стани до нескінченності, одержуючи таким способом послідовність станів. На змістовному рівні: набір перемінних - це пам'ять, стан – це вміст пам'яті, а функція дії – це програма. Тепер можна визначити процес, як простір станів, функцію дії в цьому просторі і деякі особливі елементи простору станів, що названі початковим станом.

Процес здатний виконуватись за допомогою агента, що виконує відповідну програму. Агентом служить процесор. Процесор - це щось, що виконує інструкції; залежачи від природи інструкцій, процесор може бути використаний в обладнанні окремо, чи в комбінації з обладнанням і ПЗ. Перетворення будь-якого комп'ютера у віртуальну машину може розглядатися як комбінація комп'ютерного обладнання й ОС з метою надання процесора для запуску процесів користувачів (тобто процесор, здатний виконувати інструкції користувача).

2. Конкурентність і детермінованість ОС у термінах процесів.

Концепції процесу і процесора можуть легко бути використані при інтерпретації конкурентності і недетермінованості, які являються двома характеристиками ОС.

Конкуренцією може вважатися активація декількох процесів одночасно. Якщо існує один процесор, то процеси конкурують при переключенні процесора з одного процесу на інший. Якщо переключення виконується за прийнятно малі часові інтервали, то система створює ілюзію конкуренції (якщо взяти до уваги великий проміжок часу). Вдавана конкуренція проявляється в ефекті, що досягається чергуванням процесів на єдиному процесорі. В цьому випадку (коли виконується декілька процесів) створюється враження, що процесор виконує всі процеси одночасно, хоча насправді відбувається переключення процесора між виконанням процесів. При цьому процесор кожний раз повинен помічати те місце на якому він зупинився в попередньому завданні, щоб продовжити його з цього місця пізніше.

Дійсна конкуренція може існувати між різними процесорами, що виконують різні процеси. Вона проявляється тільки тоді, коли число процесорів рівне числу процесів, які потрібно виконати.

Конкуруюче виконання (чи паралельне виконання) позначає моментальний стан системи, у якому можуть спостерігатися декілька процесів будь-де між їх початковою і кінцевою точками. Зрозуміло, що це визначення включає як дійсну, так і вдавану конкуренцію.

Недетермінованість - це друга характеристика системи, що легко описується в термінах процесів. Якщо ми вважаємо процес послідовністю дій, яка може бути перервана між кроками, то недетермінованість відбивається в непередбаченому порядку можливих переривань і, отже, у визначеному порядку, у якій ця послідовність дій протікає. Переривання процесу повинне супроводжуватися записом інформації, яка буде необхідна для його поновлення. Записувана інформація та ж, яка потрібно при переключенні між процесами при досягненні конкурентності. Насправді, переривання процесу може вважатися переключенням, викликаним непередбаченою подією не залежним від процесора і процесу.

Отже, процес - це послідовність дій і тому він - динамічний, у той час, як програма - послідовність інструкцій і тому вона статична. Недетермінованість і конкурентність можуть бути описані в термінах переривання процесу між діями і переключенні процесорів між процесами. Щоб здійснювати переривання чи переключення, вся інформація про процес повинна бути збережена для подальшого його поновлення.

3. Комунікація процесів.

Процеси в комп'ютерній системі, звичайно, не діють в ізоляції. З одного боку, вони повинні кооперуватися для досягнення бажаної мети запущеної користувачем задачі; з іншого боку, вони знаходяться в змаганні за використання обмежених ресурсів, таких как процесор, пам'ять чи файли. Ці два елементи: - кооперація і змагання потребують деякої форми комунікації між процесами. Основні аспекти комунікації процесів можуть бути категоризовані наступним чином: взаємні виключення, синхронізація, блокування.

3.1. Взаємні виключення.

Системні ресурси можуть класифікуватися як поділяючі, розуміючи під цим, що вони можуть бути використані декількома процесами одночасно, і неподіляючі, розуміючи при цьому, що їхнє використання обмежене одним процесом в даний час.

Неподіл ресурсів відбувається по одній з наступних причин:

1. Фізична природа ресурсу робить неможливим його поділ. Типовий приклад - принтер, неможливо переключити його між друком аркушів різних процесів.

2. Ресурс такий, що якщо він використовується декількома процесами одночасно, дія одного процесу може взаємодіяти з діями іншого. Найбільше загальним прикладом є розподіл пам'яті, що містить змінні, доступні для більш, ніж одного процесу: якщо один процес перевіряє перемінні в той час, як інший модифікує їх, то результат буде непередбачений і звичайно згубний. Наприклад, припустимо в системі резервування авіаквитків наявність вільного місця представлено вмістом деякого місця в пам'яті. У цьому випадку, якщо це місце доступне більш, ніж одному процесу в один і той самий час, то стало б можливо двом агентам резервувати місце одночасне. Тому, не поділяючі ресурси включають в себе більшість периферійних пристроїв, доступних для запису файлів і області даних, що є предметом модифікації. Поділювані ресурси включають CPU, файли тільки для читання, області пам'яті, що містять процедури і дані, захищені від модифікації.

Проблема взаємного виключення полягає в тому, щоб забезпечити доступ до не поділюваних ресурсів тільки одному процесу в деякий момент часу.

3.2. Синхронізація

Швидкість одного процесу стосовно іншого непередбачена, тому що вона залежить від частоти переривань кожного процесу і від того, як довго кожен процес займає процесор. Процеси запускаються асинхронно по відношенню один до одного. Однак, для досягнення успішної взаємодії процеси повинні синхронізувати свої дії. Існує механізм, за допомогою якого процес не може продовжуватися доти, поки інший процес на завершить вихідну дію. Наприклад, процес, що планує

проходження завдань користувача не може продовжуватися доти, поки вхідний процес не прочитає принаймні одне завдання в машину. Подібним чином, обчислювальний процес, що робить вивід, не здатний продовжуватися доти, поки його попередні виходи не будуть виведені. ОС відповідальна за забезпечення механізмів, за допомогою яких досягається синхронізація.

3.3. Блокування

Коли декілька процесів змагаються за використання ресурсів, може виникнути ситуація, коли ні один із процесів не може продовжитися через те, що ресурси, необхідні одному, зайняті іншим і навпаки. Ця ситуація називається блокуванням чи смертельними обіймами і є аналогом транспортної пробки, у якій два протилежних потоки намагаються перетнути шлях один одного і стають цілком нерухомими, тому що кожен потік займає дорожній простір, необхідний іншому. Запобіганню блокуванню, чи обмеження їхньої дії, очевидно, також відповідальна ОС.

4. Семафори

Найбільш важливим внеском у комуникацію процесів була введена (Dijkstra, 1965) концепція семафорів і примітивних операторів **wait** і **signal**, що впливають на них. (**wait** і **signal** часто мають позначення P і V, що є початковими буквами відповідних датських букв.)

Семафор – це перемінна цілого типу, значення якої не негативне і змінювати його можуть тільки оператори **wait** і **signal**.

Signal (s) - значення семафора s збільшується на одиницю. Операція є неподільної.

Wait (s) - значення семафора s зменшується на 1 за умови, що результат буде ненегативний. Операція є неподільної.

Неподільність операції означає, що в кожен момент часу тільки один процес може виконувати операцію **wait** або **signal** над даним семафором. Операція **wait** приводить до потенційної затримку у випадку, коли вона діє на семафор зі значенням '0'; процес, що виконує цю операцію, може продовжитися тільки тоді,

коли деякий інший процес збільшить значення семафора на '1' за допомогою операції **signal**. Неподільність операції означає, що якщо кілька процесів затримуються, то тільки один з них може успішно завершити операцію, якщо значення семафора стало позитивним. Ніяких припущень не робиться про те, який це буде процес. Дія операторів **wait** і **signal** може бути узагальнена в такий спосіб:

wait (s) : when s>0 do decrement s

signal (s): increment s

де s - будь-який семафор.

З цих визначень можна побачити, что кожна операція **signal** збільшує значення семафора на 1 і кожна наступна успішна операція **wait** зменшує це значення на 1. Отже значення семафора залежить від співвідношення числа операторів **wait** і **signal**.

$$\mathbf{val}\ (s)=\mathbf{C}\ (s)+\mathbf{ns}\ (s)-\mathbf{nw}\ (s), \qquad (1)$$

де

val (s) - значення семафора s;

C (s) - початкове значення семафора s,

ns (s) - число операторів **signal** на семафорі,

nw (s) - число успішно виконаних операторів **wait** на семафорі.

Але по визначенню **val** (s)≥0, отже, отримаємо співвідношення:

$$\mathbf{nw}\ (s)\leq \mathbf{ns}\ (s)+\mathbf{C}\ (s), \qquad (2)$$

у якому рівність виконується тоді і тільки тоді, коли **val** (s)=0.

Відношення є інваріантним щодо операторів **wait** і **signal**, тобто воно вірне при виконанні багатьох операторів. Використовуємо семафори **wait** і **signal** для рішення перерахованих вище задач

4.1. Взаємні виключення.

Неподільні ресурси, чи то периферія, чи файли даних в пам'яті, можуть бути захищені від одночасного доступу шляхом використання попередніх заходів від

одночасного виконання процесами ділянок програм. Ці ділянки програм називаються критичними секціями і взаємні виключення у використанні ресурсів еквівалентні взаємному виключенню виконання цих_критичних секцій. Тобто критична секція є частиною процесу, що повинна виконуватися без переривання з боку інших процесів. Таким чином, відбувається виключення одного процесу іншим. Наприклад, процеси можуть бути взаємовиключаючими від доступу до таблиці даних. Якщо всі підпрограми, що читають чи обновляють таблицю написані як критичні секції так, то тільки одна б могла бути виконана одночасно.

