МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“Київський політехнічний інститут імені Ігоря Сікорського”

В.Г. Зайцев, Є.І. Цибаєв

КОМП’ЮТЕРНІ СИСТЕМИ РЕАЛЬНОГО ЧАСУ

НАВЧАЛЬНИЙ ПОСІБНИК

Рекомендовано методичною радою КПІ ім. Ігоря Сікорського

як навчальний посібник для здобувачів ступеня магістра за освітньою програмою ”Системне програмування та спеціалізовані комп’ютерні системи” спеціальності 123 “Комп’ютерна інженерія”

Київ

«КПІ ім. Ігоря Сікорського»

2019

Рецензенти: Забара С.С., д.т.н., проф.
Савченко Ю.Г., д.т.н., проф.
Відповідальний редактор: Тарасенко В.П., д.т.н., проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол №8 від 25.04 2019р.) за поданням Вченої ради Факультету прикладної математики (протокол № 9 від 22.04.2019 р.)

Електронне мережне навчальне видання.
Зайцев Володимир Григорович, *д-р техн. наук, проф.*,
Цибаєв Євгеній Ігорович, кандидат техн. наук.

Комп'ютерні системи реального часу

Навчальний посібник

Комп'ютерні системи реального часу, навчальний посібник [Електронний ресурс]: навч. посіб. для здобувачів ступеня магістра за освітньою програмою "Системне програмування та спеціалізовані комп'ютерні системи" спеціальності 123 «Комп'ютерна інженерія»/ В. Г. Зайцев, Є. І. Цибаєв; КПІ ім. Ігоря Сікорського. - Електронні текстові дані (1 файл: 4 Мбайт). - Київ: КПІ ім. Ігоря Сікорського, 2019. - 162 с.

Навчальний посібник призначений для закріплення теоретичних знань, отриманих студентами при вивченні дисципліни «Системи реального часу» щодо типових підходів до розробки і особливостей функціонування основних компонентів систем реального часу (СРЧ) та набуття практичного досвіду з їх проектування, а також застосування типових механізмів побудови СРЧ при вирішенні конкретних інженерних задач.

ЗМІСТ

Вступ

Згідно визначенню, системи реального часу (СРЧ) - це системи, для яких специфіковані вимоги щодо часу виконання задач. У залежності від особливостей цих вимог розрізняють м'які та жорсткі СРЧ. СРЧ - належать до класу жорстких (hard), якщо порушення встановленого часу виконання задач неприпустимо. СРЧ називають м'якою (light), якщо отримання результату після наперед встановленого строку призводить лише до зниження якості функціонування системи.

Основною проблемою в операційних системах реального часу (ОС СРЧ) є таке планування задач, яке б забезпечувало прогнозовану поведінку системи за всіх можливих обставин.

Спочатку розглянемо жорсткі СРЧ, в яких задачі виконуються циклічно, створюючи неперервний потік задач на виконання. Зазвичай, в ОС СРЧ використовують планування з пріоритетами при якому привілеї надаються задачам, що мають найвищий пріоритет .

При створенні СРЧ вважається, що розробка планувальника в СРЧ має дві складові: планувальник періоду розробки (off-line) та планувальник періоду виконання (on-line, run-time). Зупинимось на планувальнику періоду розробки. На етапі розробки СРЧ у розпорядженні розробника має бути інформація про прикладні задачі та особливості їх взаємодії. Створення планувальника періоду розробки базується на обробці проектної інформації до початку роботи системи. Планувальник періоду виконання є складовою системи і починає діяти при її запуску. Алгоритм його роботи базується на інформації, яка отримана у результаті періоду розробки.

Планування періоду розробки включає аналіз можливості виконання вимог до СРЧ (schedulability analysis, feasibility analysis), необхідність в якому виникає, коли обмеження на час виконання задач системи чітко сформульовані і порушення цього часу може призвести до виходу системи з ладу.

Успішне завершення аналізу дає гарантію того, що за різних обставин виконання всіх задач буде завершено у заданий час.

При виконані робіт аналізують два різні типи планувальників: статичні та динамічні .

Статичне планування використовують тоді, коли ще до початку роботи системи є повна інформація про задачі, що вирішуються: час виконання, структура та часові характеристики задач, особливості їх взаємодії та способи обміну інформацією. У цьому випадку, на етапі розробки системи можна побудувати статичний графік, який має повну інформацію на період виконання, і планувальник періоду виконання вироджується у простий інтерпретатор статичного графіку.

Алгоритми планування для простих систем, де питання ефективності не є визначальним, не використовують системи пріоритетів. Однак, у більшості випадків, використовують алгоритми планування, що засновані на пріоритетах задач. Всі готові задачі впорядковуються по значеннях пріоритету і управління завжди отримує задача з найбільшим пріоритетом. Пріоритети задач визначають з урахуванням їх часових характеристик. Пріоритети можуть бути фіксовані або динамічні. Відповідно, по способу призначення пріоритетів задачам алгоритми планування поділяють на алгоритми планування з фіксованими або динамічними пріоритетами.

У першому випадку, пріоритети задачам призначаються при їх включенні у склад системи при розробці та залишаються незмінними на весь період роботи системи. При другому - вони змінюються у залежності від стану обчислювального процесу у системі.

Алгоритми, що використовують динамічні пріоритети, більш ефективні, але досить складні у реалізації. Алгоритми з фіксованими пріоритетами менш ефективні, але простіші в реалізації.

Пріоритетні планувальники поділяються на витисняючи і не витисняючі. Витисняючі - при появі більш пріоритетної задачі миттєво передають їй

управління процесором, а не витисняючі - дозволяють завершити менш пріоритетну задачу, що до того виконувалась.

Витисняючі планувальники забезпечують більшу ефективність планування і тому в ОС СРЧ набагато частіше використовуються .

Класичні алгоритми призначення пріоритетів задачам використовують тільки унікальні пріоритети кожній задачі. Однак, у ряді випадків бажаним є призначення ряду задач одного пріоритету. У такому випадку найбільш простим та ефективним методом розв'язання відносин між задачами одного пріоритету є обслуговування задач одного пріоритету у порядку надходження (FIFO) . Тобто, загальний алгоритм планування може бути віднесено до типу MQS (Multilevel Queue Scheduling), а алгоритм кожної черги - складової до типу FIFO. Для створення планувальника періоду розробки для комплексу програм реального часу, що періодично виконуються і утворюють неперервний у часі потік вимог на виконання, на першому етапі слід мати можливість дати оцінку:

- часу виконання програм додатку у монопольному режимі виконання;
- повного часу виконання програм з урахуванням переривань та часу очікування у відповідних пріоритетних чергах;

Більшість досліджень проблеми оцінки часу виконання програм на даний час пов'язана з вимірами часу виконання всієї програми або її окремих лінійних блоків у монопольному режимі виконання. При цьому, в якості вхідної інформації використовується текст програми мовою високого рівня, а засобами виміру обирають стандартні засоби виміру часу обраної операційної системи.

Для прогнозування часу виконання програм у програмних комплексах і системах, що використовують планування задач, як правило, застосовують методи, які базуються на аналізі моделей систем масового обслуговування (СМО), де запит на виконання кожної задачі ототожнюють з вимогою на обслуговування, а процес вирішення задачі - з обслуговуванням вимоги обслуговуючим пристроєм .

Окрім того, не слід забувати, що СРЧ - це фактично програмно - технічний комплекс, який повинен встигати з реакцією на події, що відбуваються на об'єкті управління. Тобто, встигати реагувати у передбачуваний час на непередбачуваний потік подій. Цей критичний час (deadline) для кожної події має бути визначений попередньо на стадії розробки технічного проекту на СРЧ.

Основною метою даного посібника є ознайомлення студентів з особливостями побудови програмного забезпечення (ПЗ) СРЧ і методами попередньої оцінки (шляхом моделювання) можливості забезпечити вимоги щодо часу виконання як окремих програм і реакцій СРЧ на зовнішні події, так і всього комплексу програм створюваної СРЧ. Така попередня оцінка потрібна для того, щоб правильно обрати засоби обчислювальної техніки, відповідну операційну систему реального часу (ОС РЧ), на базі яких у майбутньому можна буде гарантовано створити СРЧ з заданими характеристиками.

Обсяг і зміст посібника відповідають нормативним обсягам та змісту дисципліни, що вказані в освітньому стандарті спеціальності "Комп'ютерна інженерія", і тому є мінімально достатніми для засвоєння їх студентами. Для засвоєння матеріалу посібника необхідно мати математичну підготовку з дисципліни "Математичний аналіз" в обсязі, прийнятому для технічних університетів, а також програмістську підготовку з дисциплін "Програмування" та "Операційні системи" в обсязі, передбаченому освітніми стандартами по спеціальності "Комп'ютерна інженерія."

Матеріали посібника підготовлені викладачами кафедри "Системного програмування і спеціалізованих комп'ютерних систем" в Національному технічному університеті - Київський політехнічний інститут імені Ігоря Сікорського.

Розділ 1. Визначення та основні особливості систем реального часу

1.1. Визначення систем реального часу

На даний час існує кілька визначень систем реального часу (СРЧ) (real time operating systems (RTOS)), однак більшість з них суперечить один одному. Наведемо деякі з цих визначень, щоб продемонструвати різні погляди на призначення і основні завдання СРЧ:

1. Системою реального часу називається система, в якій успішність роботи будь-якої програми залежить не тільки від її логічної правильності, а й від часу, за який вона отримала результат. Якщо часові обмеження не задоволені, то фіксується збій в роботі систем.

Таким чином, часові обмеження повинні бути гарантовано задоволені. Це вимагає від системи бути передбачуваною, тобто незалежно від свого поточного стану і завантаженості видавати потрібний результат за необхідний час. При цьому бажано, щоб система забезпечувала якомога більший відсоток використання наявних ресурсів.

Прикладом завдання, де потрібно СРЧ, є управління роботом, що бере деталь зі стрічки конвеєра. Деталь рухається, і робот має лише невелике часове вікно, коли він може її взяти. Якщо він запізниться, то деталь вже не буде на потрібній ділянці конвеєра, і, отже, робота не буде зроблена, незважаючи на те, що робот знаходиться в правильному місці. Якщо він позиціонується раніше, то деталь ще не встигне під'їхати, і він заблокує їй шлях.

Іншим прикладом може бути космічний апарат, що знаходиться на автопілоті. Сенсорні серводатчики повинні постійно передавати в керуючий комп'ютер результати вимірювань. Якщо результат будь-якого вимірювання буде пропущено, то це може привести до неприпустимої невідповідності між реальним станом систем космічного апарату і інформацією про нього в керуючій програмі.

Розрізняють важкі (сильні - hard) і слабкі (soft) вимоги реального часу. Якщо запізнення програми призводить до повного порушення роботи керованої системи, то говорять про сильні вимоги у реальному часі (жорсткі СРЧ).

Якщо ж запізнювання приводить тільки до втрати продуктивності, то говорять про слабкі вимоги у реальному часі (м'ягкі СРЧ). Більшість програмного забезпечення орієнтоване на слабкий реальний час, а завдання хорошої СРЧ - забезпечити рівень безпечного функціонування системи, навіть якщо керуюча програма ніколи не закінчить своєї роботи.

2. Стандарт POSIX 1003.1 визначає СРЧ наступним чином: «Реальний час в операційних системах - це здатність операційної системи забезпечити необхідний рівень сервісу в заданий проміжок часу».

3. Іноді системами реального часу називають системи постійної готовності (on-line системи), або «інтерактивні системи з достатнім часом реакції». Зазвичай це роблять фірми-виробники з маркетингових міркувань. Якщо інтерактивну програму називають працюючою в реальному часі, то це означає, що вона встигає обробляти запити від людини, для якої затримка в сотні мілісекунд навіть непомітна.

4. Часто поняття «система реального часу» ототожнюють з поняттям «швидка система». Це не завжди правильно. Час затримки реакції СРЧ на подію уже не так і важливий (він може досягати декількох секунд). Головне, щоб цього часу було досить для розглянутого додатку і гарантовано. Часто алгоритм з гарантованим часом роботи менш ефективний, ніж алгоритм, що такою властивістю не володіє. Наприклад, алгоритм «швидкого» сортування (quicksort) в середньому працює значно швидше багатьох інших алгоритмів сортування, але його гарантована оцінка складності значно гірша.

5. У багатьох важливих сферах виконання додатків СРЧ вводяться свої поняття «реального часу». Так, процес цифрової обробки сигналу називають, що він іде в «реальному часі», якщо аналіз (при введенні) і / або генерація (при виведенні) даних може бути проведено за той же час, що і аналіз та / або генерація тих же даних без цифрової обробки сигналу.

Наприклад, якщо при обробці аудіо даних потрібно 2,01 секунди для аналізу 2,00 секунди звуку, то це не процес реального часу. Якщо ж потрібно 1,99 секунди, то це процес реального часу. Виходячи з вище сказаного, визначення системи реального часу можна подати в наступній інтерпретації.

Визначення. Система реального часу реагує в передбачуваний час на непередбачувану появу зовнішніх подій.

Це визначення пред'являє до системи цілком певні базові вимоги. Розглянемо вимоги, що пред'являються до систем реального часу.

1.2. Вимоги, що пред'являються до систем реального часу

Своєчасна реакція. Після того як відбулася подія, реакція має бути не пізніше, ніж через необхідний час. Перевищення цього часу розглядається як серйозна помилка.

Одночасна обробка інформації, яка характеризує зміну процесу декількох подій. Навіть якщо одночасно відбувається кілька подій, реакція на жодне з них не повинно запізнюватися. Це означає, що система реального часу повинна мати вбудований паралелізм. Паралелізм досягається використанням декількох процесорів в системі і / або багатозадачним підходом.

Розглянемо основні ознаки систем жорсткого і м'якого реального часу.

Ознаки систем жорсткого реального часу:

- неприпустимість жодних затримок, ні за яких умов;
- непотрібність результатів при запізненні;
- катастрофа при затримці реакції;
- ціна запізнення нескінченно велика.

Приклад системи жорсткого реального часу - бортова система управління літаком.

Ознаки систем м'якого реального часу:

- за запізнення результатів доводиться платити;
- зниження продуктивності системи, викликане запізненням реакції

на події, що відбуваються.

Приклад - автомат роздрібної торгівлі та підсистема мережевого інтерфейсу. В останньому випадку можна відновити пропущений пакет, використовуючи мережевий протокол, що повторює передачу пропущених пакетів. При цьому, звичайно, відбудеться зниження продуктивності системи.

Введемо поняття операційної системи (ОС). Операційна система - це комплекс програм для управління і координації роботи всіх пристроїв системи, управління процесом виконання прикладних програм і забезпечення діалогу з користувачем.

Не існує операційних систем жорсткого або м'якого реального часу. Поняття системи реального часу і операційної системи реального часу (ОС РЧ) часто змішуються.

Система реального часу - це конкретна обчислювальна система, пов'язана з реальним об'єктом. Вона включає в себе необхідні апаратні засоби, операційну систему і прикладне програмне забезпечення.

Операційна система реального часу - це тільки інструмент, що допомагає побудувати конкретну систему реального часу. Тому безглуздо говорити про операційні системи жорсткого або м'якого реального часу. Можна говорити тільки про те, чи можна за допомогою даної операційної системи побудувати систему реального часу. Конкретна ОС РЧ може тільки надати можливість створити систему жорсткого реального часу. Але володіння такою ОС РЧ зовсім не робить систему "жорсткою". Для створення системи жорсткого реального часу необхідне поєднання відповідних апаратних засобів, адекватної операційної системи і правильного проектування прикладного програмного забезпечення.

Визначимо операційну систему реального часу як операційну систему, за допомогою якої можна побудувати систему жорсткого реального часу. Обов'язкові вимоги до ОС РЧ:

Вимога 1: ОС РЧ повинна бути багатонітіевою або багатозадачною і підтримувати диспетчеризацію з витісненням.