Семафор, максимальне значення якого дорівнює одиниці, називається двійковим семафором. Двійковий семафор організовує взаємне виключення заключенням критичних ділянок у "дужки", роль яких відіграють оператори **wait** і **signal**. Таким чином кожна критична секція програмується в так:

wait (s);

критична секція;

signal (s);

де s - ім'я семафора;

Якщо початкове значення семафора дорівнює одиниці, то взаємне виключення гарантовано, оскільки процес може виконати **wait** (s) до того, як інший виконає **signal** (s). Крім того, процес без необхідності не перекриває входи всередину своєї критичної секції; вхід затримується тільки тоді, коли деякий інший процес вже всередині своєї власної секції. Процес не дозволяє вхід, тільки, якщо значення mutex=0. Це означать, що:

nw (s)=**ns** (s)+1.

Іншими словами, число успішних операторів **wait** на семафорі s перевищує число сигналів на 1, що означає, что деякий процес знаходиться всередині його критичної секції.

Висновок полягає в наступному: в кожний момент часу процес, що бажає одержати доступ до подільного ресурсу, повинен зробити це за допомогою критичної секції, захищеної семафором, як це було показано вище.

Як приклад, припустимо, что ОС містить два процеси А і В, що відповідно додають і видаляють елемент із черги. Щоб покажчики черги не стали

переплутаними необхідно обмежити доступ до черги одному процесу одночасно. Таким чином операції додавання і видалення елементів можна вказати як критичні секції.

Program for Process A	Program for Process B
…	…
wait (s);	**wait** (s);
додати елемент із черги	видалити елемент із черги
signal (s);	**signal** (s);
…	…

Це просте рішення, яке використовує двійкові семафори, можна застосовувати також і у випадку великого числа процесів. В цьому головна перевага підходу з застосуванням семафора.

4.2. Синхронізація.

Найпростішою формою синхронізації є те, що процес А не може продовжитися з точки L1 доти, поки інший процес В не досягне точки L2. Приклади такої синхронізації виникають щораз, коли А вимагає в т. L1 інформацію, яку представляє В, коли досягне L2. Синхронізація програмувальна в такий спосіб:

Program for process A	Program for proces B
…	…
L1: **wait** (s);	L2: **signal** (s);
…	…

де s - семафор з початковим значенням 0.

З приведених вище фрагментів програм ясно, что А не може продовжуватися за L1 доти, поки В не виконає оператор **signal** у т. L2. (Звичайно, якщо В виконає **signal** до того, як А досягне L1, то А не завершиться зовсім). Одержимо знову співвідношення (2), яке буде мати наступний вигляд:

nw (s)≤ **ns** (s) (3)

звідки випливає, що A не може пройти L1 до того, як B не пройде L2.

Приведений приклад асиметричний тому що A регулюється B, і не навпаки. Випадок, при якому процеси регулюють прогрес один одного ілюструються класичною проблемою постачальника і споживача. Тому що ця проблема типова для багатьох внутрішніх комунікаційних процесів, ми опишемо її більш детально.

Безліч процесів постачальників і безліч процесів споживачів зв'язуються за допомогою буфера в який виробники поміщають елементи і з якого споживачі їх витягають. Виробники весь час повторюють цикл “зробити елемент - помістити в буфер”, а споживачі повторюють подібний цикл “витягти елемент - спожити його”. В обчислювальному процесі це може виглядати як периміщення одним процесом своїх результатів в буфер і витяг та друк цих рядків іншим. Буфер має обмежену ємність і здатний зберігати ***N*** елементів еквівалентного розміру. Синхронізація вимагає дотримання двох умов:

1. Виробники не можуть покласти елементи в переповнений буфер.

2. Споживачі не можуть витягти елементи, якщо буфер порожній.

Іншими словами, якщо число поміщених у буфер елементів є ***d***, а число витягнутих - ***e***, то

$$0 \le d - e \le N.$$

Більш того, буфер повинен бути захищений від одночасного доступу процесів користувачів; які б не були дії одного процесу (наприклад відновлення покажчиків), вони впливає на дії іншого. Отже, периміщення і витяг елементів повинні бути закодовані як критичні секції.

Вирішимо проблему такий чином:

Програма виробника	Програма споживача
begin	**Begin**
утворити елемент;	**wait** (item available);
wait (space available);	**wait** (buffer manipulation);
wait (buffer manapulation);	Витягнути елемент з буфера;

помістити елемент в буфер;	**signal** (buffer manipulation);
signal (buffer manipulation);	**signal** (space available);
signal (item available);	Використати елемент;
end;	**end**;

Синхронізація досягається через семафори space available (доступне місце) і item available (доступний елемент), з початковими значеннями ***N*** і 0 відповідають. Взаємні виключення процесів від доступу до буфера здійснюється за допомогою семафора buffer manipulation (буферне маніпулювання) з початковим значенням 1.

Тепер покажемо, що це рішення задовольняє співвідношенням

nw (space available) ≤ **ns** (space available)+***N*** (4)

і

nw (item available) ≤ **ns** (item available) (5)

З порядку операцій програми-виробника випливає, що

ns (item available)≤ ***d*** ≤ **nw** (space available) (6)

З порядку операцій програми споживача випливає, що

ns (space available) ≤ ***e*** ≤ **nw** (item_available) (7)

Отже, з (6) випливає

d ≤ **nw** (space available) ≤ **ns** (space available)+***N***<=***e***+***N***

Аналогічно, з (7) отримуємо:

e ≤ **nw** (item available) ≤ **ns** (item available) ≤ ***d***

Комбінація цих двох результатів дає співвідношення

e ≤ ***d*** ≤ ***e***+***N***,

що показує, що співвідношення (3) задовольняється, як це і потрібно.

Рішення проблеми виробник-споживач може бути використано як посібник у будь-якій іншій ситуації, коли один процес посилає інформацію іншому.

4.3. Блокування.

Як підкреслювалося раніше, блокування може статися щораз, як процеси конкурують за русурси. Відзначимо, что блокування може також статися внаслідок очікування процесами закінчення деяких дій один-одного. Як приклад розглянемо два процеси A і B, що оперують семафорами X і Y, як це приведено нижче.

Process A	Process B
…	…
wait (x);	**wait** (y);
…	…
wait (y);	**wait** (x);
…	…

Якщо початкові значення X і Y дорівнюють 1, то кожен процес може завершити свій **wait**-оператор, зменшуючи значення семафора до 0. Ясно, що жоден із процесів не може дійти до свого наступного **wait**-оператора і відбувається блокування.

Ситуація фактично аналогічна тій, у якій блокування виникає в конкуренції за необхідні ресурси. Якщо семафор розуміти як ресурс і оператори wait і **signal** вважати як пред'являчий і забираючий його, то блокування виникає через взаємну потребу ресурсу процесами A і B. Роздільність семафора визначається його початковими значенням n, якщо n>1. Якщо n = 1 (як це було вище), те семафор нероздільний.

5. Посилка повідомлень

Деякі системи (особливо клієнт-сервер) реалізують зв'язок між процесами і синхронізацію за допомогою посилки повідомлень між процесами. Посилка повідомлень між процесами особливо важлива, тому що немає особливої потреби щоб зв’язані процеси запускатися на одному й тому самому процесорі чи комп'ютері, за умови, якщо звичайно існує апаратний зв'язок між ними.

ОС надає процедури, що дозволяють процесам посилати і приймати повідомлення. Наприклад такими процедурами можуть бути

send_message (distsnation, message); і

receive_message (source, message); де

distination - ідентифікують процес, якому посилається повідомлення;

source - ідентифікує процес, від якої потрібно одержати повідомлення;

message - посилання на область пам'яті, що містить повідомлення.

Кожна з цих процедур у загальному має дві форми: що блокує і неблокує. Блокуюча операція **send** очікує до прийняття отримуючим процесом повідомлення. Неблокуюча операція **send** просто поміщає повідомлення в деяку форму черги і дозволяє відправнику продовжувати виконання.

Блокуюча операція **recieve** очікує надходження повідомлення (крім випадку, коли є одне повідомлення, що очікує прийняття). Неблокуюча операція **recieve** приймає повідомлення, якщо воно очікує, в противному випадку вона встановлює деякий індикатор, що повідомлення немає в наявності. Обробка неблокуючих повідомлень вимагає, щоб ОС надавала деякий вид буферної області всередині якої повідомлення, що очікують, можуть копіюватися і зберігатися, поки не будуть прийняті. Це збільшує для передаючого процесу витрати деякої системної пам'яті.

Ідентифікація процесів також має свої варіанти.

Процес-призначення може бути спеціалізований особливим ідентифікатором, що означає, що повідомлення послані групі процесів чи навіть усім процесам, це так звані широкомовні_повідомлення. Схожим чином, процес-відправник може може бути спеціалізований як будь-який процес з групи чи як будь-який процес у системі; це могло б бути використане там, де процес-сервер готується прийняти роботу від будь-якого іншого процесу в системі. Зазвичай повідомлення короткі (декілька байт чи десятків байт); простіше, якщо вони фіксованого розміру, тому що це спрощує керування буферним простором. Повідомлення від процесів-клієнтів до процесів-серверів зазвичай містять індикатор необхідного сервісу разом з потрібними параметрами. Будь-яка подальша інформація, необхідна серверу, в загальному випадку повідомляється по зазначеному посиланню до області поділюваної пам'яті, де ця інформація знаходиться.

Лекція 5

Процеси і потоки

1. Архітектура QNX.

Архітектура — це то, чим QNX, незважаючи на велику зовнішню подібність, відрізняється від операційних систем сімейства UNIX. Саме архітектура робить QNX операційною системою *жорсткого* реального часу.

Центральним поняттям у QNX є *мікроядро*. Мікроядро (саме його і називають Neutrino) майже нічого саме не робить, а є свого роду комутуючим елементом, до якого за допомогою додаткових програмних модулів додається та чи інша функціональність. Крім мікроядра в ОСРВ QNX є ще один важливий компонент — *адміністратор процесів.* Мікроядро Neutrino скомпоновано з адміністратором процесів у єдиний модуль procnto — головний (і єдиний безумовно необхідний) компонент QNX. Якщо нам треба, щоб система реально робила якусь роботу, ми повинні запустити процес, що виконує цю роботу. Програми, що реалізують сервісні функції, називають *адміністраторами ресурсів.* Є адміністратори ресурсів, що забезпечують доступ до дисків, мережі і т.д. Усі ці програми зв'язані в одне ціле з мікроядром і злагоджено взаємодіють за допомогою механізму повідомлень.

Варто відмітити, що існують різні додаткові варіанти модуля procnto (не говорячи про версії для різних процесорів):

- procnto-smp — варіант модуля procnto з підтримкою симетричної багатопроцесорності;
- procnto-instr - варіант модуля procnto, обладнаний засобами трасування подій;
- procnto-smp-instr — самі догадайтеся, для чого.

Насправді функціональність ОС може розширюватися не тільки за допомогою процесів, але і за допомогою бібліотек, що динамічно приєднуються, (Dynamic Link Library, DLL). Правільніше буде сказати, що як функціональність ОС розширюється за допомогою процесів, так функціональність процесів розширюється за допомогою

DLL. Більш того, адміністратор ресурсів може бути реалізований чи як програма, чи як DLL.

Таким чином, у загальному випадку QNX складається:

- з мікроядра Neutrino;
- адміністратора процесів;
- адміністраторів ресурсів;
- прикладних програм.

Що ж таке процес? *Процес* (process) — це програма, яка виконується. Процес (чи “задача”) включає код і дані програми, а також різну додаткову інформацію – перемінні системного оточення і т.п.

Крім процесу важливим поняттям є “потік керування” (thread, він же просто “потік”; раніше його іноді називали "ниткою"). *Потік керування*— це фрагмент процесу, що містить неперервну послідовність команд, що можуть виконуватися паралельно з іншими потоками того ж чи інших процесів.

Процес є, по суті, контейнером потоків і містить мінімум один потік.

Для чого взагалі потрібні потоки? Вони дуже корисні в ряді випадків, тому підтримка потоків — обов'язкова властивість POSIX-сумісних ОС. Зазвичай потоки використовують:

- для распараллеливания задачі на многопроцессорных ЕОМ;
- для більш ефективного використання процесора (наприклад, коли один потік очікує вводу користувача , інший може виконувати розрахунки);
- для полегшення спільного використання даних (усі потоки процесу мають вільний доступ до даних процесу).

2. Механізми мікроядра.

Отже, мікроядро Neutrino — головний і обов'язковий компонент ОСРЧ QNX. Воно виконує наступні функції:

- створення і знищення потоків;
- диспетчеризація потоків;
- синхронізація потоків;

- механізми IPC (Inter Process Communication);
- підтримка механізму обробки переривань;
- підтримка годин, таймерів і таймаутов.

Більше Neutrino не робить нічого. Вся інша функціональність QNX забезпечується адміністраторами ресурсів. Зверніть увагу на те, що мікроядро Neutrino працює тільки з потоками і нічого не знає про процеси. За процеси відповідає *адміністратор процесів.*

На етапі виконання, потік може знаходитися в одному з трьох станів: виконання на процесорі (RUNNING), чекання процесора, чи готовність до виконання (READY) і блокування в чеканні звільнення деякого ресурсу (назва блокованого стану залежить від того, у чеканні якого ресурсу заблокований потік). Є ще стан DEAD (у UNIX його називають ZOMBIE) — коли фізично потік знищений, але адміністратор процесів ще зберігає деякі структури даних з інформацією про нього, щоб передати батьківському потоку код завершення.

Диспетчеризація потоків.

QNX — багатозадачна ОС. Це значить, що в системі може існувати досить багато процесів (і потоків). При чому деякі з них можуть одночасно бути в стані готовності до виконання. Як визначити, якому з цих потоків надати вільний процесор? Для цього мікроядро використовує пріоритет, призначений кожному потоку. Усього в QNX Neutrino версії 6.3 мається 256 рівнів пріоритету. Найнижчий пріоритет — 0, найвищий — 255. Нульовий пріоритет має спеціальний потік адміністратора процесів за назвою idle (що в перекладі з англійського означає "лінивець"), на технічному жаргоні його називають "холодильником". Цей потік завжди знаходиться в стані готовності до виконання READY. Не здумайте задати своєму потоку нульовий пріоритет — він ніколи не одержить процесор.

Диспетчеризація виконується мікроядром у трьох випадках:

- потік, що виконується на процесорі, перейшов у блокований стан;

- потік з більш високим, ніж у потоку, що виконується, пріоритетом перейшов у стан готовності, тобто відбувається витіснення потоку (цю властивість ОС називають що *витісняючою багатозадачністю)*;
- потік, що виконується, сам передає право виконання на процесорі іншому потоку (викликає *функцію sched_yield()).*

Але як бути в тому випадку, коли декілька готових до виконання процесів мають рівні пріоритети? Щоб запобігти цій (типовій, треба сказати) ситуації, використовують *дисципліни диспетчеризації.* Мікроядро Neutrino дозволяє задавати для кожного потоку одну з наступних дисциплін диспетчеризації:

- FIFO (First In First Out — "перший увійшов — перший вийшов"). Якщо потоку призначена дисципліна диспетчеризації FIFO, то інший потік з таким же пріоритетом одержить керування, тільки якщо потік, що виконується, заблокується чи сам уступить право виконання;
- "карусельна" (Round Robin) диспетчеризація — потік виконується протягом *кванту часу* і передає керування наступному потоку з таким же пріоритетом. У QNX 6.3 квант часу (timeslice) за замовчуванням дорівнює 4 мс (для процесорів з частотою вище 40 МГц);
- Спорадична диспетчеризація (тобто завдання, наступний термін надходження яких може наступити не раніше деякого часу після їх попереднього надходження) – призначена для встановлення ліміту використання потоком процесора протягом визначеного періоду часу. Цей механізм замінив адаптивну диспетчеризацію, яка була в колишніх версіях QNX і введений у порядку експерименту.

При спорадичній дисципліні потоку задається кілька параметрів:

- нормальний пріоритет (N);
- нижній пріоритет (L);
- початковий бюджет (C);
- період відновлення (T);
- максимальна кількість пропусків відновлення.

Коли потік переходить у стан READY, його пріоритет має значення N протягом інтервалу часу C, після чого пріоритет потоку знижується до значення L. Але час перебування потоку в стані з пріоритетом L обмежено. Через інтервал часу T (вважаючи разом із C) пріоритет потоку знову стане рівним N. Оскільки потік при пріоритеті L може взагалі не одержати керування, задається обмеження максимальної кількості пропусків відновлення при досягненні якого, потоку буде примусово повернутий пріоритет N. Таким чином, спорадична диспетчеризація гарантує, что потік не забере більше C / T процесорного часу (якщо не він один має пріоритет, рівний N).

В додаток до дисциплін диспетчеризації, Neutrino підтримує механізм клиент-керованих пріоритетів. Це потрібно для серверного потоку, що обробляє запити потоків-клієнтів. Ідея полягає в тому, що пріоритет сервера встановлюється рівним максимальному з пріоритетів клієнтів, заблокованих у чеканні звільнення сервера. Дійсно, було б не дуже добре, якби потік із пріоритетом 100 чекав, поки сервер закінчить обробку попереднього запиту з пріоритетом, наприклад, 5.

Синхронізація потоків.

Для синхронізації потоків у QNX застосовують кілька способів. Причому тільки два з них (mutex і condvar) реалізовані в мікроядрі, інші — надбудови над цими двома:

- взаємовиключаюче блокування (Mutual exclusion lock -mutex, "мутекс") — цей механізм забезпечує винятковий доступ потоків до розмежованих даних. Тільки один потік в один момент часу може володіти мутексом. Якщо інші потоки спробують захопити мутекс, вони стають мутекс-заблокованими;
- умовна перемінна (condition variable, чи condvar) — призначена для блокування потоку доти, поки виконується деяка умова. Ця умова може бути складною. Умовні перемінні завжди використовуються з мутекс-блокуванням для визначення моменту зняття мутекс-блокування;
- бар'єр – встановлює місце для декількох взаємодіючих потоків, на якому вони повинні зупинитися і дочекатися "відсталих" потоків. Як тільки всі потоки з

контрольованої групи досягли бар'єра, вони розблокуються і можуть продовжити виконання;

- очікуюче блокування – спрощена форма спільного використання умовної перемінної з мутексом. На відміну від прямого застосування mutex + condvar має деякі обмеження;
- блокування читання/запису (rwlock) - проста й зручна у використанні "надбудова" над умовними перемінним і мутексами;
- семафор – це, можна сказати, мутекс із лічильником. Точніше мутекс є семафором з лічильником, рівним одиниці. Семафор можуть захопити кілька потоків (їхнє число рівне значенню лічильника), всі інші потоки, що намагаються захопити семафор, будуть заблоковані. Семафори бувають іменовані і неіменовані. Іменований (більш повільний) варіант семафора можна використовувати для синхронізації потоків, що виконуються в різних процесах і навіть різних вузлах мережі.

Більшість цих механізмів працює тільки в межах одного процесу, але це обмеження можна уникнути шляхом розділення пам'яті.

Крім перерахованих способів синхронізацію можна здійснити за допомогою FIFO-диспетчеризації, QNX-повідомлень і атомарних операцій.

3. Механізми IPC.

У мікроядрі Neutrino реалізована підтримка декілька механізмів IPC:

- синхронні повідомлення QNX – поряд з архітектурою мікроядра є фундаментальним принципом QNX. Це самий швидкий спосіб обміну даними довільного розміру у QNX, при цьому мікроядру байдуже, між якими потоками відбувається обмін QNX-повідомленнями – усередині процесу, між різними процесами чи навіть між процесами, що працюють на різних вузлах локальної обчислювальної мережі;
- Pulses (цей тип повідомлень називають "імпульсами") — це фіксовані повідомлення, що мають розмір 40 біт (8-бітний код імпульсу і 32 біта даних), що не блокують відправника;

- сигнали POSIX (як прості, так і реального часу).