Поведінка ОС РЧ має бути передбачуваною. Це не означає, що ОС РЧ повинна бути швидкою, але означає, що максимальний проміжок часу для виконання будь-якої операції повинен бути відомий заздалегідь і повинен бути узгоджений з вимогами програми. Наприклад, Windows 3.11 - навіть на процесорі Pentium Pro з тактовою частотою 200 МГц - непридатна для побудови систем реального часу, оскільки один додаток може назавжди захопити управління і заблокувати всі інші програми.

Перша вимога полягає в тому, щоб така ОС була багатонітієвою або багатозадачною і, крім того, планувальник повинен мати можливість витісняти будь-яку нитку (завдання) і передавати управління тій нитці (задачі), яка найбільше цього потребує. Для забезпечення витіснення на рівні переривань структура обслуговування переривань (в тому числі і апаратна архітектура) повинна бути багаторівневою.

Вимога 2: Має існувати поняття пріоритету нитки (завдання). Як знайти нитку (завдання), яка має потребу в ресурсах найбільше? В ідеальному випадку ОС РЧ надає ресурси тому завданню або драйверу, у яких залишилося менше всього часу до закінчення терміну реакції на подію (назвемо таку ОС - ОС керуючої критичними термінами). Однак для реалізації цього механізму потрібно вміти прогнозувати, скільки часу знадобиться задачі для завершення своєї роботи і скільки часу знадобиться іншим завданням для того, щоб вони встигли до своїх критичних термінів. Подібна ОС РЧ поки ще не створена через складність реалізації. Тому розробники ОС використовують інший метод: вони вводять концепцію пріоритетів для ниток (завдань).

При побудові конкретної системи реального часу розробник повинен вибудувати пріоритети завдань таким чином, щоб кожне з них встигало з реакцією до свого критичного терміну, тобто він повинен трансформувати базову вимогу реального часу "встигнути з реакцією до потрібного моменту" в комбінацію пріоритетів і в сценарій їх динамічної зміни. Очевидно, що при цій трансформації можливі помилки, що призводять до неправильної роботи системи. Для вирішення цього питання використовують різні теорії, такі як,

теорію монотонного планування або різні методи і засоби моделювання. Однак, ці методи виявляються не завжди ефективними. Як би там не було, у всіх сучасних ОС РЧ доводиться використовувати механізм пріоритетів як один з інструментів передбачуваності поведінки системи. На даний момент немає іншого рішення, поняття пріоритету потоку для систем реального часу неминуче.

Вимога 3: ОС повинна підтримувати передбачувані механізми синхронізації ниток (завдань). Всі нитки (завдання) поділяють дані (ресурси) і повинні обмінюватися між собою інформацією, тому необхідні механізми міжзадачної взаємодії.

Вимога 4: Повинен існувати механізм успадкування пріоритетів (система повинна бути захищена від інверсії пріоритетів). Під інверсією пріоритетів будемо розуміти зміну їх звичайного порядку. Насправді саме ці механізми синхронізації і той факт, що різні нитки виконуються в одному і тому ж просторі пам'яті, і визначають відмінність між нитками і процесами. Процеси не поділяють один і той же простір пам'яті.

Комбінації пріоритетів ниток і поділ між ними ресурсів призводить до класичної проблеми інверсії пріоритетів. Для створення умови інверсії пріоритетів повинно бути задіяно як мінімум три нитки. Якщо нитка з найнижчим пріоритетом заблокувала ресурс (який вона ділить з самою високопріоритетною ниткою), в той час як працює нитка з проміжним пріоритетом, виникає наступний ефект: нитка з найвищим пріоритетом очікує звільнення ресурсу; нитка з проміжним пріоритетом витісняє фонову нитку і працює, поки не завершиться; управління отримує фонова нитка, яка звільняє ресурс, і тільки після цього нитка з високим пріоритетом може продовжити свою роботу. В цьому випадку час, необхідний для завершення нитки з найвищим пріоритетом, залежить від часу роботи нитки з більш низьким пріоритетом - це і є інверсія пріоритетів. Очевидно, що в такій ситуації високопріоритетна нитка може "проґавити" критичну подію.

Щоб уникнути таких ситуацій, ОС РЧ повинна бути забезпечена механізмом успадкування пріоритетів, тобто блокуюча нитка повинна наслідувати пріоритет нитки, яку вона блокує (звичайно, тільки, в тому випадку, якщо заблокована нитка має більш високий пріоритет). Поведінка ОС має бути передбачуваною. Спадкування означає, що блокуючий ресурс (тред) успадковує пріоритет треду, який він блокує (це справедливо лише в тому випадку, якщо тред, що блокується, має більш високий пріоритет).

Тут є ще одна проблема: кількість можливих пріоритетів визначає обрана ОС РЧ. Більшість сучасних ОС РЧ допускають використання як максимум 256 пріоритетів. Чим більше пріоритетів в розпорядженні проектувальника, тим більш передбачувану систему можна створити. Тому при проектуванні системи різним ниткам бажано присвоювати різні пріоритети.

Розглянемо часові вимоги до операційних систем. Розробник повинен знати всі часи виконання системних викликів і вміти передбачати поведінку системи в будь-яких ситуаціях. Тому розробник ОС РЧ обов'язково повинен давати інформацію про наступні часові характеристиках системи:

- затримці переривання (interrupt latency) - тобто час від моменту появи запиту на переривання до початку його обробки;
- максимальному часу виконання кожного системного виклику. Він повинен бути передбачуваним і не залежати від кількості об'єктів в системі;
- максимальному часу, на який ОС і драйвери можуть блокувати переривання.

Розробник також повинен знати і враховувати наступне:

- рівні системних переривань;
- рівні переривань пристроїв, максимальний час, який займають програми обробки переривань, і т.д.

Якщо всі перераховані вище часові характеристики відомі, є всі передумови для створення системи жорсткого реального часу. При цьому вимоги до продуктивності системи мають бути узгоджені з характеристиками обраної ОС РЧ і апаратури.

Ті місця в програмах, в яких відбувається звернення до критичних ресурсів, називаються критичними секціями. Вирішення цієї проблеми полягає в організації такого доступу до критичного ресурсу, коли тільки одному процесу дозволяється входити в критичну секцію.

Ресурси, які не допускають одночасного використання декількома процесами, називаються критичними. Якщо кільком обчислювальним ресурсам необхідно користуватися критичним ресурсом в режимі поділу, їм слід синхронізувати свої дії таким чином, щоб ресурс завжди знаходився в розпорядженні не більше ніж одного з процесів.

Будь-яка система реального часу взаємодіє із зовнішнім світом через апаратуру комп'ютера. Зовнішні події перетворюються в переривання і обробляються драйвером пристрою. Проблема критичних секцій програм та синхронізації процесів, як правило. вивчається в курсах звичайних операційних систем і в подальшому розглядатись не буде.

Доступ до апаратури мають тільки драйвери. Оскільки додатки реального часу часто працюють зі специфічними зовнішніми пристроями, які вимагають і специфічного управління, розробник системи реального часу повинен вміти розробляти драйвери пристроїв.

У ОС РЧ розробник в першу чергу дізнається, на які пріоритети працюють драйвери інших пристроїв. Тут зазвичай існує вільний простір для переривань з пріоритетами, які вище пріоритетів стандартних драйверів.

Вимога 5: Політика управління пам'яттю в ОС РЧ. При проектуванні системи реального часу необхідно розглянути і інше важливе питання: як побудувати політику управління пам’яттю в ОС РЧ. Від вирішення цієї проблеми багато в чому залежить продуктивність створюваної системи.

1.3. Багатозадачність (multitasking)

Багатозадачність - властивість операційної системи забезпечувати можливість паралельної (або псевдопаралельної) обробки декількох завдань.

По суті, істинної багатозадачності немає в більшості ОС. Проходить лише перемикання між завданнями, але за рахунок того, що ці перемикання відбуваються досить часто, у користувача створюється враження, що додатки виконуються одночасно. Справжня багатозадачність операційної системи можлива тільки в багатопроцесорних / багатоядерних конфігураціях або в розподілених обчислювальних системах.

Багатозадачність призначена для підвищення ефективності використання системних ресурсів. Найчастіше під ефективністю розуміють:

- пропускну здатність - кількість завдань, виконаних за одиницю часу;

- зручність роботи користувача - можливість одночасно виконувати і управляти роботою декількох програм на одній ЕОМ;

- реактивність системи - здатність системи витримувати за раніше задані інтервали часу між запуском програми і по отриманню результатів.

Багатозадачність на однопроцесорної ЕОМ досягається наступним чином :

1. Завдання виконується до того, поки її виконання не буде перервано (ОС) або поки їй не доведеться чекати звільнення деякого ресурсу.

2. Зберігається контекст завдання.

3. Завантажується контекст іншої задачі.

4. Цей цикл повторюється до тих пір, поки є завдання, очікуючі виконання.

Багатозадачність збільшує обсяг роботи, що виконується системою (накладні витрати). Однак загальна ефективність роботи зростає, тому що більшість завдань не можуть виконуватися безперервно. Періодично завдання припиняє виконання і чекає, наприклад, поки повільний пристрій введення-виведення завершить пересилку даних або поки інше завдання використовує ресурс, необхідний цьому завданню. Завдяки багатозадачності, поки одна задача очікує, час процесора не витрачається даремно.

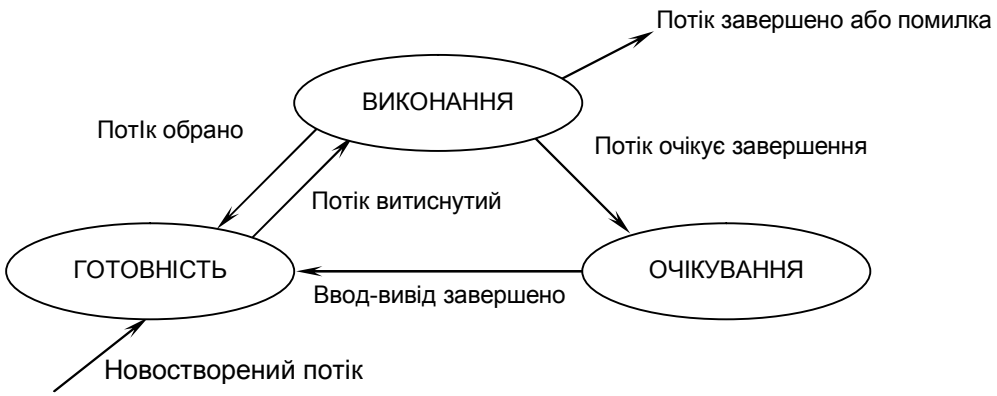


Рис.1.1. Реалізація багатозадачності

Однією з різновидів багатозадачності є багатопроцесорна обробка (multiprocessing) - виконання декількох задач на декількох процесорах. ОС з багатопроцесорною обробкою поділяються на дві категорії - з асиметричною або симетричною обробкою Операційні системи з асиметричною багатопроцесорною обробкою (asymmetric multiprocessing, ASMP) обирають для виконання власного програмного коду один і той же процесор, в той час як інші процесори виконують тільки призначені для користувача завдання.

Переваги ASMP: просто створити, удосконаливши існуючу однопроцесорну ОС; ідеально підходять для роботи на асиметричному обладнанні (співпроцесор).

Недоліки: важко зробити ОС переносною (відмінності за типомі ступенями асиметрії апаратури).

Системи з симетричною багатопроцесорною обробкою (Symmetric multiprocessing, SMP) дозволяють коду ОС виконуватися на будь-якому вільному процесорі або на всіх процесорах одночасно, причому кожному з процесорів доступна вся пам'ять.

Переваги SMP: повніше реалізуються можливості декількох процесорів і збільшується продуктивність, так як сама ОС може використовувати значну частину процесорного часу комп'ютера в залежності від того, які програми на ньому виконуються.

Системи SMP скорочують час простою через несправності, оскільки при відмові одного процесора код ОС може виконуватися на інших. Є можливість створення переносних ОС SMP.

Щоб підтримувати багатозадачність, ОС повинна визначити і оформити для себе ті внутрішні одиниці роботи, між якими буде розділятися процесор і інші ресурси комп'ютера. В даний час в більшості операційних систем визначені два типа одиниць роботи. Більша одиниця роботи, зазвичай носить назву **процесу**, вимагає для свого виконання кількох більш дрібних робіт, для позначення яких використовують терміни **«потік »,** або **« нитка ».**

Примітка. При використанні цих термінів часто виникають складнощі. Це відбувається з кількох причин. По-перше, із специфіки різних ОС, коли однакові по суті поняття отримали різні назви, наприклад завдання (task) в OS / 2, OS / 360 і процес (process) в UNIX, Windows NT, NetWare. По-друге, у міру розвитку системного програмування і методів організації обчислень деякі з цих термінів отримали нове смислове значення, особливо це стосується поняття «процес», який поступився багатьма своїми властивостями новому поняттю «потік». По-третє, термінологічні складності породжуються наявністю декількох варіантів перекладу англомовних термінів на українську мову. Наприклад, термін «Thread» перекладається як «нитка», «потік», «полегшений процес», «мінізадача» і т.п.

Далі як назва одиниці роботи в ОС РЧ будуть використовуватися терміни «процес» і «потік». У тих же випадках, коли різниці між цими поняттями не гратимуть суттєвої ролі, вони будуть об'єднуватися під узагальненим терміном «завдання» або «задача».

1.4. Основні поняття систем реального часу

Розглянемо більш детально суть основних понять, які використовуються в системах реального часу. Одним з основних понять даної дисципліни є поняття процесу.

Процес - це динамічна сутність програми, її код, що виконується.

В даному випадку під програмою розуміється опис на деякому формалізованому мовою алгоритму, поставленої задачі, що вирішується.

Програма є статичною одиницею, тобто незмінною з точки зору операційної системи, що її виконує.

Процес має:

- власні області пам'яті, де здійснюється зберігання коду та даних;
- власний стек;
- власне відображення віртуальної пам'яті на фізичну (в системах з віртуальною пам'яттю);
- власний стан.

Процес може перебувати в одному з наступних типових станів (точна кількість і властивості того чи іншого стану залежать від операційної системи):

1) «зупинений» - процес зупинений і не використовує процесор; наприклад, в такому стані процес знаходиться відразу після створення;

2) «термінований» - процес термінований і не використовує процесор; наприклад, процес закінчився, але ще не видалений операційною системою;

3) «чекає» - процес чекає деякої події (яким може бути апаратне або програмне переривання, сигнал або інша форма міжпроцесорної взаємодії);

4) «готовий» - процес не зупинено, ще не терміновано, не очікує, не видалений, але і не працює; наприклад, процес може не отримувати доступу до процесора, якщо в даний момент виконується інший, більш пріоритетний процес;

5) «виконується» - процес виконується і використовує процесор; в ОС РЧ це зазвичай означає, що цей процес є пріоритетним серед всіх процесів, що знаходяться в стані «готовий».

Поняття обчислювального процесу (або просто «процесу») було введено для реалізації ідей мультипрограмування і мультизадачности. Як поняття процес є певною абстракцією. Послідовний процес (іноді називають «завданням або задачею») - це виконання окремої програми з її даними на послідовному процесорі.

У концепції, що отримала широке поширення в 70 - ті роки, під завданням розуміли сукупність пов'язаних між собою модулів і даних, що утворюють єдине ціле і вимагають певних ресурсів обчислювальної системи для своєї реалізації. У наступні роки завданням стали називати одиницю роботи, для виконання якої надається центральний процесор. Обчислювальний процес може включати в себе кілька завдань.

Концептуально процес розглядається в двох аспектах: по - перше, він є носієм даних і, по - друге, він одночасно виконує операції, пов'язані з їх обробкою. Визначення концепції процесу має на меті виробити механізми розподілу і управління ресурсами. Поняття ресурсу, так само як і поняття процесу, є одним з основних при розгляді операційних систем реального часу. Термін ресурс зазвичай застосовується по відношенню до повторно використовуваних, відносно стабільних і часто відсутніх об'єктів, які запитуються, використовуються і звільняються процесами в період їх активності. Іншими словами, ресурсом називається будь-який об'єкт, який може розподілятися всередині системи. Ресурс - це об'єкт, необхідний для роботи процесу або завданню. Ресурси можуть бути розподілювані, коли кілька процесів можуть їх використовувати одночасно (в один і той же момент часу) або паралельно (протягом деякого інтервалу часу процеси використовують ресурс поперемінно), а можуть бути і неподільними.