На прохання замовників фірма QSS у порядку експерименту ввела таке нововведення, як *асинхронні повідомлення.* Цей механізм, як і механізм імпульсів, використовує буфер для збереження повідомлень. Функції асинхронних повідомлень мають такі ж назви, як функції звичайних повідомлень, але з префіксом *asyncmsg_* (наприклад, *asyncmsg_MsgSend()).*

Крім цих механізмів ОСРЧ QNX підтримує несколько додаткових способів IPC:

- черги повідомлень POSIX (реалізовані в адміністраторі черг mgueue);
- поділювана пам'ять (реалізована в адміністраторі процесів);
- іменовані канали (реалізовані в адміністраторі файлової системи QNX4);
- неіменовані канали (реалізовані в адміністраторі каналів pipe).

Сигнали являються не блокуючи відправника методом IPC. Сигнали мають структуру, подібну до імпульсу (1 байт коду + 4 байти даних), і теж можуть ставитися в чергу. Для відправлення процесу сигналу адміністратор може використовувати дві утиліти: стандартну UNIX-утиліту kill і більш гнучку QNX-утиліту slay. За замовчуванням обидві ці утиліти посилають сигнал SIGINTR, при одержанні якого процес звичайно знищується. Строго говорячи, процес може визначити три варіанти поводження при одержанні сигналу:

- *ігнорувати сигнал* — для цього варто задати відповідне значення такому атрибуту потоку, як сигнальна маска (тому звичайно говорять не "ігнорувати", а "маскувати" сигнал). Треба сказати, что три сигнали не можуть бути ігноровані: SIGSTOP, SIGCONT і SIGTERM;
- *використовувати оброблювач за замовчуванням* – тобто ничего не робити. Зазвичай оброблювачем за замовчуванням для сигналів є знищення процесу;
- *зареєструвати власний оброблювач* — тобто власну функцію, що буде викликана при одержанні сигналу.

Потік може викликати функцію чекання того чи іншого сигналу (сигналів). При цьому потік стане SIGNAL-блокованим.

Специфікація POSIX визначає порядок обробки сигналів тільки для процесів. Тому при роботі з багато потоковими процесами в QNX необхідно враховувати наступні правила:

- Дія сигналу поширюється на весь процес.
- Сигнальна маска поширюється тільки на потік.
- Неігнорований сигнал, призначений для якого-небудь потоку, доставляється тільки цьому потоку.
- Неігнорований сигнал, призначений для процесу, передається першому не SIGNAL-блокованому потоку.

4. Адміністратор процесів QNX.

До цього часу ми говорили, про різні механізми QNX, про взаємодію потоків, синхронізації потоків і т.п., навмисно намагаючись уникати при цьому слова "процес". І робили ми це зі зрозумілої причини – мікроядро Neutrino поняття не має ні про які процеси. Воно знає тільки потоки і знає, як з ними поводитися. Але повнофункціональну POSIX-систему неможливо представити без процесів, що працюють в ізольованих адресних просторах. Так: підтримку процесів у QNX забезпечує *адміністратор процесів.* А оскільки керування процесами і пам'яттю є дуже важливою і критичною функцією операційної системи, адміністратор процесів скомпонований з мікроядром Neutrino у єдиний програмний модуль procnto.

Отже, розглянемо функції, що виконує адміністратор процесів:

- керування процесами;
- керування механізмами захисту пам'яті;
- підтримка механізму поділюваної пам'яті і IPC на її основі;
- керування простором шляхових імен.

Керування процесами

Згадаємо, що процес є контейнером потоків. Власне, усю реальну роботу роблять саме потоки, тому будь-який процес складається не менше ніж з одного потоку.

Процес має ряд атрибутів:

- ідентифікатор процесу (process ID, PID);
- ідентифікатор батьківського процесу (parent process ID, PPID);
- реальні ідентифікатори власника і групи (UID і GID);
- ефективні ідентифікатори власника і групи (EUID і EGID);
- поточний робочий каталог;
- керуючий термінал;
- маска створення файлів (umask);
- пріоритет;
- дисципліна диспетчеризації.

Життєвий цикл процесу включає чотири етапи:

- створення – процес може бути створений лише іншим (батьківським) процесом, при цьому адміністратор процесів створює в себе необхідні структури даних;
- завантаження коду і даних процесу в ОЗУ;
- виконання потоків;
- завершення.

Завершення процесу відбувається в дві стадій. Першу стадію виконує *потік завершувача* (termination thread) адміністратора процесів. При цьому звільняються ресурси, зв'язані з процесом (сторінки ОЗУ, відкриті файлові дескриптори і т. п). Потік завершувача виконується з ідентифікатором знищуваного процесу.

Друга стадія завершення процесу відбувається усередині адміністратора процесів, при цьому код повернення процесу, що завершується, передається процесу-батьку. Але тут можливі варіанти:

- процес-батько був заблокований у чеканні коду завершення дочірнього процесу. У цьому випадку код повернення відразу буде переданий батьку, батько розблокується і дочірній процес завершиться;
- процес-батько відмовився від одержання коду завершення дочірнього процесу, тобто процес, що завершується, мав прапорець SPAWN_NOZOMBIE. У цьому випадку дочірній процес буде негайно завершений;
- процес-батько не відмовлявся від одержання коду повернення дочірнього процесу, але і не викликав функцію одержання цього коду. У цьому випадку

дочірній процес блокується доти, поки батько не прочитає код завершення, тобто процес, що завершується, стає DEAD-блокованим чи "зомбі".

Керування механізмами захисту пам'яті

Адміністратор процесів QNX забезпечує підтримку повного захисту пам'яті (так названу "віртуальну пам'ять") процесів. Кожному процесу дається 4 Гбайта адресного простору, з них код і дані процесу можуть займати простір від 0 до 3,5 Гбайт. Діапазон адрес від 3,5 до 4 Гбайт належить модулю procnto (ці цифри відносяться до ЕОМ на базі процесора x86, у реалізаціях QNX для інших апаратних платформ ці значення можуть бути іншими). Для відображення віртуальних адрес на фізичну пам'ять використовуються апаратні *блоки керування пам'яттю* (MMU — Memory Management Unit).

Керування простором шляхових імен.

Простір шляхових імен є однією з особливостей ОС QNX. Справа в тім, що керування ресурсами вводу/виводу не вбудовано в мікроядро, а реалізується за допомогою додаткових процесів – адміністраторів ресурсів. Наприклад, записом файлів на диск керує адміністратор файлової системи, відправленням даних по мережі – адміністратор мережі. Як же адміністратори ресурсів в ОС QNX інтегрують свої послуги? Для цього адміністратор процесів надає механізм простору шляхових імен.

Простір шляхових імен являє собою дерево каталогів і файлів, у вершині якого знаходиться кореневий каталог ”/” (root). При запуску кожен адміністратор ресурсів реєструє в адміністратора процесів свою зону відповідальності, чи “префікс” (можна сказати, гілку дерева). Сам модуль procnto при завантаженні реєструє в просторі шляхових імен кілька префіксів:

- / - корінь (root) файлової системи, до якого монтуються всі інші префікси;
- /proc/ - каталог, у який відображається інформація про запущені процеси, представлених їхніми ідентифікаторами (PID);
- /proc/boot/ - каталог, у який у виді "плоскої" файлової системи відображаються файли, що входять до складу завантажувального образу QNX;

- /dev/zero – пристрій, при читанні з якого завжди повертає нуль. Використовується, наприклад, для того, щоб заповнити нулями сторінки пам'яті;
- /dev/mem — пристрій, що представляє усю фізичну пам'ять.

Префікс, до якого адміністратор ресурсів запитує приєднання свого підкаталогу, називається *точкою монтування.* Точки монтування можуть перевантажуватися, тобто до однієї точки можуть підключати свої підкаталоги декілька адміністраторів ресурсів. Крім того, адміністратор процесів дозволяє створювати символічні префікси, тобто реєструвати в просторі шляхових імен посилання на існуючий файл.

Механізм простору шляхових імен дозволяє адміністратору процесів визначати, який з адміністраторів ресурсів повинен виконати запит, що надійшов, на ввід/вивід даних. Алгоритм буде наступним:

1. Процес виконує системний виклик *open()* для того, щоб відкрити файл. При цьому вказується повне (шляхове) ім'я файлу. Бібліотечна функція *open()* шле адміністратору процесів QNX-повідомлення визначеної структури, запитуючи щось вроді: "Який з адміністраторів ресурсів відповідає за цей файл?"
2. Адміністратор процесів співставляє шляхове ім'я файлу з деревом префіксів і повертає функції *open()* повідомлення- відповідь, що містить ідентифікатор адміністратора ресурсів.
3. Функція *open( )* напряму звертається до адміністратора ресурсів із запитом на відкриття файлу.
4. Адміністратор ресурсу створює в себе структуру даних, яка називається *блоком керування відкритим контекстом* (OCB — Open Control Block) і повертає функції *open()* файловий дескриптор.

Тепер прикладний процес може запитувати операції читання/запису, використовуючи отриманий файловий дискриптор. Адміністратор ресурсів, у свою чергу, використовує OCB для того, щоб розрізняти запити різних процесів. OCB включає: файловий дескриптор, унікальний усередині одного процесу; ідентифікатор процесу, унікальний для вузла мережі; інформацію про файл

(наприклад, інформацію про поточний файловий вказівник). Новий ОСВ створюється при кожнім виклику функції *open().*

Поділювана пам'ять і бібліотеки, що динамічно приєднуються.

Механізм поділюваної пам'яті реалізований в адміністраторі процесів і призначений для швидкого обміну великими обсягами даних між процесами. Суть цього механізму полягає в тім, що процес запитує адміністратора процесів про виділення деякого регіону пам'яті. Потім цей регіон може відображатися різними процесами на їх власний адресний простір. Оскільки декілька процесів можуть одержати доступ до однієї ділянки пам'яті, з метою збереження цілісності даних необхідно синхронізувати операції читання/запису з розділюваною пам'яттю. Іноді програмний код, що виконує стандартні операції (яскравий приклад — пересилання повідомлень), доцільно виділити в окремий програмний модуль, доступний для одночасного використання декількома процесами. Такий модуль називають р*озділюваним* об'єктом чи *бібліотекою, що динамічно* приєднується, а операцію "підключення" процесу до розділюваного *об'єкта - динамічним компонуванням.* При використанні розділюваного об'єкта до складу додатка включається не програмний код, а інформація про динамічну бібліотеку.

Яким же чином виконується динамічне компонування? При створенні процесу адміністратор процесів копіює у фізичну пам'ять сегменти програми, що завантажуються, і розшифровує заголовок програми. Якщо в заголовку зазначено, що програмі необхідна динамічна бібліотека, то адміністратор процесів завантажує розділювану бібліотеку, що містить динамічний компонувальник, і передає цьому динамічному компонувальнику керування. Динамічний компонувальник виконує пошук і завантаження в пам'ять необхідної динамічної бібліотеки. Ця бібліотека додається у внутрішній список усіх завантажених бібліотек, супроводжуваний динамічним компонувальником.

Динамічний компонувальние здійснює пошук потрібної розділюваної бібліотеки в наступному порядку.

1. Спочатку виконує перевірку, чи була завантажена в пам'ять поділювана бібліотека, переглянувши список завантажених бібліотек. Якщо бібліотека ще не завантажена, то компонувальник починає пошук.
2. Якщо зазначене абсолютне шляхове ім'я, то завантажується саме зазначена бібліотека. Якщо бібліотека з зазначеним ім'ям не існує, то подальший пошук не здійснюється.
3. Якщо зазначене не абсолютне шляхове ім'я, то динамічний компоновщик здійснює пошук бібліотеки в каталогах, перерахованих в перемінній оточення LD_LIBRARY_PATH.
4. Якщо розділювана бібліотека як і раніше не знайдена і якщо динамічна секція коду, що виконується, містить запис DT_RPATH, то пошук виконується і по шляху, зазначеному у DT_R PATH.
5. Якщо розділювана бібліотека як і раніше не знайдена то компонувальник виконує пошук у каталогах, визначених в перемінній оточення LD_LIBRARY_PATH модуля **procnto.** Якщо ця перемінна не була визначена, то використовується значення за замовчуванням — каталог, що містить завантажувальний образ ОС.

Крім того, процес може завантажувати поділювану бібліотеку під час виконання, використовуючи функцію *dlopen,* і, коли більше не має потреби в ній, вивантажувати її за допомогою функції *dlclose.* Пошук, завантаження і вивантаження динамічної бібліотеки й у цьому випадку виконує динамічний компонувальник (він міститься в поділюваному об'єкті libqnx.so).

Одержання інформації про процеси

Події створення і знищення процесів і потоків фіксуються за допомогою пакета трасування SAT (System Analysis Toolkit). Однак часто буває потрібно швидко одержати інформацію про поточний стан процесів і потоків. Для цього до складу QNX входить кілька утиліт:

- ps – основна POSIX-утиліта для моніторингу процесів. Вона включена в QNX як для сумісності POSIX, так і для зручності адміністраторів, що недавно працюють у QNX;
- sin – дуже інформативна QNX - утиліта моніторингу процесів. За допомогою sin можна, задавши відповідну опцію, одержати інформацію про процеси на іншому вузлі мережі Qnet. За замовчуванням sin видає для кожного процесу: PID, розмір коду, розмір стека і використання процесора.

За допомогою аргументів-команд можна одержати додаткову інформацію:

- args — показати аргументи процесів;
- cpu — показати використання ЦПУ;
- env – показати перемінні оточення процесів;
- fds — показати відкриті файлові дескриптори;
- flags — показати прапорці процесів;
- info — показати загальну інформацію про систему;
- memory — показати пам'ять, використовувану процесами;
- net — показати інформацію про вузли мережі;
- registers — показати стан регістрів;
- signals — показати сигнальні маски;
- threads — показати інформацію по потокам;
- timers — показати таймери, встановлені процесами;
- users — показати реальні й ефективні ідентифікатори власників і груп процесів.

Для виконання команд досить ввести перші два символи, наприклад команда sin flags рівнозначна команді sin fi.

pidin - ця утиліта з'явилася в QNX тільки з 6-й версії і призначена для одержання детальної інформації про потоки.

Крім цих утиліт, ефективний засіб моніторингу процесів є в складі QNX IDE - перспектива QNX System Information.

Додаткові способи IPC.

Як уже згадувалося, поза мікроядром Neutrino за допомогою додаткових адміністраторів ресурсів реалізовано кілька способів міжзадачних взаємодій (МЗВ):

- черги повідомлень POSIX (реалізовані в адміністраторі черг mqueue);
- іменовані канали (реалізовані в адміністраторі файлової системи POSIX);
- неіменовані канали (реалізовані в адміністраторі каналів pipe).

Черги повідомлень POSIX.

Черги повідомлень POSIX реалізовані в QNX за допомогою адміністратора черг mgueue. Адміністратор mqueue реєструє у просторі шляхових імен префікс /dev/mqueue, що має тип "каталог". Черги повідомлень POSIX – це іменовані об'єкти, тому даний механізм можна використовувати для обміну даними між процесами як у рамках однієї ЕОМ, так і між процесами, що працюють на різних вузлах мережі. Повідомлення POSIX не копіюються безпосередньо з адресного простору одного процесу в адресний простір іншого, як це роблять повідомлення QNX, а використовують проміжний адресний простір адміністратора mqueue. Тому повідомлення POSIX працюють відносно повільно.

Іменовані і неіменовані канали.

Канали, чи *програмні канали,* — це один із традиційних способів (МЗВ) у UNIX. Призначення каналу – забезпечити односпрямовану передачу даних від одного процесу до іншого. При цьому вивід однієї програми з'єднується з вводом іншої. У системному адмініструванні неіменовані канали використовуються для створення так званих *конвеєрів* — ланцюжків послідовно виконуваних команд, коли результат однієї команди є вхідними даними для іншої, наприклад:

```
cat /etc/services | grep telnet
```

Тут показано, що канал створюється за допомогою символу "|". У результаті виконання цього конвеєра на екран будуть виведені тільки ті рядки файлу /etc/services, що містять слово "telnet".

Відмінності між іменованими і неіменованими каналами:

- іменовані канали являють собою особливий тип файлу, що зберігається у файловій системі (тобто один процес пише дані в цей файл, а іншой – читає), тому іменовані канали працюють повільніше, але можуть використовуватися для МЗВ між будь-якими процесами в мережі;
- неіменовані канали реалізовані за допомогою адміністратора каналів pipe, що і виконує буферизацію даних;
- неіменовані канали можуть використовуватися для МЗВ тільки між процесами, зв'язаними відносинами "батьківський" - "дочірній".

Незважаючи на те, що неіменований канал працює швидше іменованого, він уступає по швидкості QNX-повідомленням.

Лекція 6

Відмовостійкі системи реального часу.

1. Вимоги до систем реального часу.

Системи реального часу - структура, що складається з безлічі пов'язаних між собою елементів, що виконує певні функції, яка реагує протягом певного часового проміжку на будь-який вхідний вплив. Це визначення дає уявлення про вимоги, що надаються до систем даного роду. Дані вимоги можна звести до трьох понять: Дедлайн (deadline) - директивний термін завдання. Це поняття зводиться до того, що виконання будь обробки сигналу має відбуватися не більше, ніж за строго певний часовий період, і закінчитися до певного моменту часу, який якраз і називається дедлайном. Джиттер (jitter) - розкид у значеннях. У даному випадку мова йде про час відгуку, тобто часу обробки сигналу. При цьому в середньому, незалежно від того коли прийшов на систему сигналу, значення повинні бути однаковими. Це означає, що в СРЧ джіттер повинен бути максимально мінімізований. Латентність (latency) - затримка реакції системи. Так як визначено поняття дедлайну, природним стає мінімізація затримки реакції системи на переривання, тобто мінімізація латентності. При цьому значення латентності завжди менше значення часу обробки розглянутого сигналу, бо інакше спостерігається порушення дедлайну. Таким чином, СРЧ повинна реагувати на будь-яке вхідний вплив протягом певного часу, заданого при проектуванні систем. При цьому для різних типів сигналів час реакції системи може бути по-різному (з урахуванням мінімізації значення джитера), але обов'язково строго регламентовано в рамках розробленої системи. СРЧ повинна реагувати на кілька одночасно находжених на вхід сигналів, при цьому жоден із сигналів не повинен оброблятися довше заданого для нього часу (потрібне дотримання поняття дедлайну).

2. Параметри операційні системи реального часу.

Операційні системи реального часу - операційні системи, що забезпечують інтерфейс для роботи з СРЧ. На відміну від вимог до СРЧ, вимоги до ОСРВ більше, тому що мова йде не про теорію, а про практику, тобто про можливість реалізувати

задану систему насправді. У свій час, Мартін Тіммерман, дослідник в області операційних систем реального часу, сформулював вимоги, необхідних для будь-якої операційної системи (ОС), яка претендує на звання системи реального часу:

- ОС повинна бути мультипрограмною і мультизадачною;

- ОС повинна використовувати переривання для диспетчеризації, тобто допускати витіснення однієї оброблюваної програми інший. Ця вимога приводить до наступного: - ОС повинна бути з розвиненою системою пріоритетів. ОС повинна мати систему наслідування пріоритетів, тобто підвищення рівня пріоритету потоку до рівня потоку, який його викликає, для виключення явища інверсії (коли завершення обробки завдання з більш високим пріоритетом залежить від обробки завдання з меншим пріоритетом). Для опису цієї вимоги можна навести наступний умовний приклад: завдання з пріоритетом 3 викликає підзадачу з пріоритетом 9. Під час роботи підзадачі, в систему надходить завдання з пріоритетом 6. Через це процес з пріоритетом 9 перерветься, і дасть можливість роботи тільки що надійшовшому процесу. Однак це неправильно, так як підзадача була лише проміжним етапом у роботі процесу з пріоритетом 3, який важливіший, ніж знову надійшовши процес, а, отже, має бути завершений раніше. ОС повинна забезпечувати механізми синхронізації завдань. Поведінка ОС має бути добре прогнозовано, кажучи більш точно - поведінка системи має бути суворо детерміновано.