При розробці перших систем ресурсами вважалися процесорний час, пам'ять, канали введення / виводу і периферійні пристрої. Однак, з плином часу поняття ресурсу стало набагато більш універсальним і загальним. Різного роду програмні та інформаційні ресурси також можуть бути визначені для системи як об'єкти, які можуть розділятися і розподілятися, і доступ до яких, необхідно відповідним чином контролювати. В даний час поняття ресурсу перетворилося в абстрактну структуру з цілим рядом атрибутів, що характеризують способи доступу до цієї структури і її фізичне представлення в системі.

У перших обчислювальних системах будь-яка програма могла виконуватися тільки після повного завершення попередньої програми. Оскільки перші

обчислювальні системи були побудовані відповідно до принципів фон Неймана, всі підсистеми і пристрої комп'ютера керувалися виключно центральним процесором. Центральний процесор здійснював і виконання обчислень, і управління операціями вводу / виводу даних. Відповідно, поки здійснювався обмін даними між оперативною пам'яттю і зовнішніми пристроями, процесор не міг виконувати обчислення. Введення до складу обчислювальної машини спеціальних контролерів дозволило поєднати в часі (розпаралелити) операції виведення отриманих даних і наступні обчислення на центральному процесорі. Однак, процесор продовжував часто і довго простоювати, чекаючи завершення чергової операції введення / виводу. Тому було запропоновано організувати мультипрограмний (мультизадачність) режим роботи обчислювальної системи. Суть його полягає в тому, що поки одна програма (один обчислювальний процес або завдання) очікує завершення чергової операції введення / виводу, інша програма (а точніше, інша задача) може бути поставлена на реалізацію.

Якщо в операційній системі можуть одночасно існувати кілька процесів або / і завдань, які перебувають у стані «виконується», то кажуть, що це багатозадачна система, а ці процеси називають паралельними.

Якщо процесор один, то в кожен момент часу насправді реально виконується тільки один процес або завдання. Система розділяє час між такими, що «виконуються» процесами, даючи кожному квант часу, пропорційний його пріоритету. Цей квант часу часто не залежить від специфіки розв'язуваної задачі реального часу, тому такий підхід зазвичай не використовується в СРЧ. Зазвичай в СРЧ в стані виконання може бути тільки один процес. У «хорошої» СРЧ це можна змінити програмним шляхом.

Завдяки поєднанню в часі виконання двох програм загальний час виконання двох завдань виходить менше, коли б ми виконували їх по черзі (запуск тільки одного завдання після повного завершення іншого). Але час виконання кожного завдання в загальному випадку стає більше, ніж, якби ми виконували кожен з них як єдиний по черзі.

При мультипрограмуванні підвищується пропускна здатність системи, але окремий процес ніколи не може бути виконаний швидше, ніж, коли б він виконувався в однопрограмному режимі (будь-який поділ ресурсів уповільнює роботу одного з учасників за рахунок додаткових витрат часу на очікування звільнення ресурсу).

Система підтримує мультипрограмування і намагається ефективно використовувати ресурси шляхом організації до них черг запитів, що складаються в той чи інший спосіб. Ця вимога досягається підтриманням в пам'яті більше одного процесу, який чекає процесор, і більше одного процесу, готового використовувати інші ресурси, як тільки останні стануть доступними. Загальна схема виділення ресурсів така. При необхідності використовувати будь-який ресурс (оперативну пам'ять, пристрій введення / виведення, масив даних і т.п.), завдання звертається до супервізору операційної системи - її центральному керуючому модулю, який може складатися з декількох модулів, наприклад: супервизор введення / виведення, супервізор переривань, супервізор програм, диспетчер задач і т.д. - за допомогою спеціальних викликів (команд, директив) і повідомляє про свою вимогу. При цьому вказується вид ресурсу і, якщо треба, його обсяг. Директива звернення до операційної системи передає їй управління, переводячи процесор в привілейований режим роботи, який обов'язково існує в СРЧ.

Ресурс може бути виділений задачі, яка звернулася до супервізора з відповідним запитом, якщо:

- він вільний і в системі немає запитів від завдань більш високого пріоритету до цього ж ресурсу;
- поточний запит і раніше видані запити допускають спільне використання ресурсів;
- ресурс використовується завданням нижчого пріоритету і може бути тимчасово відібраний (колективні ресурси).

Отримавши запит, система або задовольняє його і повертає управління завданню, що видало цей запит, або, якщо ресурс зайнятий, ставить задачу в чергу до ресурсу, переводячи її в стан очікування (блокуючи).

Після закінчення роботи з ресурсом завдання знову за допомогою спеціального виклику супервізора повідомляє операційній системі про відмову від ресурсу, або операційна система забирає ресурс сама, якщо управління повертається супервізору після виконання будь-якої системної функції. Супервізор операційної системи, отримавши управління за цим зверненням, звільняє ресурс і перевіряє, чи є черга до звільненого ресурсу. Якщо черга є - в залежності від прийнятої дисципліни обслуговування і пріоритетів заявок він виводить зі стану очікування завдання, чекає ресурс, і переводить її в стан готовності до виконання. Після цього управління або передається цьому завданню, або повертається до тієї, яка тільки що звільнила ресурс.

У загальному випадку при організації управління ресурсами в СРЧ завжди потрібно прийняти рішення про те, що в даній ситуації доцільніше: швидко обслуговувати окремі найбільш важливі запити, надавати всім процесам рівні можливості, або обслуговувати максимально можливу кількість процесів і найбільш повно використовувати ресурси.

Стек (stack) - це область пам'яті, в якій розміщуються локальні змінні, аргументи і повертаються значення функцій. Разом з областю статичних даних повністю задає поточний стан процесу.

Віртуальна пам'ять - це «пам'ять», в адресному просторі якої працює процес. Віртуальна пам'ять:

1) дозволяє збільшити обсяг пам'яті, доступної процесам за рахунок дискової пам'яті;

2) забезпечує виділення кожному з процесів віртуально безперервного блоку пам'яті, що починається (віртуально) з однієї й тієї ж адреси;

3) забезпечує ізоляцію одного процесу від іншого.

Трансляцією віртуальної адреси (ВА) в фізичну адресу займається операційна система. Для прискорення цього процесу багато комп'ютерні

системи мають підтримку з боку апаратури, яка може бути або прямо в процесорі, або в спеціальному пристрої управління пам'яттю. Серед механізмів трансляції ВА переважає сторінковий, при якому віртуальна і фізична пам'ять розбиваються на частини рівного розміру, звані сторінками (типовий розмір - 4Кб), між сторінками віртуальної і фізичної пам'яті встановлюється взаємно однозначне (для кожного процесу) відображення. Відзначимо, що ОС РЧ прагнуть отримати максимальну продуктивність на наявному обладнанні, тому деякі ОС РЧ не використовують механізм віртуальної пам'яті через затримки, що вносяться при трансляції адреси.

Міжпроцесна взаємодія - це той чи інший спосіб передачі інформації з одного процесу в інший. Найбільш поширеними формами взаємодії процесів є (не всі системи підтримують такі можливості):

1) колективна пам'ять - два (або більше) процеси мають доступ до одного і того ж блоку пам'яті. У системах з віртуальною пам'яттю організація такого виду взаємодії вимагає підтримки з боку операційної системи, оскільки необхідно відобразити відповідні блоки віртуальної пам'яті процесів на один і той же блок фізичної пам'яті;

2) семафори - два (або більше) процесів мають доступ до однієї змінної, що приймає значення 0 або 1. Сама змінна часто знаходиться в області даних операційної системи і доступ до неї організовується за допомогою спеціальних функцій;

3) сигнали - це повідомлення, що доставляються за допомогою операційної системи процесу. Процес повинен зареєструвати обробник цього повідомлення у операційної системи, щоб отримати можливість реагувати на нього. Часто операційна система сповіщає процес сигналом про настання якого-небудь збою, наприклад, поділ на 0, або при будь-якому апаратному перериванні, наприклад, переривання таймера;

4) поштові скриньки - це черга повідомлень (зазвичай тих чи інших структур даних), які поміщаються в поштову скриньку процесами і / або операційною системою. Кілька процесів можуть чекати надходження повідомлення в

поштову скриньку і активізуватися після його надходження. Необхідна підтримка з боку операційної системи.

Подія - це оповіщення процесу з боку операційної системи у тій чи іншій формі про взаємодію між процесами, наприклад, про прийняття семафором потрібного значення, про наявність сигналу, про надходження повідомлення в поштову скриньку.

Створення, забезпечення взаємодії, розподіл процесорного часу вимагає від операційної системи значних обчислювальних витрат, особливо в системах з віртуальною пам'яттю. Це пов'язано, перш за все, з тим, що кожен процес має своє відображення віртуальної пам'яті на фізичну, яке треба міняти при перемиканні процесів і при забезпеченні їх доступу до об'єктів взаємодії (спільної пам'яті, семафорам, поштовим скринькам). Часто буває так, що потрібно запустити кілька копій однієї і тієї ж програми, наприклад, для управління декількома одиницями одного і того ж обладнання. В цьому випадку ми несемо подвійні накладні витрати: тримаємо в оперативній пам'яті кілька копій коду однієї програми і витрачаємо додатковий час на забезпечення їх взаємодії.

Завдання (або потік, або нитка, thread) - це як би одна з гілок виконання процесу:

- розділяє з процесом область пам'яті під код і дані;
- має власний стек;
- розділяє з процесом відображення віртуальної пам'яті на фізичну (в системах з віртуальною пам'яттю);
- має власний стан.

Таким чином, у двох завдань в одному процесі вся пам'ять розділяється і додаткові витрати, пов'язані з різним відображенням віртуальної пам'яті на фізичну, зведені до нуля. Для завдань так само, як для процесів, визначаються поняття стану завдання і міжзадачної взаємодії. Відзначимо, що для двох процесів зазвичай потрібно організувати щось спільне (пам'ять, канал і т.д.) для

їх взаємодії, в той час як для двох потоків часто потрібно організувати щось спільне (наприклад, область пам'яті), що має своє значення в кожному з них.

Пріоритет - це число, приписане операційною системою кожному процесу і задачі. Чим більше це число, тим важливіше цей процес або завдання і тим більше процесорного часу він або вона отримає. При неправильному плануванні пріоритетів в ОС РЧ, завдання з меншим пріоритетом може взагалі не отримати управління при наявності в стані готовності завдання з великим пріоритетом.

Зв'язування (лінкування, linkage) - це процес перетворення скомпільованого коду (об'єктних модулів) в завантажувальний модуль (тобто те, що може виконуватися процесором за підтримки операційної системи). Розрізняють:

- статичне зв'язування, коли код, необхідний для роботи програми бібліотечних функцій фізично додається до коду об'єктних модулів для отримання завантажувального модуля;
- динамічне зв'язування, коли в результуючому завантажувальному модулі проставляються лише посилання на коди необхідних бібліотечних функцій; сам код буде реально доданий до завантажувального модуля тільки при його виконанні.

При статичному зв'язуванні завантажувальні модулі виходять дуже великого розміру. Тому переважна більшість сучасних операційних систем використовує динамічне зв'язування, незважаючи на те, що при цьому початкове завантаження процесу на виконання повільніше, ніж при статичному зв'язуванні через необхідність пошуку і завантаження коду потрібних бібліотечних функцій (часто тільки тих з них, які не були завантажені для інших процесів). При цьому для уникнення недетермінованої затримки на завантаження програми і на виконання всіх необхідних процесів реального часу їх запускають при старті системи (заздалегідь, а не на вимогу).

1.5. Класи задач систем реального часу

Всякий процес містить одну або кілька завдань (задач). Операційна система дозволяє завданню породжувати нові завдання. Завдання (задачі), за своїми особливостями та способами виконання, можна розділити на 3 категорії:

1. Циклічні завдання (задачі). Характерні для процесів управління та інтерактивних процесів.
2. Періодичні завдання. Характерні для багатьох технологічних процесів і завдань синхронізації.
3. Імпульсні завдання. Характерні для задач сигналізації і асинхронних технологічних процесів.

Щоб система могла керувати завданнями, вона повинна мати всю необхідну для цього інформацію. З цією метою на кожну задачу (процес) заводиться спеціальна інформаційна структура, яка називається дескриптором процесу (описувачем завдання). У загальному випадку дескриптор процесу містить наступну інформацію:

- ідентифікатор процесу (так званий PI - process identificator);
- тип (або клас) процесу, який визначає для супервізора деякі правила надання ресурсів. Управління вводом / виводом здійснюється операційною системою, компонентом, який називають супервізором введення / виведення;
- пріоритет процесу, відповідно до якого супервізор надає ресурси. В рамках одного класу процесів в першу чергу обслуговуються більш пріоритетні процеси;
- змінну стану, яка визначає, в якому стані знаходиться процес (готовий до роботи, в стані виконання, очікування пристроїв введення / виводу і т.д.);
- захищену область пам'яті (або адреса такої зони), в якій зберігаються поточні значення регістрів процесу, якщо процес переривається, не закінчивши роботи. Ця інформація називається **контекстом завдання**;

- інформацію про ресурси, якими процес володіє і / або має право користуватися (покажчики на відкриті файли, інформація про незавершені операції введення / виводу і т.д.);
- місце (або його адреса) для організації спілкування з іншими процесами;
- параметри часу запуску (момент часу, коли процес повинен активізуватися, і періодичність цієї процедури);
- у разі відсутності системи управління файлами - адреса завдання на диску в її початковому стані і адреса на диску, куди інформація вивантажується з оперативної пам'яті, якщо її витісняє інше завдання.

Коли говорять про процеси, то тим самим хочуть підкреслити, що система підтримує їх відособленість: кожен процес має свій віртуальний адресний простір, кожному процесу призначаються свої ресурси - файли, вікна, семафори і т.д. Така відособленість потрібна для того, щоб захистити один процес від іншого, оскільки вони, спільно використовуючи всі ресурси обчислювальної системи, конкурують один з одним. У загальному випадку процеси (завдання) ніяк не пов'язані між собою і можуть належати навіть різним користувачам, що розділяє одну обчислювальну систему. Іншими словами, в разі процесів система вважає їх абсолютно не пов'язаними і не залежними. При цьому саме система бере на себе роль арбітра в конкуренції між процесами з приводу ресурсів.

Бажано мати можливість задіяти внутрішній паралелізм, який може бути в самих процесах. Такий внутрішній паралелізм зустрічається досить часто і його використання дозволяє прискорити їх виконання. У однопроцесорной системі завдання поділяють між собою процесорний час так само, як це роблять звичайні процеси, а в мультипроцесорної системі можуть виконуватися одночасно, якщо не зустрічають конкуренції через звернення до інших ресурсів.

Таким чином, головне, що забезпечує багатозадачність, - це можливість паралельно виконувати декілька видів операцій в одній прикладній програмі. Паралельні обчислення (а, отже, і більш ефективне використання ресурсів

центрального процесора, і менше сумарний час виконання завдань) тепер уже часто реалізуються на рівні завдань, і програма, оформлена у вигляді декількох завдань (потоків, ниток) в рамках одного процесу, може бути виконана швидше за рахунок паралельного виконання її окремих частин.

1.6. Контрольні запитання до розділу 1.

1. Дайте визначення системам реального часу.
2. Які вимоги до систем реального часу.
3. Які характерні відмінності систем м’якого і жорсткого реального часу.
4. Які особливості і додаткові вимоги до систем реального часу.
5. Дайте визначення поняттям “потік”, “процес”, “задача”.
6. Що визначає поняття “ресурс”.
7. Охарактеризуйте типи задач систем реального часу.
8. Для яких цілей системи реального часу використовують пріоритети.
9. Що означають терміни “подія” , та “міжпроцесна взаємодія ”.
10. Охарактеризуйте засоби зв’язку між процесами.
11. Що означає поняття “зв’язування”.
12. Які існують методи забезпечення багатозадачності.