Розглянувши вимоги до ОСРЧ, можна зробити висновок про параметри, важливих для розробки систем. До них належать:

- час реакції системи на переривання (розглянуте вище при перерахуванні вимог до СРЧ);

- час перемикання контексту (час перемикання від одного процесу до іншого).

- розмір системи управління (фактично, мова йде про розмір ядра).

При цьому слід зазначити, що з часом цей параметр стає все менш важливим. Всі великі обсяги пам'яті стають доступними для використання в будь-яких ситуаціях. Можливість виконання системи з ПЗУ, що означає можливість створення компактних вбудованих СРЧ підвищеної надійності, з обмеженим енергоспоживанням, без зовнішніх накопичувачів.

3. Архітектура ОСРЧ.

Для виконання відповідних функцій потрібна певна архітектура операційної системи. У процесі досліджень було виділено 3 види ОС, при цьому у кожного виду архітектури є свої недоліки. Розглянемо їх:

Монолітна архітектура.

Переваги: підвищену швидкодію системи, так як ядро збігається з системою.

Недоліки: як не дивно, але наявність безлічі модулів забезпечує складність системи, що призводить до непередбачуваності поведінки системи [6] в процесі експлуатації в деяких системах (дуже складно передбачити всі можливі реакції системи на ті чи інші вхідні потоки даних).

Багатошарова архітектура.

Переваги: можливість звернення до апаратури безпосередньо, минаючи сервіси системи, забезпечує зниження часу відгуку і збільшує передбачуваність реакції системи (простіше відстежити, яка частина дає збій, у разі помилки).

Недолік: відсутність багатозадачності [6].

«Клієнт-сервер» архітектура.

Переваги: підвищується надійність системи та її відмовостійкість, так як система фактично складається з безлічі сервісів, відмови яких легко простежуються і не критичні для роботи системи в цілому (можлива перезавантаження сервісу без перезавантаження системи). Також подібна архітектура покращує масштабованість системи, тому що можна видаляти непотрібні сервіси і легко додавати додаткові компоненти.

Недолік: введення захисту пам'яті підвищує час перемикання з одного процесу на інший (час перемикання контексту), що збільшує час роботи системи.

4. Механізми реального часу.

Найважливішим критерієм оцінки роботи системи реального часу є час, тому для розробників особливо важливо вміти управляти процесами так, щоб тимчасові обмеження виконувалися. У теорії існує система, яка управляється шляхом опису можливих подій на об'єкті. Для кожної події вказується лише адреса виконуваної команди і дедлайн. Проте в реальності таких систем немає, і розробникам

доводиться розглядати всі параметри систем реального часу для складання сценарію роботи ОСРЧ. Для цього програмісти використовують спеціальні механізми реального часу. Система пріоритетів процесів (завдань) і алгоритми диспетчеризації (планування). У розробці ефективного планування використовуються два підходи: статичні і динамічні алгоритми диспетчеризації. У першому випадку пріоритети присвоюються задачах до початку їх виконання, в другому ж пріоритети призначаються динамічно, враховуючи дедлайн завдань. При цьому, як правило, планувальники ОСРЧ враховують можливість витіснення завдань у разі потреби. Механізми взаємодії між завданнями та розподілу ресурсів. До таких механізмів відносяться: семафори, м'ютекси, події, сигнали, засоби для роботи з пам'яттю, канали даних (pipes), черги повідомлень. При цьому важливим залишається швидкість виділення того чи іншого ресурсу, а також час, на який він блокується використовуваним процесом. Засоби для роботи з таймером. Ці механізми забезпечують безпосередню роботу з часом. Вони дозволяють задавати і вимірювати різні проміжки часу, а також генерувати переривання.

5. Поняття відмовостійкості.

5.1 Причини збоїв.

Відмови з причини виникнення діляться на конструкційні та експлуатаційні. Експлуатаційні виникають при неправильному використанні системи, в той час як конструкційні відбуваються з вини розробників, які не передбачили всі можливі реакції системи. У системах реального часу можливе виникнення двох класів збоїв: програмних і апаратних. При цьому виникнення перших найчастіше означає порушення принципів реального часу, і фактично характеризує систему як систему загального призначення (не застосовні до завдань реального часу). Поняття надійності програмного забезпечення неконструктивне, це означає, що на етапі тестування програми не були виявлені всі помилки, тому передбачається, що вони відсутні. Забезпечити відмовостійкість системи на апаратному рівні складніше. При виникненні того чи іншого збою система повинна продовжувати функціонування. Ясно, що в разі монолітної системи відмова будь-якій частині призведе до відмови

системи в цілому. Розглядаючи апаратні збої, можна виділити наступні основні класи відмов апаратури:

- відмова процесора (якогось елементи системи);
- відмова зв'язку між елементами системи.

Ідентифікація відмови процесора якогось вузла мережі класифікується як відмова всього вузла: він ізолюється від решти мережі на логічному рівні і за наявності відповідної підтримки відключається на апаратному рівні (вимикається живлення). Ідентифікація відмови лінії зв'язку між елементами призводить лише до зменшення ступеня зв'язності вузлів мережі. Лінія зв'язку ізолюється на логічному рівні шляхом зміни маршрутів передачі повідомлень між вузлами мережі. Основна особливість мережевий відмовостійких технології - відсутність будь-якого (навіть самого незначного) єдиного компонента (ресурсу), вихід з ладу якого призводить до фатального відмови всієї системи. Така система не може містити будь-якого "центрального" (головного) вузла, розміщеного в одному з процесорних елементів системи, вона може складатися тільки з "рівноправних" частин, розміщених в кожному вузлі мережі. Передбачити і усунути всі причини збоїв не реально. З плином часу виявляються всі нові джерела небезпеки для систем. Зрештою, завдання розробника - створити систему, для якої була б визначена реакція на будь-яку подію, навіть помилку. Таким чином, будь-який збій приведе до заздалегідь певної поведінки, що має забезпечити роботу системи фактично в будь-яких умовах.

5.2 Елементи відмовостійкості.

-*Ізоляція додатків* - гарантія того, що жодна з програм не зможе зруйнувати ні інші додатки, ні базову операційну систему. Зазвичай це завдання вирішується шляхом використання блоків управління пам'яттю (диспетчерів пам'яті), для чого в операційній системі є або набір користувальницьких API-інтерфейсів (спеціально організованих низькорівневих методів, що надаються системою додаткам) для призначення прав читання/запису, або, що зустрічається більш часто, механізм створення областей пам'яті з обмеженим доступом. При спробі звернення в заборонену область виконання програми припиняється і запускається механізм відновлення після збою. І що найважливіше, обробка виняткових ситуацій може

бути реалізована в окремому адресному просторі додатка і ніяк не впливати на інші частини системи. Існує кілька дуже важливих елементів відмовостійкості, що гарантують високу надійність системи.

-*"Відновлення" ресурсів* - після завершення роботи прикладної програми (незалежно з якої причини), виділені їй ресурси повинні бути повернуті в систему, якщо тільки вони не оголошені як резервні. За відсутності цієї можливості система з частими запусками/зупинками додатків швидко приходить до стану неефективної роботи у зв'язку з недоліком необхідних ресурсів і необхідності її перезавантаження. Враховуючи потребу в обслуговуванні, в цих випадках потрібна певна гнучкість, що дозволяє контролювати живучість об'єктів, відповідальних за підтримання рівнів сервісу в процесі модернізації. Наприклад, серверна завдання, витягується запити з черги повідомлень, може бути замінена без будь-якого повідомлення клієнтських програм, оскільки черга продовжує існувати незалежно від заміни і може накопичувати запити, поки оновлений сервер не почне працювати. До складу операційної системи повинні входити засоби перепризначення прав володіння, для того щоб постійно існуючі об'єкти не поверталися в систему під час відновлення ресурсів.

-*"Користувацький/адміністративний" режими* - якщо додаток виконується в "системному", або адміністративному режимі, то йому стають доступними деякі привілейовані операції, які можуть призвести до незворотних наслідків. Ідеальна операційна система повинна володіти достатньою гнучкістю, забезпечуючи створення привілейованих користувальницьких "процесів", які можуть виконуватися в адміністративному режимі, але тільки після відповідної аутентифікації.

-*Захист від перевитрати ресурсів* - коли для користувача програми пишуться без контролю над використанням ними системних ресурсів або містять "розмножуючі" потоки виконання, надійність системи внаслідок виконання таких програм може серйозно знизитися. Захист від перевитрати ресурсів, наприклад, могла б передбачати обмеження рівнів пріоритету користувача завдань, контроль переповнення користувацького стека щоб уникнути руйнування вмісту пам'яті інших завдань і т.д. Ще більш підвищити відмовостійкість системи можна шляхом

визначення "м'яких" (що означають видачу попереджувальних повідомлень) і "жорстких" (сигнал про вичерпання ліміту користування ресурсом) порогів, при порушенні яких запускаються відповідні дії з відновлення.

5.3 Способи забезпечення відмовостійкості.

Для забезпечення надійного вирішення завдань в умовах відмов системи застосовуються два принципово різні підходи - відновлення рішення після відмови системи (або її компонента) і запобігання відмови системи (відмовостійкість). Системи, стійкі до відмов, або маскують відмови, або поводяться у разі відмови заздалегідь певним чином. У міру того як операційні системи реального часу і вбудовані комп'ютери все частіше використовуються в критично важливих додатках, розробники створюють нові ОС реального часу високої готовності. Ці продукти містять у собі спеціальні програмні компоненти, які ініціюють попередження, запускають системну діагностику для того, щоб допомогти виявити проблему, або автоматично перемикаються на резервну систему. Забезпечення відмовостійкості передбачає парирування дії константних відмов і маскування збоїв (перемежованих відмов), тобто запобігання поширення наслідків збою на продовження виконання системою своїх функцій. Парирування дії відмов завжди пов'язане з введенням в систему того чи іншого виду надмірності. Існує кілька способів забезпечення відмовостійкості роботи систем. Один з них заснований на фіксуванні константної відмови або збою системи в цілому або в її окремих частинах з наступною реконфігурацією системи. Такий спосіб пов'язаний з перериванням функціонування системи, тобто не забезпечує відмовостійкість у системах реального часу.