Розділ 2 . Планування і диспетчеризація в системах реального часу

2.1. Типи планувальників

Протягом існування процесу виконання його потоків може бути багаторазово перерване і продовжене. Перехід від виконаннання одного потоку до іншого здійснюється в результаті планування і диспетчеризації. Робота по визначенню того, в який момент необхідно перервати виконання поточного активного потоку і якому потоку дати можливість виконуватися, називається **плануванням**.

Планування потоків здійснюється на основі інформації, що зберігається в дескрипторах процесів і потоків (пріоритет потоків, час їх очікування в черзі, накопичений час виконання, інтенсивність звернень до вводу-виводу і інші фактори). ОС планує виконання потоків незалежно від того, чи належать вони одному або різним процесам. Існує безліч різних алгоритмів планування, які можуть переслідувати різні цілі і в найбільшій мірі визначають специфіку операційної системи. Наприклад, в одному випадку вибирається такий алгоритм планування, при якому гарантується, що жоден потік / процес не буде займати процесор довше визначеного часу (ОС РЧ), в іншому випадку метою є максимально швидке виконання «коротких» завдань (спеціальні ОС), а в третьому випадку - переважне право зайняти процесор отримують потоки інтерактивних додатків (ОС загального призначення).

У більшості операційних систем універсального призначення планування здійснюється динамічно, тобто рішення приймаються під час роботи системи на основі аналізу поточної ситуації. ОС працює в умовах невизначеності - потоки і процеси з'являються в випадкові моменти часу і також непередбачено завершуються. Таке динамічне планування дає можливість гнучко пристосуватись до ситуації, що змінюється і не використовувати ніяких передбачень про мультипрограмні суміші. Для того, щоб оперативно знайти в

умовах такої невизначеності оптимальний за деяким критерієм план виконання завдань, ОС повинна витрачати значних зусиль на порядок виконання завдань.

Інший тип планування - статичний - може бути використаний в спеціалізованих системах, в яких весь набір одночасно виконуваних завдань визначено заздалегідь, наприклад, в деяких системах реального часу.

Результатом роботи статичного планувальника є таблиця, звана розкладом, в якій вказується, якому потоку / процесу, коли і на який час повинен бути надано процесор. Для побудови розкладу планувальнику потрібні якомога повніші попередні знання про характеристику набору завдань, наприклад, про максимальний час виконання кожного завдання, обмеження передування, обмеження щодо взаємного виключення, граничні терміни і т. д. Накладні витрати ОС на виконання розкладу виявляються значно меншими, ніж при динамічному плануванні, і зводяться лише до диспетчеризації потоків / процесів.

Диспетчеризація полягає в реалізації знайденого в результаті планування (динамічного чи статичного) рішення, тобто в перемиканні процесора з одного потоку на інший.

Диспетчеризація зводиться до наступного:

- збереження контексту поточного потоку, який потрібно змінити;
- завантаження контексту нового потоку, обраного в результаті планування;
- запуск нового потоку на виконання.

В контексті потоку можна виділити частину, загальну для всіх потоків даного процесу (посилання на відкриті файли), і частину, що стосується тільки до даного потоку (вміст регістрів, лічильник команд, режим процесора). Тому в деяких системах вводиться певна ієрархія контекстів. Наприклад, в середовищі NetWare 4.x розрізняються три види контекстів: глобальний контекст (контекст процесу), контекст групи потоків і контекст окремого потоку.

Очевидно, що така ієрархічна організація контекстів прискорює перемикання потоків, так як при цьому в межах однієї групи немає необхідності

замінювати контексти груп або глобальні контексти, досить лише замінити контексти потоків, які мають менший обсяг.

2.2. Витисняючі і невитисняючі алгоритми планування

З найзагальніших позицій вся безліч алгоритмів планування можна розділити на два класи: витисняючі і невитисняючі алгоритми планування.

Витисняючи (preemptive) алгоритми - це такі способи планування завдань, в яких рішення про переключення процесора з виконання одного завдання на виконання іншого приймається операційною системою, а не активним завданням.

Невитисняючі (non-preemptive) алгоритми засновані на тому, що активній задачі дозволяється виконуватися, поки вона сама з власної ініціативи не віддасть управління операційній системі для того, щоб та вибрала з черги інше готове до виконання завдання.

Основною відмінністю між витисняючими і невитисняючими алгоритмами є ступінь централізації механізму планування потоків. При витисняючому мультипрограмуванні функції планування цілком зосереджені в операційній системі і програміст пише свій додаток, не піклуючись про те, що вон буде виконуватися одночасно з іншими завданнями. При цьому операційна система виконує наступні функції: визначає момент зняття з виконання активного потоку, запам'ятовує його контекст, вибирає з черги готових потоків наступний, запускає новий потік на виконання, завантажуючи його контекст.

При невитисняючому мультипрограмуванні механізм планування розподілений між операційною системою і прикладними програмами, які, отримавши управління від операційної системи, самі визначають момент завершення чергового циклу свого виконання і тільки потім передають керування ОС за допомогою якого-небудь системного виклику. ОС формує черги потоків і вибирає відповідно до деякого правила (наприклад, з урахуванням пріоритету) наступний потік на виконання.

Переваги витисняючого алгоритму:

- спрощення розробки програм;
- незалежність роботи програм одна від одної;
- більш надійна робота системи в цілому.

Переваги невитисняючого алгоритму

- більш висока швидкість перемикання;
- можливість гнучкого планування перемикання завдань розробником програми;
- спрощення синхронізації (звернення до даних виконується тільки одним завданням).

Алгоритми планування, засновані на квантуванні В основі багатьох витисняючих алгоритмів планування лежить концепція квантування. Відповідно до неї кожному потоку почергово для виконання надається обмежений безперервний період процесорного часу - квант. Зміна активного потоку проходить, якщо:

- потік завершився і залишив систему;
- виникла помилка;
- потік перейшов в стан очікування;
- вичерпаний квант процесорного часу, відведений даному потоку.

Потік, який вичерпав свій квант, переводиться в стан готовності і чекає, коли йому буде надано новий квант процесорного часу, а на виконання відповідно до визначеного правила вибирається новий потік з черги готових потоків (рис. 2.1).

Кванти, що виділяються потокам, можуть бути однаковими для всіх потоків або різними. Розглянемо, наприклад, випадок, коли всім потокам надаються кванти однакової довжини q (рис. 2.2). Якщо в системі є n потоків, то час, який потік проводить в очікуванні наступного кванта, можна грубо оцінити як q (n -l). Чим більше потоків в системі, тим більше час очікування, тим менше

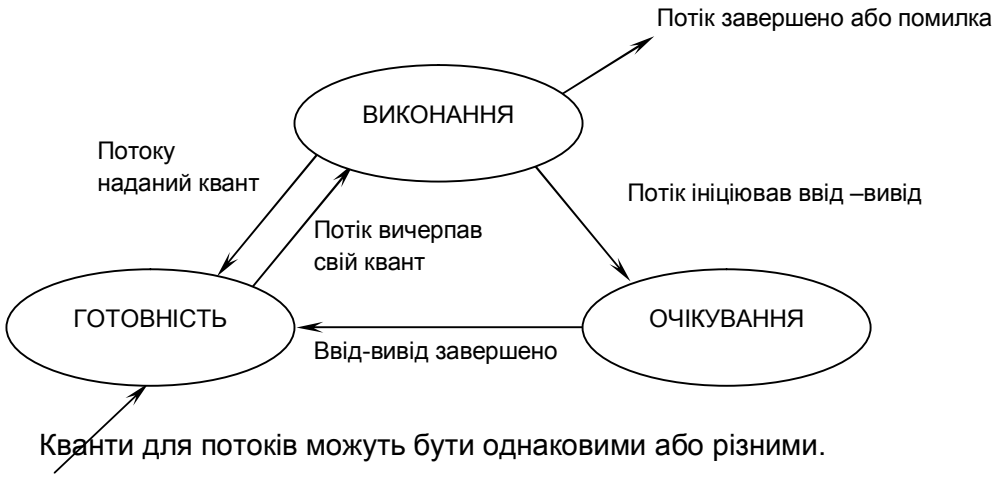


Рис 2.1. Граф станів потоку у системі з квантуванням

можливості вести одночасну інтерактивну роботу декільком користувачам. Але якщо величина кванта обрана дуже невеликою, то значення виразу q (n - l) все одно буде достатньо невеликою для того, щоб користувач не відчував дискомфорту від присутності в системі інших користувачів. Типове значення кванту в системах поділу часу становить десятки мілісекунд.

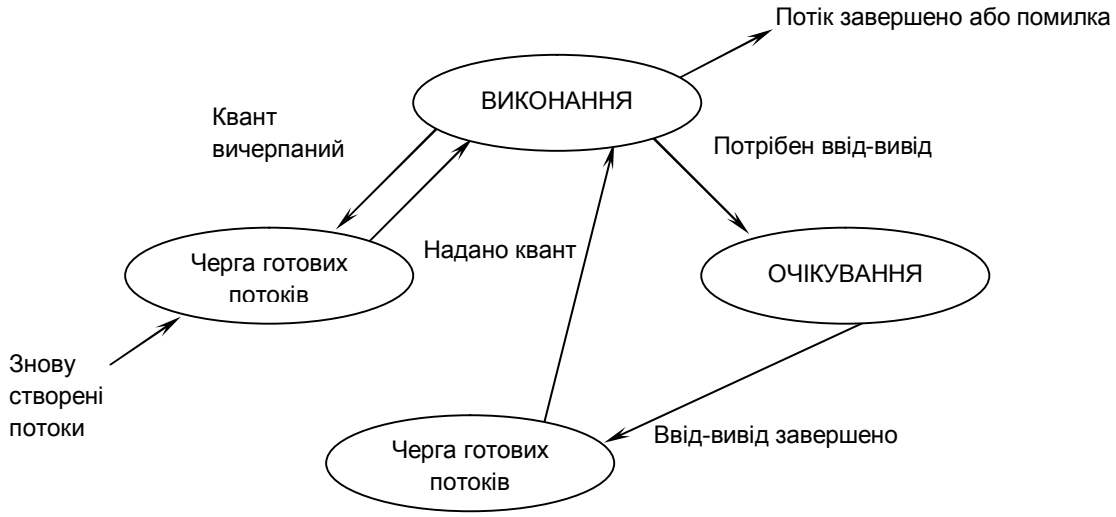


Рис.2.2. Всім потокам надаються кванти однакової довжини

Якщо квант короткий, то сумарний час, який проводить потік в очікуванні процесора (W), прямо пропорційно часу, необхідному для його виконання

(тобто часу, який необхідно було б для виконання цього потоку при монопольному використанні обчислювальної системи). Дійсно, оскільки час очікування між двома циклами дорівнює q (n-l), а кількість циклів B / q, де B - необхідний час виконання, то W = B (n - l). Зауважимо, що ці співвідношення є досить грубі оцінки, засновані на припущенні, що B значно перевищує q. При цьому не враховується, що потоки можуть використовувати кванти не повністю, що частину часу вони можуть витрачати на введення-виведення, що кількість потоків в системі може динамічно змінюватися і т. д.

Багатозадачні ОС втрачають деяку кількість процесорного часу для виконання допоміжних робіт під час переключення контекстів задач. Витрати на ці допоміжні дії не залежать від величини кванта часу, тому чим більше квант, тим менше сумарні накладні витрати, пов'язані з перемиканням потоків.

Модифікації алгоритму планування на основі квантування.

1. Динамічна величина кванта. Нехай, наприклад, попередньо кожному потоку призначається досить великий квант, а величина кожного наступного кванта зменшується до деякої наперед заданої величини. У такому випадку перевагу отримують короткі завдання, які встигають виконуватися протягом першого кванту, а тривалі обчислення будуть проводитися в фоновому режимі. Можна уявити собі алгоритм планування, в якому кожний наступний квант, що виділяється певному потоку, більше попереднього. Такий підхід дозволяє зменшити накладні витрати на перемикання завдань у тому випадку, коли відразу кілька задач виконують тривалі обчислення.

2. Компенсація невикористаного кванта. Потоки отримують для виконання квант часу, але деякі з них не використовують його в повному обсязі, наприклад через необхідність виконати введення або виведення даних. В результаті виникає ситуація, коли потоки з інтенсивними зверненнями до вводу-виводу використовують тільки невелику частину виділеного їм процесорного часу. Алгоритм планування може виправити цю «несправедливість». У якості компенсації за невикористані повністю кванти

потоки отримують привілеї при подальшому обслуговуванні. Для цього планувальник створює дві черги готових потоків (рис. 2.3). Черга 1 утворена потоками, які прийшли в стан готовності в результаті операція введення-виведення. При виборі потоку для виконання насамперед проглядається друга черга і, тільки, якщо вона порожня, квант виділяється потоку з першої черги.

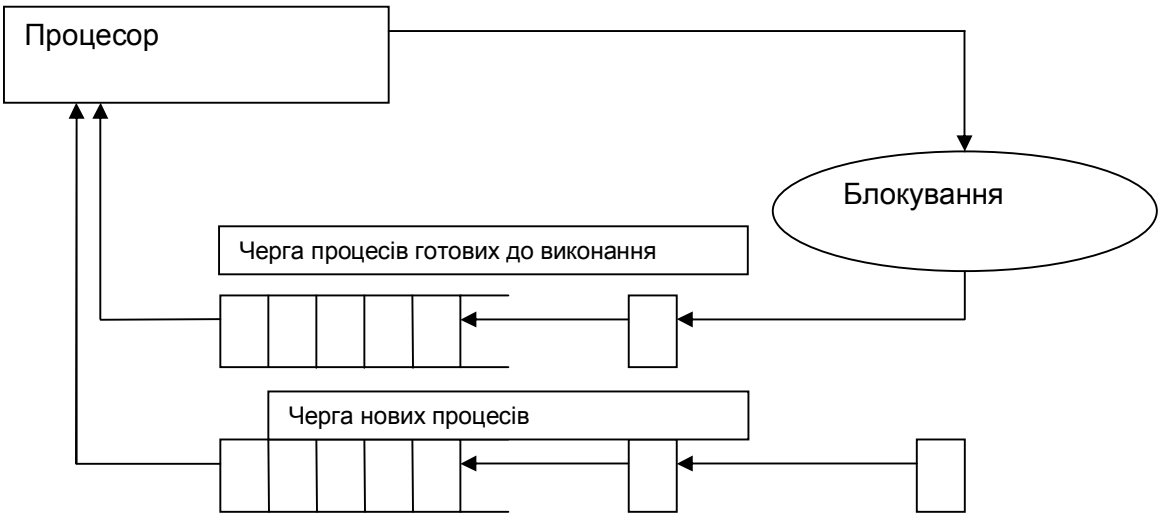


Рис. 2.3 Квантування з перевагами для потоків з інтенсивним вводом-виводом

2.3. Алгоритми планування, засновані на пріоритетах

Іншою важливою концепцією, що лежить в основі багатьох витисняючих алгоритмів планування, є пріоритетне обслуговування. Воно передбачає наявність у потоків деякої спочатку відомої характеристики - пріоритету, на підставі якої визначається порядок їх виконання. Пріоритет - це число, характеризує ступінь привілейованості потоку при використанні ресурсів обчислювальної машини, зокрема процесорного часу: чим вище пріоритет, тим вище привілеї, тим менше проводитиме потік в чергах.

Пріоритет може виражатися цілим або дробовим, позитивним або від'ємним значенням. У деяких ОС прийнято, що пріоритет потоку тим вище, чим більше

(в арифметичному сенсі) число, що позначає пріоритет. В інших системах, навпаки, чим менше число, тим вище пріоритет.

У більшості операційних систем, що підтримують потоки, пріоритет потоку безпосередньо пов'язаний з пріоритетом процесу, в рамках якого виконується даний потік. Пріоритет процесу призначається операційною системою при його створенні. Значення пріоритету включається в дескриптор процесу і використовується при визначенні пріоритету потокам цього процесу. При призначенні пріоритету новоствореному процесу ОС враховує:

1) чи є цей процес системним, чи прикладним,

2) який статус користувача, що запустив процес,

3) чи була явна вказівка користувача на привласнення процесу певного рівня пріоритету,

4) що потік може бути ініційований не тільки по команді користувача, але і в результаті виконання системного виклику іншим потоком. У цьому випадку при призначенні пріоритету новому потоку ОС повинна приймати до уваги значення параметрів системного виклику.

У багатьох ОС передбачається можливість зміни пріоритету протягом життя потоку. Зміни пріоритету можуть відбуватися:

1) з ініціативи самого потоку, коли він звертається з відповідним викликом до операційної системи,

2) по ініціативі користувача, коли він виконує відповідну команду,

3) крім того, ОС сама може змінювати пріоритети потоків у залежності від ситуації, що складається в системі. В останньому випадку пріоритети називаються динамічними на відміну від незмінних, фіксованих, пріоритетів.

Існують два різновиди пріоритетного планування: обслуговування з відносними пріоритетами і обслуговування з абсолютними пріоритетами. В обох випадках вибір потоку на виконання з черги готових здійснюється однаково: вибирається потік, що має найвищий пріоритет. Однак проблема визначення моменту зміни активного потоку вирішується по-різному.

1. У системах з відносними пріоритетами активний потік виконується до тих пір, поки він сам не покине процесор, перейшовши в стан очікування, або ж станеться помилка, або потік завершиться (рис. 2.4).

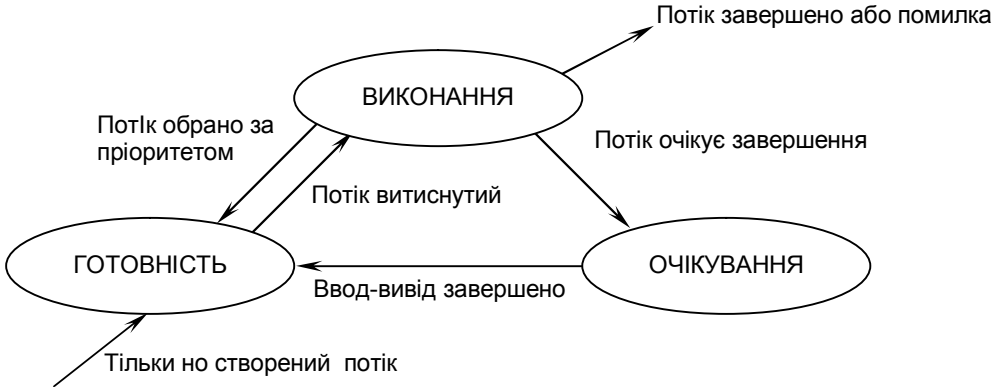


Рис. 2.4. Граф стану потоків в системі з відносними пріоритетами

2. У системах з абсолютними пріоритетами виконання активного потоку переривається, крім зазначених вище причин, ще за однієї умови: якщо в черзі готових потоків з'явився потік, приоритет якого вище пріоритету активного потоку. В цьому випадку перерваний потік переходить в стан готовності (рис. 2.5).

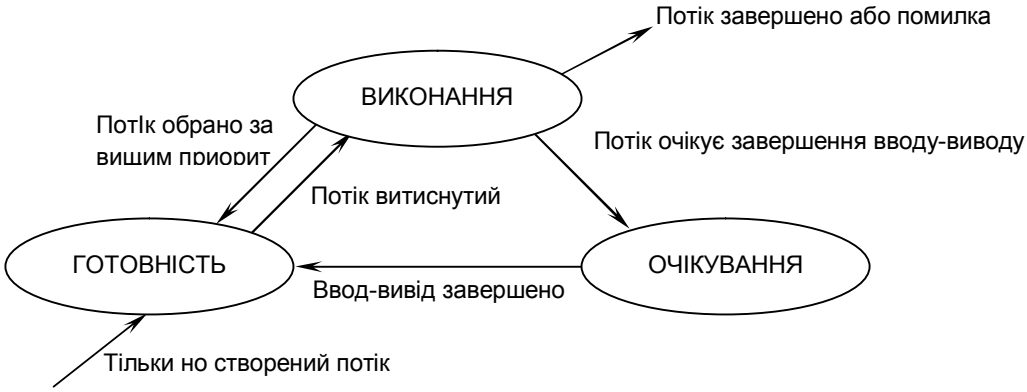


Рис. 2.5. Граф стану потоків в системі з абсолютними пріоритетами

Змішані алгоритми планування. У багатьох операційних системах алгоритми планування побудовані з використанням як концепції квантування, так і пріоритетів. Наприклад, в основі планування лежить квантування, але

величину кванта і / або порядок вибору потоку з черги готових визначається пріоритетами потоків.

Windows NT: квантування поєднується з динамічними абсолютними пріоритетами. На виконання вибирається готовий потік з найвищим пріоритетом. Йому виділяється квант часу. Якщо під час виконання в черзі готових з'являється потік з більш високим пріоритетом, то він витісняє виконуваний потік. Останній повертається в чергу готових, причому він стає попереду всіх інших потоків, що мають такий же пріоритет.

UNIX System V Release 4: витісняюча багатозадачність, основана на використанні пріоритетів і квантування. Кожен процес в залежності від завдання, яке він вирішує, відноситься до одного з трьох визначених у системі пріоритетних класів: класу «реального часу» (фіксовані пріоритети, але користувач може їх змінювати), класу системних процесів (фіксовані пріоритети) або класу процесів поділу часу (динамічні пріоритети).

Характеристики планування процесів реального часу включають дві величини: рівень глобального пріоритету і квант часу. Для кожного рівня пріоритету за умовчанням є своя величина кванта часу. Процесу дозволяється захоплювати процес на вказаний квант часу, а по його закінченні планувальник знімає процес з виконання.

2.4. Контрольні запитання до розділу 2

1. Які існують планувальники для систем реального часу.

2. У чому різниця між витисняючими і не витисняючими алгоритмами планування.

3. Які особливості планувальників, заснованих на пріоритетах.

4. Чим відрізняються планувальники, що використовують абсолютні і відносні пріоритети.

5. Які особливості алгоритмів планування, що засновані на квантуванні.

6. Які переваги та недоліки витисняючих і невитисняючих алгоритмів планування.

7. Який зв’язок між пріоритетами процесу і потоку.

8. Яка різниця між статичними і динамічними планувальниками у системах реального часу.

9. Який результат роботи статичного планувальника.

10. Які алгоритми планування переважно використовують у системах реального часу і чому.

Розділ 3. Обмін інформацією між процесами

3.1. Загальні області пам’яті

Взаємодіючі процеси потребують обміну інформацією. Тому багатозадачна операційна система повинна забезпечувати необхідні для цього засоби. Обмін даними повинен бути прозорим для процесів, тобто передані дані не повинні змінюватися, а сама процедура повинна бути легко доступна для кожного процесу.

Найпростіший метод - використання загальних областей пам'яті, до яких різні процеси мають доступ для читання / запису. Очевидно, що така область є поділяємим ресурсом, доступ до якого має бути захищений, наприклад, семафором. Головна перевага загальних областей пам'яті полягає в тому, що до них можна організувати прямий і миттєвий доступ, наприклад один процес може послідовно записувати поля, а інший потім зчитувати цілі блоки даних.

При програмуванні на машинному рівні загальні області розміщуються в оперативній пам'яті за відомими адресами. У мовах високого рівня замість цього використовуються глобальні змінні, доступні кільком дочірнім процесам. Так, наприклад, відбувається при породженні потоків, для яких змінні батьківського процесу є глобальними і працюють як загальні області пам'яті. В разі можливих конфліктів доступу до загальних областей вони повинні бути захищені семафорами.

3.2. Поштові скриньки

Інший метод, що дозволяє одночасно здійснювати обмін даними і синхронізацію процесів, - це поштові скриньки. Поштова скринька являє собою структуру даних, призначену для прийому і зберігання повідомлень Для обміну повідомленнями різного типу можна визначити кілька поштових скриньок.

У багатьох операційних системах поштові скриньки реалізовані у вигляді логічних файлів, доступ до яких аналогічний доступу до фізичних файлів. З поштовими скриньками дозволені наступні операції: створення, відкриття, запис / читання повідомлення, закриття, видалення. У деяких системах підтримуються додаткові службові функції, наприклад, лічильник повідомлень в поштовій скриньці або читання повідомлення без видалення його із скриньки.

Поштові скриньки розміщуються в оперативній пам'яті або на диску і існують лише до виключення живлення або перезавантаження. Якщо вони фізично не розташовані на диску, то вважаються тимчасовими файлами, їх знищують після виключення системи. Поштові скриньки не мають імен подібно реальним файлам - при створенні їм присвоюються логічні ідентифікатори, які використовуються процесами при зверненні.

Для створення поштової скриньки операційна система визначає покажчики на область пам'яті для операцій читання / запису і відповідні змінні для захисту доступу. Основними методами реалізації є або буфер, розмір якого задається при створенні скриньки, або пов'язаний список, який, в принципі, не накладає ніяких обмежень на кількість повідомлень в поштовій скриньці.

У найбільш поширених реалізаціях процес, який посилає повідомлення, записує його в поштову скриньку за допомогою оператора, схожого на оператор запису у файл:

put_mailbox (# 1, message)

Аналогічно, для отримання повідомлення процес зчитує його з поштової скриньки за допомогою оператора виду:

get _mailbox (# 1, message)

Запис повідомлення у поштову скриньку означає, що воно просто копіюється у вказану поштову скриньку. Може трапитися, що в поштовій скриньці не вистачає місця для зберігання нового повідомлення, тобто поштова скринька або занадто мала або зберігаються в ній повідомлення, що ще не прочитані.

При читанні з поштової скриньки найстаріше повідомлення пересилається в приймаючу структуру даних і видаляється зі скриньки. Поштова скринька це приклад класичної черги, організованої за принципом FIFO. Операція читання з порожньої скриньки призводить до різних результатів в залежності від способу реалізації - або повертається порожній рядок (нульової довжини), або операція читання блокується до отримання повідомлення. В останньому випадку, щоб уникнути небажаної зупинки процесу, необхідно попередньо перевірити число повідомлень, що є в даний момент у скриньці.

3.3. Канали

Канал (pipe) являє собою засіб обміну даними між двома процесами, з яких один записує, а інший зчитує символи. Цей механізм був спочатку розроблений для середовища UNIX як засіб перенаправлення входу і виходу процесу. В ОС UNIX фізичні пристрої введення / виведення розглядають як файли, а кожна програма має стандартний пристрій введення (вхід) і стандартний пристрій виводу (вихід), клавіатуру та екран монітора - можна перевизначити, наприклад, за допомогою файлів. Коли вихід однієї програми перенаправляється на вхід іншої, створюється механізм, званий каналом (в операційних системах для позначення каналу використовується символ "|"). Канали застосовуються в операційних системах UNIX, OS / 9 і Windows NT як засіб зв'язку між процесами (програмами).

Канали можна розглядати як окремий випадок поштової скриньки. Різниця між ними полягає в організації потоку даних - поштові скриньки працюють з повідомленнями, тобто даними, для яких відомі формат і довжина, а канали принципово орієнтовані на неструктуровані потоки символів. У деяких операційних системах, однак, можливо визначити структуру переданих по каналу даних. Зазвичай процес, що виконує операцію читання з каналу, чекає, поки в ньому не з'являться дані. В даний час операційні системи включають

методи, що дозволяють уникнути блокування програми, якщо це небажано з точки зору її логіки.

Операції над каналами еквівалентні читання / запису фізичних файлів. Вони включають функції, як визначити, відкрити, читати, записати, закрити, видалити. Додаткові операції можуть встановлювати прапори режиму доступу, визначати розмір буфера і т.д.

Розглянемо приклад, де для передачі повідомлень обрана синхронна модель. Вона будується на основі посередника з назвою “канал”. Кожний канал має унікальне ім'я й передає інформацію тільки в одному напрямку, і так само використовується тільки одним процесом-передавачем і одним процесом-приймачем.

Для передачі значення виразу через канал можна використати простий процес-примітив виводу:

channel ! expression або

ім'я_каналу ! вираз.

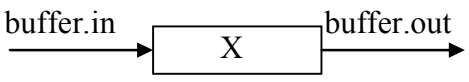
Символ “!” позначає вивід у канал. Для прийому значення з каналу в деяку змінну використовується процес-примітив введення:

channel ? variable або

ім'я_каналу ? змінна.

Символ “?” позначає введення значення змінної.

Зв'язок між процесами синхронний. Тому перший процес, що виконав одну із цих команд буде припинений доти, доки його не “наздожене” процес-контрагент, що виконує процес-примітив вводу-виводу. Як приклад розглянемо реалізацію буфера даних:



SEQ

buffer.in ? x

buffer.out ! x

Цей процес має на увазі, що паралельно із ним функціонує хоча б один процес, що містить процес-примітив виводу в каталог з ім'ям **buffer.in** і процес-примітив введення з каналом з ім'ям **buffer.out**

Завдяки тому, що введення / виведення в файл і на фізичні пристрої та вхід / вихід процесів трактуються однаково, канали є природним засобом взаємодії між процесами в системах "клієнт-сервер". Механізм каналів в UNIX може в деяких випадках залежати від протоколу TCP / IP, а в Windows NT канали працюють з будь-яким транспортним протоколом. Слід мати на увазі, що зовні простий механізм каналів може вимагати великих накладних витрат при реалізації, особливо в мережевих системах.

3.4. Віддалений виклик процедур

Модель "клієнт-сервер" побудована на обміні повідомленнями "регулярної" структури, які можна передавати, наприклад, через механізм каналів. Однак основною процедурою обміну даними і синхронізації в середовищі "клієнт-сервер" є віддалений виклик процедур (Remote Procedure Call - RPC). Останній може розглядатися як виклик підпрограми, при якому операційна система відповідає за маршрутизацію і доставку виклику до вузла, де знаходиться ця підпрограма. Нотація звернення до процедури не залежить від того, чи є вона локальною або віддаленою по відношенню до викликаючої програми. Це істотно полегшує програмування.

В системі реального часу істотно, є RPC блокуючим чи ні. Блокуючий RPC не повертає управління викликаючому процесу, доки він не закінчить свою роботу, наприклад, доки не підготує дані для відповіді. Неблокуючий RPC повертає управління викликаючій процедурі після закінчення деякого часу (time out) незалежно від того, чи завершила роботу процедура; в будь-якому випадку програма, що викликає, отримує код, що ідентифікує результат виконання виклику, - код повернення. Таким чином, неблокіруючі RPC мають важливе значення, з точки зору гарантії живучості системи.

3.5. Порівняння методів синхронізації і обміну даними

Може здатися, що основні завдання, пов'язані з паралельним програмуванням, взаємним виключенням, синхронізацією і комунікаціями між процесами, мають мало спільного, але, по суті - це просто різні способи досягнення однієї мети. Методи синхронізації можна використовувати для організації взаємного виключення і комунікацій. Аналогічно, за допомогою техніки комунікацій між процесами можна реалізувати функції синхронізації і взаємного виключення процесів.

Наприклад, семафор еквівалентний поштовій скриньці, у якій накопичуються повідомлення нульової довжини, - операції signal і wait еквівалентні операціям put і get поштової скриньки, а поточне значення семафора еквівалентно числу поміщених в поштову скриньку повідомлень. Аналогічно можна організувати взаємне виключення і захист ресурсів за допомогою поштових скриньок. У цьому випадку повідомлення виконує функцію "маркера". Процес, який отримав цей "маркер", набуває право входити в критичну секцію або розпоряджатися ресурсами системи. При виході із секції або звільнення ресурсу процес поміщає "маркер" в поштову скриньку. Наступний процес читає з поштової скриньки, отримує "маркер" і може увійти в критичну секцію.