Відомий спосіб забезпечення відмовостійкості, що дозволяє маскувати збої і заснований на мажоризації (математичний термін з теорії множин), тобто використанні 2n +1 каналів (в даному способі кількість процесорів не грає ролі, робота може проводитися і на одному, важливо збільшення числа каналів) і схеми голосування, яка відбирає ті вихідні дані, які представляють більшість. Такий спосіб і використовується для систем реального часу. Мажоризацію може бути здійснено або апаратно, або програмно, або в комбінації цих способів. Недоліком таких

способів є значна кількість обладнання, навіть у мінімальному варіанті при n = 1. Іншим недоліком способів мажоризації є значні втрати продуктивності. При апаратній реалізації втрата продуктивності пов'язана з необхідністю синхронізації процесів в резервованих каналах. При програмній реалізації швидкодію системи знижується через витрати часу на обмін інформацією між каналами. Причина такої неефективності полягає в тому, що і при апаратній, і при програмній організації механізм маскування збоїв, тобто голосування, визначення несправного каналу, його блокування та подальше включення в нормальну роботу, використовується в кожному такті роботи системи незалежно від наявності або відсутності збоїв. Ці тимчасові втрати при практичній реалізації досягають 30-50%. До недоліків мажоризації при його реалізації слід віднести також велику кількість зв'язків між каналами і значні труднощі при проектуванні. Слід зазначити, що при апаратній мажоризації нормальне функціонування при деградації системи до одного каналу забезпечується лише при додаткових апаратних і тимчасових витратах. При програмній мажоризації у разі константних відмов реконфігурація до одного справного каналу можлива без додаткових апаратних витрат. Але збільшення кратності маскованих збоїв на відміну від апаратної мажоризації, де це можна здійснити шляхом організації багаторазового голосування при проходженні сигналів по системі або відповідно шляхом введення апаратної надмірності неможливо. Скорочення апаратної та часової надмірності і розширення функціональних можливостей в деяких випадках досягається тим, що в способі, що полягає в маскуванні збоїв шляхом резервування і включає визначення наявності збоїв, ідентифікацію і блокування несправних каналів для маскування збоїв, використовують незалежну одночасну роботу N каналів, число яких на одиницю більше кратності маскованих збоїв, сигнали яких подають на загальний вихід. За сигналами, отриманими від детекторів збоїв, що входять до складу кожного каналу, виробляють блокування проходження сигналів від каналів, в яких відбулися збої і пропускають на вихід той з сигналів від справних каналів, який приходить першим за часом. При цьому визначення наявності збою, ідентифікація і блокування несправних каналів проводиться або після проходження на вихід сигналу, що прийшов першим за часом, або до його надходження.

6. Відмовостійкість в існуючих системах реального часу.

6.1 QNX Neutrino.

QNX - перша комерційна ОС, побудована на принципах мікроядра та обміну повідомленнями. Система реалізована у вигляді сукупності незалежних (але взаємодіючих через обмін повідомленнями) процесів різного рівня (менеджери і драйверів), кожен з яких реалізує певний вид сервісу. У процесі розробки системи в якості її основного застосування розглядалися критичні ситуації, при яких відмова приводив би до великого матеріального збитку або навіть до людських жертв [13]. У зв'язку з цим у QNX Neutrino поряд з мікроядерної архітектурою з повною ізоляцією модулів в ОЗУ передбачено ряд механізмів, що забезпечують надійність автоматизованих систем. Мікроядерна архітектура з повною ізоляцією модулів в ОЗУ.

У подібній архітектурі мікроядро є зв'язуючим елементом, що забезпечує зв'язок елементів системи між собою, але при цьому не реалізує більшість системних сервісів. Це дозволяє розробнику самому вирішити, які сервіси потрібні для вирішення його прикладної задачі, дозволяє самому написати необхідні системні програми і підключити їх до системи, створюючи по можливості ОС дуже невеликого розміру і вузької спеціалізації. Одночасно з цим мікроядро відповідає за реалізацію всіх механізмів підтримки жорсткого реального часу:

- фіксовані пріоритети потоків (256 рівнів) і функція-обробник ISR (обробник переривань - спеціальна процедура, що викликається по перериванню для виконання його обробки);
- миттєве витіснення завдання з меншим пріоритетом;
- витісняються системні виклики;
- захист від інверсії пріоритетів на базі протоколу успадкування пріоритетів;
- виняток непередбачених витрат ресурсів;
- механізм трасування ядра, що дозволяє дізнатися точний час кожної операції.

Механізм адаптивного квотування ресурсів ЦПУ і ОЗУ.

Найпростіше розглянути механізм на прикладі. Є система і шість виконуваних в ній потоків (A, B, C, D, E, F), у кожного з яких має свій пріоритет (у A максимальний, у

F - мінімальний). Для квотування (поділу) ресурсів процесора створені три логічних розділи по 30%, 40% і 40% від усього процесорного часу. При цьому потік A виконується у розділі 1, потоки B, C і D - у розділі 2, а всі інші - у третьому розділі. Якщо якісь потоки не повністю використовують свій розділ (випадок розділу № 3), то процесорний час розподіляється між готовими до виконання потоками всіх розділів відповідно до їх пріоритетів (звідси приставка «адаптовані» у назві механізму).

Механізм забезпечення «гарячої» заміни сервісів.

Існує два варіанти використання механізму «гарячої» заміни.

Перший варіант - запуск «тіньового» сервера, дублера, який у разі збою основного сервера починає обробку його запитів. Можливі варіанти, при яких тіньовий сервер починає роботу до збою основного, а також ситуації, в яких один сервер має безліч дублерів. Другий варіант - оновлення сервера без перерви в обробці клієнтських запитів. У разі запуску нового сервера забезпечується перенаправлення запитів на нього шляхом реєстрації у нього імені старого сервера, в той час як він може закінчити обробку старих запитів і припинити свою роботу.

Технологія швидкої активізації пристроїв.

Цей механізм дозволяє ще до завантаження і в процесі завантаження ОС забезпечувати відгук на зовнішні події. Йдеться про можливість приступити до обробки критичних сигналів через десятки мілісекунд після подачі живлення на ЦПУ. Особливість технології полягає в тому, що після закінчення завантаження ОС обробку запитів може продовжити повноцінний драйвер без затримок часу або втрат даних. Ця технологія має назву технології міні-драйверів. Технологія міні-драйверів - це підхід з застосуванням компактних, високоефективних драйверів пристроїв, які починають виконуватися до ініціалізації ядра ОС. Міні-драйвер, визначений у завантажувальному файлі системи, може швидко і своєчасно реагувати на повідомлення, що видаються при включенні живлення, забезпечуючи прийом усіх без винятку повідомлень під час завантаження ОС. Виробляється опитування всіх пристроїв, що входять в систему, при цьому у випадку, якщо відповідь не отримана,

робиться висновок про відмову даного пристрою. Після завантаження ОС міні-драйвер може продовжити свою роботу або передати управління повної версії драйвера, яка потім може обробити всі повідомлення, що буферизованні міні-драйвером.

Механізм формування розподіленої обчислювальної середовища.

Будучи традиційним, механізм пов'язує в єдину мережеву інфраструктуру безпосередньо ядра копій QNX Neutrino, що виконуються на різних ЕОМ мережі. З точки зору додатків і системних сервісів автоматизованої системи виконання відбувається на одній ЕОМ, при цьому виникає псевдоєдина віртуальна суперЕОМ (приставка «псевдо» використовується, щоб підкреслити незалежність окремих вузлів мережі).

Механізм підтримки резервування фізичних каналів зв'язку в кластері.

Для кожного логічного з'єднання процесу-клієнта з процесом-сервером може бути визначений один з трьох варіантів фізичного каналу:

- з автоматичним балансуванням навантаження. Даний варіант характерний тим, що протокол мережі Qnet самостійно приймає рішення по якій з усіх доступних ліній зв'язку передавати пакети (при цьому аналізуються різні параметри, як той час відгуку видаленого вузла на запити і т.д.)
- з перевагою. Цей варіант відрізняється від попереднього лише тим, що задається переважний мережевий адаптер, який у разі відсутності замінюється за схемою першого варіанту.
- ексклюзивний. У даному випадку строго задається певний адаптер. У разі його відмови дані не будуть передаватися зовсім, навіть якщо для передачі доступні інші лінії зв'язку. Що важливо: зміна варіанту не вимагає перекомпіляції програм і може відбуватися на будь-якому етапі життєвого циклу системи. Таким чином, в роботі системи використовується не випадковий канал, а вибраний в ході аналізу певних параметрів при проектуванні системи (з перевагою, ексклюзивний) або в ході її роботи, тобто виключається будь-яка випадковість, що підвищує відмовостійкість системи.

Механізм забезпечення балансування навантаження на сервіси.

У разі, коли обчислювальних ресурсів системи недостатньо, в мережеву автоматизовану систему можуть бути додані додаткові ЕОМ, на які можливо перекласти вирішення частини завдань. Суть механізму полягає в тому, що при отриманні від клієнта запиту на встановлення з'єднання сервер може замість повернення клієнту ідентифікатора з'єднання перенаправити клієнта до іншого сервера.

Технологія автоматичного відновлення процесів.

В основі технології лежить монітор ключових процесів, основне призначення якого - максимально швидке визначення факту збою серверного додатку і вживання заходів для відновлення нормальної роботи сервісу. Для цього монітор на початку роботи дублює себе, а потім обмінюється з клоном інформацією про оброблюваних процесах. Як тільки один з моніторів починає давати збої, другий породжує свого дублера і так далі.

Технологія автоматизації відновлення логічних з'єднань.

Ця технологія призначена для збереження логічних з'єднань між клієнтами і серверами у випадках тимчасових відмов фізичних ліній зв'язку. Для цього в клієнтських програмах передбачена обробка кодів помилок, що виникають при розривах сполук, для подальшого відновлення зв’язків.