Зв'язок між різними підходами має практичне значення в тому випадку, якщо в системі застосовується тільки один з них, а всі інші потрібно будувати на його основі. Сучасні операційні системи, що підтримують багатозадачність і операції в реальному часі, застосовують усі згадані методи. Передача повідомлень і доступ до загальних областей пам'яті повільніше, ніж перевірка і оновлення семафора і змінної події, і вимагає додаткових накладних витрат. Якщо є вибір між різними методами синхронізації і взаємодії, слід використовувати той з них, який краще вирішує конкретну проблему - результуюча програма буде зрозуміліше і, можливо, швидше працювати. Крім того, досить важливо оцінити, наскільки ефективно в наявному програмному

середовищі реалізуються конкретні рішення. Слід уникати незнайомих і неприродних конструкцій.

У розподілених системах завжди існує ризик втратити повідомлення в мережі. Якщо мережева система налаштована так, що вона контролює правильність передачі повідомлення, і є засоби для повторної передачі втрачених повідомлень, то прикладна програма не повинна здійснювати додаткові перевірки. Зазвичай нижній рівень операційної системи і процедури мережевого інтерфейсу передають на більш високий рівень код повернення, який прикладна програма повинна перевірити, щоб переконатися, чи була спроба успішною чи ні, і при необхідності повторити її.

Якщо контроль не передбачений, наприклад, використовується служба IP без транспортного протоколу TCP, то прикладна програма несе відповідальність за перевірку результату передачі. Ця операція складніша ніж це здається. Можна використовувати повідомлення, яке підтверджує прийом, але немає гарантії, що воно саме, в свою чергу, не буде втрачено і відправник не почне нову передачу. Ця проблема не має спільного рішення - стратегії передачі повідомлень повинні в кожному випадку розглядатися індивідуально. Можливим вирішенням цієї проблеми є позначати і нумерувати кожне повідомлення таким чином, щоб відправник і одержувач могли стежити за порядком передачі. Цей метод використовується в деяких типах комунікаційних протоколів.

3.6. Контрольні запитання до розділу 3

1. Для чого процеси потребують обміну інформацією.
2. Дайте характеристику методу обміну через загальні області пам’яті.
3. Що означає термін “поштова скринька.”
4. Які існують програмні методи створення поштових скриньок.
5. Як процеси організують обмін інформацією через поштові скриньки.
6. Як організований метод обміну інформацією між процесами через “канал.”
7. Дайте характеристику методу обміну інформацією, що дістав назву віддаленого виклику процедур.
8. Порівняйте між собою відомі методи синхронізації та обміну даними між процесами .
9. Чи можна розглядати семафор як окремий випадок поштової скриньки і коли.

Розділ 4. Планування завдань

4.1. Гарантії планування

У системах **жорсткого реального часу** час завершення виконання кожного з критичних завдань має бути гарантований для всіх можливих сценаріїв роботи системи. Такі гарантії можуть бути дані:

- в результаті вичерпного тестування всіх можливих сценаріїв поведінки керованого об'єкта і керуючих програм;

- побудови статичного розкладу;

- вибору математично обґрунтованого динамічного алгоритму планування.

При виборі алгоритму планування слід враховувати дані можливої залежності завдань. Завдання є залежними, якщо має місце один з двох видів залежностей:

1) критичні секції,

2) обмеження на порядок виконання.

З практичної точки зору алгоритми планування взаємозалежних завдань важливіші, ніж алгоритми планування незалежних задач. При наявності дешевих мікроконтролерів не має сенсу організовувати мультипрограмне виконання великої кількості незалежних задач на одному комп'ютері, так як при цьому значно зростає складність програмного забезпечення. Зазвичай одночасно виконувані завдання повинні обмінюватися інформацією і отримувати доступ до загальних даних для досягнення спільної мети системи - це і є залежні завдання (задачі). Тому існування деякої переваги послідовності виконання завдань або взаємного виключення - це, скоріше, норма для систем реального часу, ніж виняток.

Проблема планування залежних задач дуже складна, пошук її оптимального вирішення вимагає великих обчислювальних ресурсів, які можна порівняти з тими, які потрібні для власне виконання завдань управління. Вирішення цієї проблеми можливе за рахунок наступних заходів:

- розділення проблеми планування на дві частини, щоб одна частина виконувалася заздалегідь, перед запуском системи, а друга, більш проста, частина - під час роботи системи. Попередній аналіз набору завдань з взаємними винятками може складатися, наприклад, у виявленні так званих заборонених областей часу, під час яких не можна призначати виконання завдань, що містять критичні секції;

- введення обмежуючих припущень про поведінку набору задач.

При такому підході планування наближається до статичного.

4.2. Основні параметри завдань (задач)

Кожна робота завдання характеризується наступними часовими параметрами:

- r (Release Time) - момент часу, коли виникає необхідність в передачі управління завданню (задачі); можна вважати, що це момент події;
- d (Absolute Deadline) - абсолютний крайній термін, момент часу, до якого завдання повинне завершити чергову роботу;
- s (Start Time) - момент часу, коли завдання починає виконуватись на процесорі;
- c (Completion Time) - момент часу, коли завдання закінчило роботу, обробивши подію;
- D (Relative Deadline) - відносний крайній термін, D = d - r;
- e (Execution Time) - час виконання завдання при виконанні нею черговий роботи, e = c - s;
- R (Response Time) - час відгуку, R = c - r.

Діаграма на рис. 4.1. ілюструє ці параметри:

Наведена на цій діаграмі робота задачі має наступні параметри: $r = 2$, $d = 11$, $s = 5$, $c = 9$, $D = 11 - 2 = 9$, $e = 9 - 5 = 4$, $R = 9 - 2 = 7$.

Згадані параметри визначаються наступним чином. Терміни переходу задач у стан готовності, по суті, визначається природою об’єкту керування. Граничні терміни (d, D) визначає розробник СРЧ, виходячи із властивостей об’єкту керування. Терміни виконання задач *e* визначається архітектурою

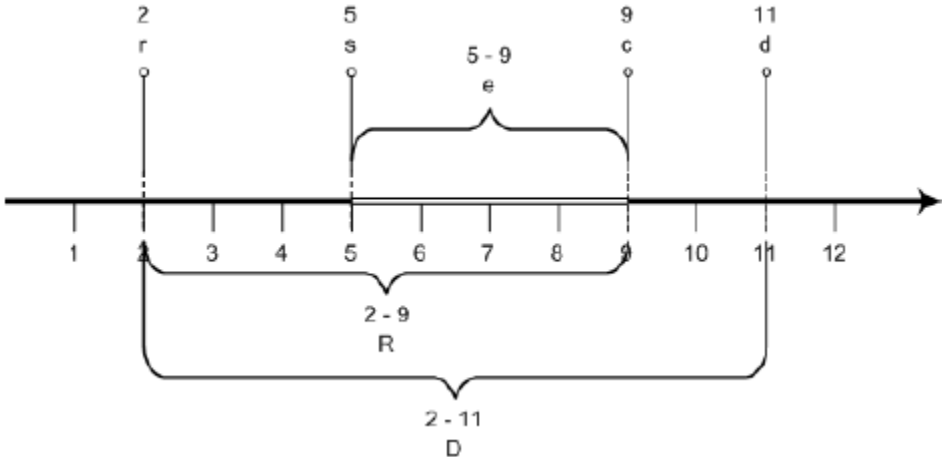


Рис. 4.1. Параметри завдань (задачі)

процесора, його тактовою частотою і конкретною реалізацією того чи іншого алгоритму. Для визначення останньої величини можна використовувати два підходи:

- підрахунок тактів процесора, необхідних на виконання тієї або іншої задачі. Такий підрахунок надзвичайно ускладнюється у разі, якщо процесор містить механізми типу конвеєрів і кешування;

- часи виконання безпосередньо вимірюються. Однак у випадку процесорів з конвеєрами і кешами такі вимірювання не дають гарантії, що буде виміряно саме максимальний час виконання того чи іншого коду. Крім того, системи, що використовують механізм підкачки сторінок, також є погано передбачуваними. Вважається, що такого роду механізми непридатні для систем реального часу.

Характер виникнення подій на керованому об'єкті відображається у наступній класифікації завдань:

- періодичні завдання;
- спорадичні завдання (неперіодичні завдання з жорстким крайнім терміном);
- аперіодичні завдання (неперіодичні завдання з м'яким крайнім терміном).

Надалі будуть використовуватися наступні позначення:

{Tj} - набір завдань; моменти часу, коли виникає необхідність у виконанні j-го завдання, - rjk. При цьому для періодичного завдання rj1 буде називатися фазою завдання φj. Періодична задача буде позначатися як Tj [φj, pj, ej, Dj].

Відзначимо, що для періодичного завдання можливі будь-які співвідношення між періодом і відносним крайнім терміном:

• pj = Dj;

• pj> Dj;

• pj <Dj (в цьому випадку в один і той же проміжок часу може існувати кілька примірників однієї й тієї ж задачі).

Коефіцієнтом використання процесорного часу періодичної задачі називається відношення часу її виконання до її періоду:

$$u_j = e_j/p_j. \qquad (4.1)$$

Коефіцієнтом використання процесорного часу системи періодичних задач називається сума відношень часу виконання кожної задачі до її періоду

$$U = \Sigma u_j = \Sigma\, e_j/p_j. \qquad (4.2)$$

4.3. Статичне планування

Розклад називають коректним, якщо витримані усі граничні часові терміни. Сукупність задач $\{Tj\}$ називають допускаючим планування деяким алгоритмом планування (АП), якщо цей АП завжди дає коректний розклад для цього набору задач.

Оптимальним АП називають такий АП, який для любого набору задач дає коректний розклад, якщо такий взагалі існує для даного набору задач.

Розглянемо систему незалежних періодичних задач $\{Tj\}$. Окрім того, будемо вважати, що серед сукупності задач можуть бути ще аперіодичні та спорадичні задачі як деякі особливі випадки.

Розкладом для періодичних завдань буде таблиця, у якій вказано, у який момент часу яку задачу треба поставити на виконання. Цей розклад складається за проміжок часу, що дорівнює *гіперперіоду* (найменшому загальному кратному періодів) усіх задач.

Таким чином, планувальник періодично повторює послідовності запуску задач, які задані у розкладі.

Приклад статичного розкладу. Розглянемо набір з чотирьох задач:
*T*1[*,4,*,1] *T*2[*,5,*,1.8] *T*3[*,20,*,1] *T*4[*,20,*,2].
Гіперперіод даної системи задач дорівнює 20. На протязі цього періоду перша задача повинна отримати управління 5 разів, друга - 4, а третя і четверта - по одному разу. При цьому мають обов'язково бути збережені усі крайні часові терміни. Розклад для такої системи задач може мати вигляд, наприклад, наступним чином (табл. 4.1.).

Планувальники розглянутого типу, як правило, працюють по перериваннях від таймеру, **задачі** при цьому являють собою просто підпрограми, що викликаються в потрібні моменти з обробника переривань.

Табл. 4.1

Приклад розкладу

№	Время	Задача	№	Время	Задача
1	0	T_1	10	10.8	I
2	1	T_3	11	12	T_2
3	2	T_2	12	13.8	T_1
4	3.8	I	13	14.8	I
5	4	T_1	14	16	T_1
6	5	I*	15	17	I
7	6	T_4	16	18	T_2
8	8	T_2	17	19.8	I
9	9.8	T_1			

Примітка. Тут I позначає «завдання» простою (Idle task).

Оскільки таймери, як правило, програмуються на генерацію переривання не через якийсь проміжок часу від початку відліку часу, а на переривання через

якийсь проміжок часу від даного конкретного моменту часу, то з практичної точки зору наведену таблицю зручніше представити в іншому вигляді: замість проміжків часу від початку циклу поміщати в неї проміжки часу <<відносні>>, від одного переривання до іншого (в рамках циклу). Відповідним чином модифікована таблиця для наведеного прикладу набору завдань буде виглядати по іншому (табл. 4.2).

Табл. 4.2

Приклад розкладу

№	Время	Задача	№	Время	Задача
1	0.2	T_1	10	1	I
2	1	T_3	11	1.2	T_2
3	1	T_2	12	1.8	T_1
4	1.8	I	13	1	I
5	0.2	T_1	14	1.2	T_1
6	1	I	15	1	I
7	1	T_4	16	1	T_2
8	2	T_2	17	1.8	I
9	1.8	T_1			

Переваги даного класу алгоритмів:

- виняткова простота, обумовлена відсутністю понять “ процес”/ “ потік “, передача управління завданню - це виклик підпрограми;

- як наслідок, результати тестувань та перевірок вельми надійні.

Завдяки цим якостям, алгоритми саме цього типу частіше застосовуються там, де потрібна висока надійність.

Недоліки:

- негнучкість. Будь-яка зміна (числа завдань, часів виконання т.д.) вимагає зупинки системи, перерахунку розкладу і перекомпіляції додатків;

- планувальник фактично «відв'язано» від зовнішнього світу, так як він працює по перериваннях від таймера. Облік спорадичних завдань дуже складний;

- розмір таблиці з розкладом може виявитися дуже великим при відповідних співвідношеннях між періодами завдань.

4.4. Динамічне планування з динамічними пріоритетами

Існують два різновиди планувальників з динамічними пріоритетами:
• **EDF (earliest deadline first)** - пріоритет завдань призначається за принципом «в кожен момент часу найвищий пріоритет має та задача, у якій залишилося найменше часу до крайнього терміну».

Модифікації бувають:

- з витісненням завдань;
- без витіснення завдань.

• **LLF (least laxity first)** - пріоритет завдань призначається за принципом «в кожен момент часу найвищий пріоритет має задача з найменшим запасом часу».

Теорема. Алгоритм EDF оптимальний для будь-якого набору періодичних завдань з будь-яким співвідношенням між періодами і крайніми термінами, якщо:

- всі завдання незалежні;
- можливе витіснення;
- витіснення не вимагає часових витрат.

Доказ ґрунтується на наступному: якщо для якогось набору завдань можна скласти коректний розклад, то цей розклад завжди можна привести до такого виду (і при цьому він зостанеться коректним), шо порядок виконання задач буде таким, який вийшов би при використанні EDF.

Для алгоритму LLF має місце така ж теорема. Пояснимо тут поняття «резерв часу» (laxity).

Резервом (запасом) часу називається різниця між часом, що залишився до крайнього терміну, і часом, який завданню ще потрібно попрацювати, тобто

$$L(t) = (d - t) - e(rem).$$

Наведена нижче діаграма ілюструє це поняття (рис. 4.2)

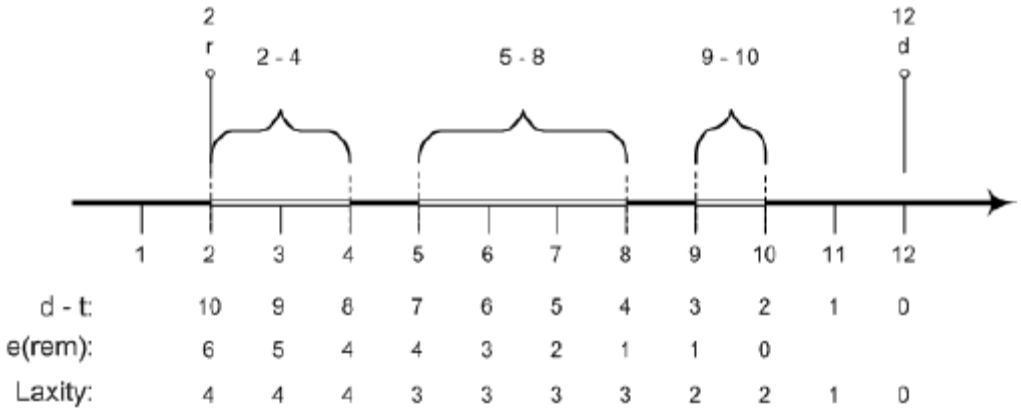


Рис. 4.2. Резерв часу

На цій діаграмі показана робота деякого завдання, яке стало готовим в момент часу 2 і має крайній термін в момент часу 12. Задача відпрацювала з витисненням. Поки вона виконує свою роботу, запас часу залишається постійним, оскільки зменшується і час до крайнього терміну, і кількість часу, яке ще потрібно пропрацювати - зрозуміло, на одну і ту ж величину. Якщо ж задача простоює через витіснення її іншими, більш пріоритетними завданнями, то крайній термін наближається, а час, який ще потрібно попрацювати, залишається постійним, тому в такі проміжки запас часу, природно, зменшується.

Теорема. Алгоритм EDF без витіснення не є неоптимальним. Довести цю теорему можна, якщо навести один єдиний контр-приклад, тобто навести такій набір задач, для якого, в принципі можливо скласти коректний розклад, а алгоритм планування EDF без витіснення дає некоректний розклад.

Нехай маємо три задачі з параметрами:

T1: r, e, d = 0, 3, 10

T2 : r,e, d= 2, 6, 14

T3 : r, e, d = 4, 4, 12

Для даного набору можна скласти (неважливо яким засобом) наступний розклад (рис. 4.3).

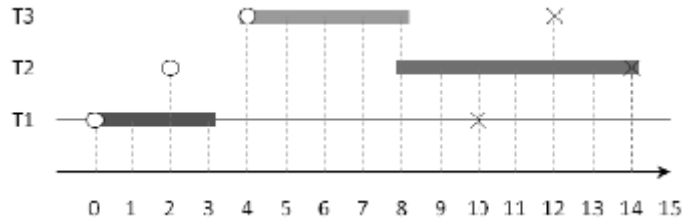


Рис. 4.3. Можливий розклад для системи з трьох задач

Алгоритм EDF без витіснення дає наступний розклад (занотуємо для ясності, що моменти прийняття рішень планувальником - це моменти, коли або з'являється задача у стані готовності, або якась задача завершує свою чергову роботу (рис.4.4).

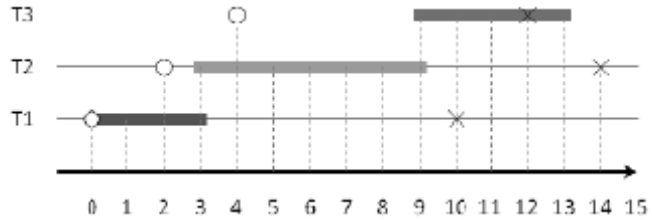


Рис. 4.4. Розклад складений EDF - планувальником без витіснення

В момент часу 3 готова тільки задача 2 , тому вона і обирається для виконання та не витісняється до тих пір, доки повністю не виконає свою роботу не глядачі на те, що у момент часу 4 у стан готовності переходить задача 3, у якої граничний термін настає раніше, ніж у задачі 2. В результаті задача 3 пропускає свій граничний термін. Таким чином, алгоритм EDF без витіснення не є оптимальним.

Для порівняння наведемо розклад, який склав би планувальник з можливістю витіснення задач (рис. 4.5).

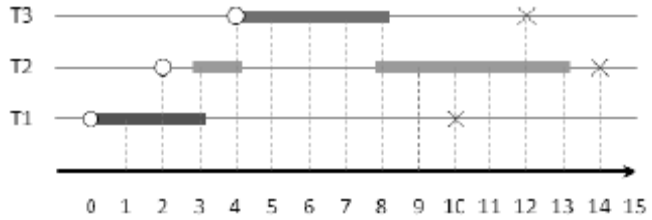


Рис. 4.5. Розклад, складений EDF - планувальником з витісненням

Як видно, у даному випадку всі задачі встигли завершити свої роботи до критичних термінів. В момент часу 2 стала готовою задача 2, але вона витісняє задачу 1, оскільки у задачі 2 крайній термін настає пізніше, ніж у задачі 1 . Задача 2 отримує управління тільки в момент часу 3. Але в момент часу 4 з'являється задача 3 і витісняє задачу 2, так як у задачі 3 крайній термін настає раніше ніж у задачі 2. В момент часу 8, коли задача 3 завершить свою роботу, управління знову отримує задача 2.

Із сказаного вище не слід робити висновок, шо EDF без витіснення гірше EDF з витісненням, оскільки докази оптимальності останнього наведено за припущенням, що власне само витіснення задач (та переключення контекстів) не потребує ніякого часу, що не відповідає дійсності. Тому при аналізі алгоритмів планування слід враховувати накладні витрати на планування, наприклад, включаючи їх у терміни виконання задач *e*.

У подальшому замість терміну виконання задачі *e,* термін виконання задачі з урахуванням накладних витрат будемо позначати ν.

Для EDF - планувальника з витісненням має місце твердження про перевірку системи періодичних задач на предмет можливості розробки розкладу для неї з використанням такого планувальника.

Теорема. Для того, щоб систем *{Tj} незалежних періодичних задач, що витісняються* (таких, що для любого j має місце *Dj>=pj*), була планована EDF-планувальником, необхідно і достатньо, щоб коефіцієнт використання процесорного часу системи ***u*** не перевищував одиниці;

$$u <= 1. \qquad (4.3)$$

Таку умову ще називають **критерієм здійсненності планування.**

Якщо ж відносні крайні терміни менше періодів, то така умова є необхідною, але не достатньою. Наприклад, виконання двох завдань

T1 / p, e, D = 2; 1; 1,9,

T2 / p, e, D = 2; 1; 1,9

не може бути коректно розплановано ніяким алгоритмом, хоча U для неї

дорівнює одиниці. Для таких випадків замість поняття коефіцієнт використання процесорного часу» використовують поняття «щільність завдання» δ, яке визначається, як

$$\delta = e / \min (D, p).$$

Сумарна щільність системи завдань визначається як сума щільності всіх завдань системи:

$$\Delta = \Sigma\delta_j = \Sigma e_j / min\ (p_j, D_j). \qquad (4.4)$$

Для систем такого роду має місце дещо інше твердження щодо можливості складання коректного розкладу:

Теорема. Для того щоб система {Tj} незалежних періодичних завдань, що витісняються (таких, що для деяких j має місце Dj <pj), була планованою EDF-планувальником, достатньо, щоб сумарна щільність системи Δ не перевищувала одиниці.

Відзначимо, що ця умова не є необхідною. Це означає, що якщо вона виконується, то для системи можна скласти коректний розклад; якщо ж ця умова не виконується, то це не значить, що коректний розклад скласти не можна.

Наприклад, для пари задач, що наведена нижче, для якої умова $\Delta \leq 1$ не виконується, коректний розклад існує:

T1 (2; 0,6;1);

T2 (5; 2,3).

Для цієї системи Δ = 0,6 / 1 + 2,3 / 5 = 1,06> 1, але, тим не менш, коректний розклад скласти можна.

4.5. Динамічне планування із статичними пріоритетами

Існує два розповсюджених способи визначення пріоритетів:

- RMS (rate monotonic scheduling). Правило визначення пріоритетів таке: чим менше період задачі, тим вищий у неї пріоритет. Тобто, чим частіше (звідси rate) задача переходить у стан готовності, тим вищий у неї пріоритет;

- DMS (deadline monotonic scheduling). У цьому алгоритмі визначаються дещо інакше: чим менший відносний крайній термін виконання задачі, тим вищий у неї пріоритет.

Нижче наведено приклад розкладу (рис. 4.6), складеного RMS - планувальником для синхронної системи з трьох періодичних задач з такими параметрами:

$T_1(3; 0{,}5)$;
$T_2(4; 1)$;
$T_3(6; 2)$.

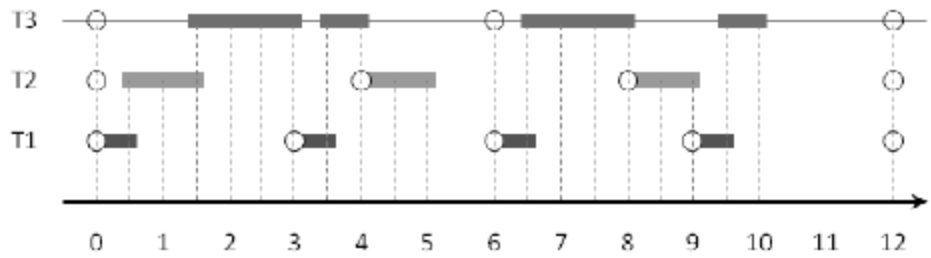


Рис. 4.6. Розклад, складений RMS - планувальником

Відносний крайній термін дорівнює періоду.

У відповідності з вищенаведеним правилом найвищий пріоритет має перша задача, а найнижчий - третя. В момент часу 0 маємо 3 готові задачі, процесорний час отримує перша. У момент часу 0,5 готові дві задачі (2 і 3) , управління отримує друга. В момент часу 3 знову стає готовою перша задача, тому вона витісняє третю. В момент часу 3,5 управління знову отримує третя задача як одна, що готова. В момент часу 4,0 готова тільки задача 2 , вона і

отримує керування. У момент часу 6 готові дві задачі (1 і 3), управління віддається першій. Далі працює третя задача, але в момент часу 8 вона витісняється другою. В момент 9 знову готова перша задача, вона відпрацьовує і, нарешті, в момент 9,5 отримує управління третя задача. Далі все повторюється спочатку (за умови, що число задач у системі залишається незмінним).

Далі наведено приклад розкладу, складеного DMS - планувальником для наступної системи задач:

T1(3; 0,5);

T2(4; 1);

T3(6; 2).

Відмінність цієї системи від попередньої тільки у тому, що у цієї задачі відносний крайній термін не дорівнює періоду, а менше його у два рази. Згідно правилу призначення пріоритетів, найбільший пріоритет буде мати друга задача, а самий низький - третя.

DMS - планувальник для такої системи задач сформує наступний розклад (рис. 4.7):

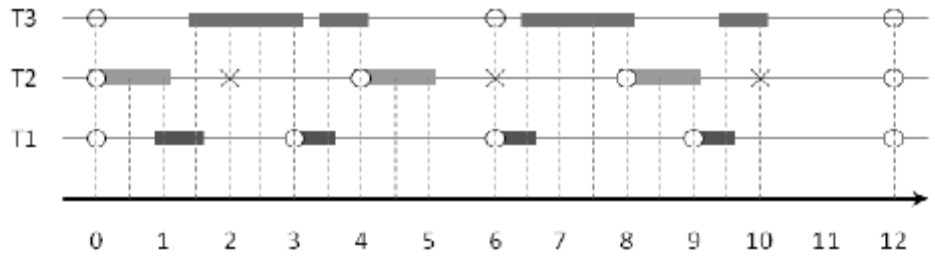


Рис. 4.7. Розклад, створений DMS - планувальником

У випадку правил, що розглядаються, для призначення пріоритетів доведена наступна теорема:

Теорема. Алгоритми DMS і RMS не є оптимальними Система {Tj} періодичних задач називається **системою з простою періодичністю,** якщо для довільної пари *Ti, Tk* (при цьому *pi* < *pk*) має місце рівняння:

pk mod *pi* = 0. Інакше кажучи, періоди усіх задач системи попарно діляться на ціле одна на одну.

Наприклад, система {*T1*(2,1), *T2*(4,1), *T3*(8,1)} - система з простою періодичністю, а система {*T!*(2,1),*T2*(5,1), *T3*(10,1)} - такою не являється (5 mod 2 = 1, не = 0).

Для систем з простою періодичністю існує наступна теорема:

Теорема. Система {*Tj*} незалежних задач з простою періодичністю, що витісняються, з періодами, що не перевишують відносні граничні крайні терміни, планується на одному процесорі тоді і тільки тоді, коли коефіцієнт використання процесорного часу цієї системи не перевищує одиниці.

Для систем з довільними співвідношеннями між періодами задач відома наступна теорема (критерій перевірки спланованості системи алгоритмом RMS):

Теорема. Система {Tj} n незалежних витісняємих періодичних завдань з періодами, рівними відносним крайнім термінам, планується на одному процесорі, якщо коефіцієнт використання процесорного часу цієї системи не перевищує величини

$$U_{RM} = n(2^{1/n} - 1) \qquad (4.5)$$

Ця умова є достатньою, але не необхідною.

4.6. Контрольні запитання до розділу 4

1. Які гарантії своєчасного завершення задач мають надаватися у СРЧ.
2. Що означає відповідність критерію здійсненності системи задач реального часу.
3. Якими параметрами характеризується кожна задача СРЧ.
4. Які особливості побудови розкладу для періодичних задач СРЧ.
5. Які переваги і недоліки алгоритмів статичного планування.
6. Які переваги алгоритмів статичного планування з використанням пріоритетів задач.
7. Які існують типи динамічних планувальників, що використовують пріоритети задач.
8. Які існують критерії здійсненності для різних типів задач і алгоритмів планування.
9. Які переваги і недоліки алгоритмів планування з витісненням і без витіснення задач.

Розділ 5. Короткий огляд поширених ОС РЧ

5.1 . Загальні характерні властивості ОС РЧ

Перераховані нижче властивості характерні для більшості розповсюджених ОС РЧ, хоча і не є обов'язковими:

- відповідають (у крайньому випадку, частково) стандарту POSIX на інтерфейс прикладних програм, що забезпечує функціонування в режимі реального часу, тобто:

- мають витиснятися ядром;

- надають витисняючий динамічний алгоритм планування, заснований на використанні статичних пріоритетів задач (завдання, як правило, оформлені у вигляді потоків);

- надають механізми синхронізації потоків;

- крім POSIX, мають і свій власний API;

- є модульними і масштабованими; ядро досить мало для того, щоб помістится в постійний запам'ятовуючий пристрій (ПЗУ) вбудованих систем, при цьому є можливість додавання засобів введення-виведення, управління файлами, мережевої взаємодії (наприклад, TCP / IP) і т.п.;

- швидкі і ефективні, що досягається за рахунок:

- побудови із застосуванням концепції мікроядра (як правило);

- низьких накладних витрат, наприклад, на роботу планувальника, перемикання контекстів, операції блокування і деблокування, м'ютексів і т.п.;

- незначного часу реакції на переривання (кілька мікросекунд);

- ретельної оптимізації критичних секцій в ядрі (тобто тих ділянок коду, які працюють без витиснення);

- підтримують двоетапну обробку переривань: спочатку виконується коротка підпрограма обслуговування переривання ISR (interrupt service routine), яка ставить у чергу на виконання більш довгу підпрограму обробки переривання IHR (interrupt handling routing); перша з них виконує мінімально необхідні дії, друга - більш часовитратні дії, наприклад, передачу даних від зовнішнього пристрою. Відзначимо, що така властивість не є прерогативою

ОС РЧ. Аналогічні механізми обробки переривань є і в ядрах ОС загального призначення. Наприклад, в ОС Linux подібний механізм носить назву upper half / bottom half, тобто є «нижня» і «верхня» «половини». «Нижній половині» відповідає поняття IHR, верхній - ISR. В ОС Windows є механізм відкладеного виклику процедур (DPC). Підпрограма, що викликається за допомогою цього механізму, аналогічна IHR;

- володіють більш-менш гнучкими схемами планування:

- число пріоритетів потоків - як мінімум 32 (це вимога POSIX); в деяких ОС РЧ це число дорівнює 128 або 256;

- для потоків з однаковим рівнем пріоритету (якщо є), застосовуються схеми планування FIFO і RR (round robin);

- як правило, не підтримують алгоритм EDF (можливо, в силу більшої складності його реалізації в порівнянні зі статичними планувальниками);

- мають досить високу точність таймерів і лічильників, номінальне - наносекунди, реальне - мікросекунди);

- як правило, не містять механізмів підкачки і захисту пам’яті; при цьому часто завдання і ядро виконуються в єдиному адресном просторі;

- відповідають схемі розробки ПО Host-Target, або Self-Hosted (рідше).