ОС VxWorks AE (Advanced Edition) - перша комерційна ОС реального часу, призначена для побудови високонадійних відмовостійких систем. У VxWorks AE підтримується автоматичне відновлення регенерація ресурсів, запобігає "витоку" ресурсів (які в іншому випадку, так чи інакше, відбуваються в будь-якій динамічній системі). Також у системі є вбудований розподілений механізм обміну повідомленнями, що сприяє певному підвищенню ступеня відмовостійкості. На відміну від звичайної VxWorks, VxWorks AE побудована за технологією "Protection Domain", що дозволяє ізолювати ядро від додатків і додатки один від одного. Дана

технологія вимагає більш уважного перегляду. Domain (домени захисту) - принцип структурування ядра ОС, при якому розробник може ізолювати процеси та ресурси за допомогою спеціального контейнера ресурсів, в якому задаються параметри середовища виконання. Існує два способи створення доменів захисту: статичний (при запуску ОС) і динамічний (в ході роботи ОС). При цьому кожен домен захисту характеризується власним адресним простором і має декілька видів доступу до нього з боку інших доменів. Однак домени захисту, в яких виповнюється багатозадачний додаток, у віртуальному адресному просторі насправді перекриваються. Але виконуюча в одному домені захисту прикладна програма не «бачить» інші домени. Таким чином, організується ізоляція додатків і, відповідно, відмов, що призводить до підвищення надійності. Кожен домен захисту додатків має свою власну таблицю символів і власний пул динамічної пам'яті (heap). Таким чином, кілька копій одного і того ж додатка може бути завантажено та запущено одночасно в декількох доменах захисту. У кожному домені захисту може виконуватися будь-яку кількість завдань, так що якщо в домені захисту є будь-якої реєнтерабельний код, то всередині цього домену може бути одночасно запущено кілька екземплярів однієї і тієї ж програми шляхом породження в цьому домені безлічі завдань з однією і тією ж точкою входу. Крім того, якщо в один і той же домен захисту завантажується код для декількох програм, то в цьому домені може бути породжене кілька завдань з окремими точками входу. Подібна гнучкість забезпечує прикладного розробнику широку свободу розбиття системи на домени захисту на власний розсуд. Контроль переповнення стека кожного завдання і можливість автоматично збільшувати розміри пулу динамічної пам'яті (у певних межах) у разі переповнення являє собою одну із заходів захисту від перевитрати ресурсів. Крім того, така система дозволяє обмежувати діапазон задання пріоритетів всередині доменів захисту. Таким чином, запобігає "некероване розмноження" додатків і в будь-якому випадку усувається їх можливий вплив на виконання інших прикладних задач. Глобальними об'єктами, що забезпечують синхронізацію виконання додатків, як і раніше є механізми взаємодії між процесами типу черг, семафорів, каналів і конвеєрів.

В операційній системі VxWorks AE домен захисту визначає також і права володіння системними ресурсами (завдання, черги повідомлень, семафори, сторінки пам'яті і т.д.). Домен захисту, якому виділені або в якому створені ці ресурси, стає для них "базовим" доменом (home domain). Незалежно від способу завершення роботи прикладної програми всередині домену захисту, всі виділені цього домену ресурси відновлюються системою для їх повторного використання. На даний момент процес відновлення ресурсів у VxWorks певною мірою є ручним процесом, виконуваним для кожного об'єкта окремо (наприклад, створення завдання, знищення завдання, створення черги, знищення черги). В ОС VxWorks AE передбачені більш потужні, зручні і наочні засоби, що дозволяють відслідковувати права власності на всі ресурси (пам'ять, файлові дескриптори і т.д.) за індивідуальними, заданим користувачем зв'язків, обмеженим контейнером домену захисту. У VxWorks AE реалізований новий масштабований механізм обміну повідомленнями, який розширює можливості локальної системи передачі повідомлень (тобто черги повідомлень в "рідній" середовищі) за межі одного процесора. ОС автоматично визначає, локальна чи ні дана чергу для процесора, і при необхідності пересилає повідомлення у вилучені черги, користуючись службою або реєстром розподілених і резервних імен. Висока надійність обміну повідомленнями гарантується протоколом передачі завдяки підтримці підтверджень прийому повідомлень і нумерації пакетів, що забезпечують цілісність прийнятих повідомлень. Підвищення відмовостійкості розподіленого обміну повідомленнями в VxWorks AE забезпечується API-інтерфейсами, що надають доступ до резервної базі даних сервера імен для віддаленої маршрутизації повідомлень. У системі немає одиничних точок відмови, і якщо будь-які сервісні функції (або весь вузол), пов'язані з обслуговуванням віддалених черг, стають недоступними, черги повідомлень можуть бути повторно створені і зареєстровані в інших вузлах завдяки наявності відповідних посилань у базі даних. Потім здійснюється оновлення бази даних сервера імен, і "пункт обслуговування" переміщається в справний вузол. Це надзвичайно потужна можливість, особливо при використанні резервного обладнання для підвищення доступності сервісу. Більше того, як виявилося, система обміну повідомленнями в VxWorks AE володіє практично необмеженими можливостями масштабування, що

було продемонстровано на існуючих VxWorks-системах з числом вузлів до 512. Важливо, що операційна система VxWorks AE є достатньо гнучкою, щоб підтримувати обмін повідомленнями там, де це дійсно має сенс, і не застосовує його для обміну інформацією між локальними процесами, де він призводить до зайвим витратам.

Архітектура високої готовності Foundation HA, розроблена в компанії Wind River, має здатність відрізняти базові причини виникають у системі збійних ситуацій (відмови) і ознаки, що з'являються як свідчення цих причин. Одиничний відмову (типу відмови якого програмного або апаратного компонента) може стати джерелом безлічі ознак. У Foundation HA є стандартний API-інтерфейс сповіщення, за допомогою якого посилені програмні компоненти передають ознаки на рівень оповіщення Foundation HA.

Призначення системи управління ознаками AMS (Alarm Management System) - забезпечення єдиної інфраструктури для коду відновлення після відмов, організованого у вигляді набору обробників ознак відмови. Оброблювачі реєструються в AMS і вказують, від яких джерел вони очікують ознаки. При їх надходженні система AMS визначає, якому оброблювачу передати той чи інший ознака для подальшого обслуговування. Якщо його причиною є відмова, то обробником є процедура відновлення після відмови. Оброблювач вміє зіставляти ознаки один з одним і групувати їх в єдиний відмову і запускати відповідну процедуру відновлення для всіх відразу. Призначення цієї процедури - ліквідувати наслідки відмов з мінімальним порушенням нормального функціонування системи. Посилені програмні компоненти класифікують ознаки відмов за ступенем їх серйозності, при цьому обробники можуть змінювати ступінь серйозності ознаки, спираючись на власні відомості про систему та додаткові умови, необхідні для коректної інтерпретації ступеня важливості ознаки відмови. Необхідно відзначити, що в системі можуть генеруватися певні набори параметрів, які насправді не є ознаками відмов. Зокрема, модуль розподілу пам'яті може генерувати ознаку відмови при виявленні нестачі пам'яті, але якщо подібна ситуація була передбачена специфікацією даної системи, то вона фактично відмовою не є.

Висновок.

У розділі були розглянуті поняття операційних систем реального часу і відмовостійкості, а також механізми забезпечення відмовостійкості. Крім загальних напрямків були розглянуті реально існуючі операційні системи і внутрішні рішення проблеми відмовостійкості компаній-розробників. Саме поняття систем реального часу засноване на тому, що створені системи повинні бути здатні не тільки гарантувати рішення поставленого завдання, але при цьому повинні дотримуватися тимчасові обмеження. Відмовостійкість систем реального часу є важливим параметром при їх проектуванні. Якщо знехтувати відмовостійкістю, можна потрапити в ситуацію, при якому система перестає функціонувати, при цьому завдання, які потребують вирішення в даний момент часу, простоюють або зовсім залишаються невирішеними. У ряді технічних додатків відмовостійкість є обов'язковою вимогою, що пред'являються державними наглядовими органами до технічних систем. У всіх способів забезпечення відмовостійкості є подібна риса - дублювання. Тільки таким чином можна забезпечити відмовостійкість системи - у разі збою одного з її елементів моментально замінити його. При цьому кожна розробляє компанія в праві самостійно визначати способи забезпечення відмовостійкості, розробляючи для цього певні алгоритми та спеціалізовану апаратуру. Основою даної роботи став розгляд двох операційних систем реального часу: QNX Neutrino і VxWorks AE. У кожній з цих систем присутні розроблені спеціально для них способи забезпечення відмовостійкості. При цьому в системі VxWorks AE розробка заснована на ізолюванні елементів системи, у той час як система QNX Neutrino відштовхується від надмірності. Кожна ОС має безліч механізмів забезпечення відмовостійкості, що дозволяє використовувати їх в різних галузях: у банківській справі, у військовій галузі, видобувних галузях, в хімічній промисловості, енергетиці і так далі. Забезпечення відмовостійкості є складним завданням, що вимагає досліджень у цій області, і одночасно затребуваною. Це дозволяє очікувати, що з часом з'являться нові способи забезпечення відмовостійкості в системах реального часу.

Список літературних джерел.

1. Сергей Зиль. Операционная система реального времени QNX от теории к практике. 2-е издание. Санкт-Петербург, “БХВ-Петербург”, 2004 -191с

2. Сергей Зиль. QNX momentics основы применения. Санкт-Петербург, “БХВ-Петербург”, 2005 -253с.

3. Роберт Кртен. Введение в QNX Neutrino. Руководство для разработчиков приложений реального времени (2-е издание). Санкт-Петербург, “БХВ-Петербург”, 2011 -368с.

4. Жданов А.А. Операційні системи реального часу. - PCWeek, 8/1999

5. Древс Ю.Г. Системи реального часу: технічні та програмні засоби: Навчальний посібник. - М.: МІФІ, 2010. 320 с.

6. Зиль С. Штатні механізми QNX Neutrino для забезпечення відмовостійкості обчислювальних систем жорсткого реального часу. - СТА, 3/2009, 118 с.

Зміст