Self-Hosted ОС РЧ - це системи, в яких користувачі можуть розробляти програми, працюючи в самій ОС РЧ. Зазвичай це визначає, що ОС РЧ підтримує файлову систему, засоби вводу - виведення, призначений для користувача інтерфейс, є компілятори, відладчик, засоби аналізу програм, текстові редактори, які працюють під керуванням ОС РЧ.

Перевага таких систем - більш простий і наочний механізм створення і запуску додатків, які працюють на тій же машині, що і користувач.

Недоліком є те, що промисловому комп'ютеру під час його реальної експлуатації часто взагалі не потрібний призначений для користувача інтерфейс і можливість запуску важких програм на зразок компілятора. Отже, більшість з описаних вище можливостей ОС РЧ просто не використовуються і тільки даремно займають пам'ять і інші ресурси

комп'ютера.

Зазвичай self-hosted ОС РЧ застосовуються на «звичайних» комп’ютерах промислового виконання.

Host / Target ОС РЧ - це системи, в яких операційна система і (або) комп’ютер, на якому розробляються додатки (host), і операційна система і (або) комп'ютер, на якому запускаються додатки (target), різні. Зв'язок між комп'ютерами здійснюється за допомогою послідовного з'єднання (COM порту), Ethernet, загальної шини VME або compact PCI. Як host система зазвичай виступає комп'ютер під керуванням UNIX абоWindows NT, як target системи - промисловий або вбудований комп'ютер під керуванням ОС РЧ. Бувають системи, в яких на одному комп'ютері працюють дві операційних системи: «Звичайна» і реального часу.

Перевага таких систем - використання всіх ресурсів «звичайної» системи (таких, як графічний інтерфейс, файлова система, швидкий процесор і великий об'єм оперативної пам'яті) для створення додатків і зменшення розмірів ОС РЧ за рахунок включення тільки потрібних додаткам компонент.

Недолік - відносна складність програмних компонент: крос-компілятора, віддаленого завантажувача і відладчика і т.д.

Відзначимо, що, з одного боку, зростання потужності промислових комп'ютерів дозволяє використовувати self-hosted системи на більшій кількості обчислювальних систем. З іншого боку, збільшується поширення вбудованих систем (в різноманітному промисловому і побутовому обладнанні), що розширює сферу застосування host / target систем (оскільки при великих обсягах випуску ціна системи є визначальним фактором).

5.2. Система CHORUS

Система CHORUS випускається фірмою Chorus Systems (Saint Quentin Yvelines, France). У листопаді 1997 р фірма придбана компанією Sun Microsystems (Menlo Park, CA, USA).

Основні характеристики:

- тип: host-target (CHORUS / Micro і CHORUS / ClassiX) і selfhosted (CHORUS / MiX);

- архітектура: на основі мікроядра;

- стандарт: власний і POSIX 1003;

- властивості як ОС РЧ:

- багатозадачність: POSIX 1003 (багатопроцесність і багатопоточність).

- багатопроцесність: так.

- багатопроцесорність: так;

- рівнів пріоритетів: 256;

- планування: пріоритетне, FIFO; preemptive ядро;

- ОС розробки (host): CHORUS / UNIX / Windows;

- процесори (target): Intel 80x86, Motorola 68xxx, Motorola88xxx, SPARC, PowerPC, T805, MIPS, PA-RISC, YMP (Cray), DEC Alpha;

- лінії зв'язку host-target: послідовний канал і ethernet;

- мінімальний розмір: 10 Kb для CHORUS / Micro, 50 Kb для CHORUS / ClassiX;

- засоби синхронізації і взаємодії: POSIX 1003 (семафори, mutex, condvar);

- засоби розробки:

- CHORUS / Harmony - інтегроване середовище розробки - C / C ++, що включає компілятори, відладчик, аналізатор;

- CHORUS / JaZZ - віртуальна машина Java.

5.3. Система LynxOS

Система LynxOS випускається фірмою Lynx Real Time Systems (Los Gatos, USA). Основні характеристики:

- тип: self-hosted;

- архітектура: на основі мікроядра;

- стандарт: POSIX 1003;

- властивості як ОС РЧ:

- багатозадачність: POSIX 1003 (багатопроцесність і багатопоточність):

- багатороцесорність: так;

- рівнів пріоритетів: 255;

- планування: FIFO, round robin, Quantum; preemptive ядро;

- ОС розробки (host): немає;

- процесори (target): Intel 80x86, Motorola 68xxx, SPARC, PowerPC;

- мінімальний розмір:

- повної системи: 256 Kb;

- скороченого варіанту системи: 124 Kb;

- тільки ядра: 33Kb;

- засоби синхронізації і взаємодії: POSIX 1003 (семафори, mutex, condvar);

- засоби розробки:

- комплекти розробника, що включають компілятор C / C ++, відладчик, аналізатор;

- X Windows / Motif для Lyux;

- Total View - багатопрцесорний відладчик;

- ядро розміром 28 Kb, що забезпечує диспетчеризацію переривань, планування завдань і механізми їх синхронізації;

- є можливість додавання KPI (kernel plug-ins), за рахунок чого цю ОС РЧ можна використовувати як звичайну ОС типу UNIX, зокрема, для розробки додатків;

- підтримує механізми захисту пам'яті (при наявності MMU) і підкачки.

5.4. Система OS-9

Система OS-9 випускається фірмою Microware (USA). Основні характеристики:

- тип: host-target;

- архітектура: на основі мікроядра;

- стандарт: власний, виклики схожі на UNIX;

- властивості як ОС РЧ:

- багатозадачність: багатопроцесність;

- багатопроцесорність: немає;

- рівнів пріоритетів: 65535;

- планування: пріоритетне, FIFO, спеціальний механізм планування - preemptive ядро;

- ОС розробки (host): UNIX / Windows;

- процесори (target): Motorola 68xxx, Intel 80x86, ARM, MIPS, PowerPC;

- лінії зв'язку host-target: послідовний канал і Ethernet;

- мінімальний розмір: 16 Kb;

- засобити синхронізації і взаємодії: колективна пам'ять, сигнали, семафори, події та ін.;

- засоби розробки:

- Hawk - інтегроване середовище розробки на C / C ++;

- Personal Java - віртуальна машина Java.

5.5. Система pSOSystem (pSOS)

Система pSOS випускається Integrated Systems (Santa Clara, USA).
У лютому 2000 р фірма придбана компанією Wind River Systems (Alameda, CA, USA). Основні характеристики:

- тип: host-target;

- архітектура: на основі мікроядра;

- стандарт: власний;

- властивості як ОС РЧ:

- багатозадачність: багатопроцесність і багатопоточність;

- багатопроцесорність: так;

- рівнів пріоритетів: 255;

- планування: пріоритетне; preemptive ядро;

- ОС розробки (host): UNIX, Windows;

- процесори (target): Motorola 68xxx, Intel 80x86, Intel 80960, ARM, MIPS, PowerPC;

- лінії зв'язку host-target: Ethernet;

- мінімальний розмір: 15 Kb;

- засоби синхронізації і взаємодії: семафори, mutex, події та ін.;

- засоби розробки:

- інтегроване середовище розробки на C / C ++ / Ada;

- об'єктно - орієнтована система;

- підтримує роботу на декількох процесорах;

- планування завдань можливо як на основі пріоритетів, так і на основі заздалегідь створеного розкладу;

- драйвери пристроїв працюють в режимі користувача, при цьому ядро не диспетчеризує переривання.

5.6. Система RTC

Система RTC (Real Time Craft) випускається фірмою GSI-TECSI (Paris, France). Основні характеристики:

- тип: host-target (RTC) і self-hosted (RTC / PC);

- архітектура: монолітна;

- стандарт: SCEPTRE (RTC) і власний (RTC / PC);

- властивості як ОС РЧ:

- багатозадачність: багатопроцесність і багатозадачність;

- багатопроцесорність: так (RTC); немає (RTC / PC);

- рівнів пріоритетів: 65532;

- планування: приоритетне, FIFO; preemptive ядро;

- ОС розробки (host): UNIX / Windows;

- процесори (target): Intel 80x86, Motorola 68xxx, Intel 80C166, Am39K

(для RTC / PC - тільки Intel 80x86);

- лінії зв'язку host-target: Ethernet;

- мінімальний розмір: 1,5 Kb (RTC); 4 Kb (RTC / PC);

- засоби синхронізації і взаємодії: SCEPTRE (семафори, сигнали, поштові скриньки);

- засоби розробки: компілятор C, відладчик Watcom TRC.

5.7. Система VRTX

Система VRTX випускається фірмою Ready Systems(Sunnyvale, USA). Це перша система, сертифікована відповідними органами для використання в системах, критичних по відношенню до здоров'я і життя людей; використовувалася в літаках моделі «Боїнг MD-11».

Основні характеристики:

- тип: host-target;

- архітектура: монолітна;

- стандарт: власний;

- властивості як ОС РЧ:

- багатозадачність: багатопроцесність і багатопоточність;

- багатопроцесорність: немає;

- рівнів пріоритетів: 255;

- планування: пріоритетне; preemptive ядро;

- ОС розробки (host): UNIX / Windows;

- процесори (target): Motorola 68xxx, Intel 80x86, Intel 80960,PowerPC;

- лінії зв'язку host-target: послідовний канал, ethernet, шина VME;

- мінімальний розмір: є різні варіанти ядер, орієнтовані на системи різного масштабу; наприклад, одна з модифікацій, що вимагає до 8 Ка ПЗУ і 1 Ка ОЗУ, призначена для стільникових телефонів і інших подібних пристроїв;

- засоби синхронізації і взаємодії: семафори, черги, сигнали, і ін.

- засоби розробки: Master Works - інтегроване середовище розробки;

- Xray - спеціалізований відладчик;

- Simulator Xray - емулятор ядра.

5.8. Система VxWorks

Система VxWorks випускається фірмою Wind River Systems (Alameda, CA, USA). Використовувалася на марсоході «Mars PathFinder».

Основні характеристики:

- тип: host-target;

- архітектура: монолітна (одна з небагатьох ОС РЧ, що має подібну архітектуру);

- стандарт: власний (досить популярний API) і POSIX 1003;

властивості як ОС РЧ:

- багатозадачність: багатопроцесність, багатопоточність;

- багатопроцесорність: так;

- рівнів пріоритетів: 256;

- планування: пріоритетне; preemptive ядро;

- ОС розробки (host): UNIX / Windows;

- процесори (target): Motorola 68xxx, Intel 80x86, Intel 80960,PowerPC, SPARC, Alpha, MIPS, ARM. У стандартній конфігурації працює тільки на одному процесорі, проте є можливість працювти на багатопроцесорних машинах;

- лінії зв'язку host-target: послідовний канал, etnernet, шина VME;

- мінімальний розмір: 5 - 8Kb;

- засоби синхронізації і взаємодії: семафори PQSIX 1003, черги, сигнали;

- засоби розробки:

- TORNADO - інтегроване середовище розробки C / C ++;

- VxSim - емулятор для UNIX;

- WindView - графічний визуализатор стану завдань;

- при наявності MMU забезпечує механізми захисту пам'яті,

при цьому спочатку всі завдання виконуються в одному адресному про-просторі, але завдання може, за своїм бажанням, створити окремий адресний простір.

5.9. Система QNX

Система QNX випускається фірмою QNX SoftWare Systems (USA). Основні характеристики:

- тип: self-hosted;
- архітектура: на основі мікроядра;
- стандарт: POSIX 1003;
- властивості як ОС РЧ:
- багатозадачність: POSIX 1003 (багатопроцесорність і багатогозадачність);
- багатогопроцесорність: так;
- рівнів пріоритетів: 32;
- планування: FIFO, Round Robin, адаптивне; preemptive ядро;
- ОС розробки (host): немає;
- процесори (target): Intel 80x86,
- лінії зв'язку host-target: Ethernet;
- мінімальний розмір: 60 Kb;
- засоби синхронізації і взаємодії: POSIX 1003 (семафори, mutex, condvar);
- засоби розробки:
- комплекти розробника, що включають компілятор C / C ++, відладчик, аналізатор від QNX і незалежних постачальників (наприклад, Watcom / SyBase);
- X Windows / Modif для QNX.

5.10. Спеціалізовані ОС РЧ

Системи, які проектуються під конкретну модель мікроконтролера або

конкретне завдання, мають певні **переваги:**

- найвищу продуктивність,
- найкраще урахування особливостей обладнання,
- найбільшу компактність.

Недоліки:

- великий час розробки,
- висока вартість,
- непереносність.

Прикладами таких систем є ОС РЧ, розроблені багатьма виробниками електронної техніки (Sony, Sagem і ін.), а також системи, розроблені під конкретну велику задачу (наприклад, система управління залізницями TGV у Франції).

5.11. Системи на основі Linux

Пристосування системи Linux до вимог реального часу відбувається за такими трьома напрямками:

1. Підтримка стандартів POSIX, що стосуються систем реального часу. Стандарт POSIX 1003.1c (thread - робота з задачами) вже підтримується, стандарт POSIX 1003.1b (розширення реального часу) підтримується лише частково: реалізовані механізми управління пам'яттю і механізми планування задач; механізми роботи з таймерами, сигналами, POSIX-семафорами і чергами повідомлень поки не реалізовані.

2. Підтримка спеціального обладнання, найважливішим з яких є шина VME. Вже існує підтримка моста VMEPCI, ведуться розробки щодо забезпечення виконання Linux з ПЗУ.

Також для систем реального часу важливим є підвищення розширення здатності таймера системи.

3. Реалізація механізму preemption для ядра системи. Цей механізм, з одного боку, необхідний для того, щоб систему можна було називати системою

реального часу, а з іншого боку, він є дуже складним для реалізації. Linux, як і всі UNIX системи, надовго забороняє переривання при вході в ядро системи, яке є невитсняючим (nonpreemptive). Існують кілька проектів реалізації premption для ядра Linux. За способом вирішення завдання їх можна розділити на:

1. Механізм preemption реалізується шляхом переписування ядра системи (благо воно є у вихідних текстах). На цьому шляху можна досягти найбільш якісних результатів, але на даний момент значних успіхів у цьому плані немає за наступних причин:

а) занадто великий обсяг роботи, пов'язаний з великим об’ємом ядра;

б) занадто висока швидкість зміни ядра, причому, зміни вносяться, не враховуючи інтереси реального часу.

2. Механізм preemption реалізується шляхом написання мікроядра, відповідального за диспетчеризацію переривань і завдань. Ядро Linux працює як завдання з низьким пріоритетом. Саме ядро лише незначно змінено для запобігання блокування їм апаратних переривань. Найбільш відомий представник цього напряму - система RT-Linux (мінімальний розмір без X-Windows - 2,7 Mb).

5.12. Системи на основі Windows NT

Кілька фірм оголосили про створення розширень реального часу для Windows NT. Це означає, що подібні продукти були затребувані, що і підтверджує динаміку їх ринкового розвитку. Насправді, поява свого часу Unix'ів реального часу означало не що інше, як спробу застосувати пануючу програмну технологію для створення додатків реального часу. Поява розширень реального часу для Windows NT має ті ж коріння, ту ж мотивацію. Величезний набір прикладних програм під Windows, потужний програмний інтерфейс WIN32, велика кількість фахівців, які знають цю систему, звичайно спокусливо отримати в системі реального часу всі ці можливості.

Незважаючи на те, що Windows NT створювалася як мережева операційна система, поєднання слів «Windows NT» і «реальний час» багатьма сприймається як нонсенс, хоча в неї при створенні були закладені елементи реального часу, а саме - дворівнева система обробки переривань (ISR і DPC), класи реального часу (процеси з пріоритетами 16 - 32 плануються відповідно до правил реального часу). Може бути, причина появи цих елементів криється в тому, що у розробників Windows NT є досвід створення класичної для свого часу операційної системи реального часу RSX11M (для комп'ютерів фірми DEC).

Звичайно, навіть поверхневий аналіз Windows NT показує, що ця система не годиться для побудови систем жорсткого реального часу (система непередбачувана - час виконання системних викликів і час реакції на переривання сильно залежить від завантаження системи; система велика; немає механізмів захисту від зависання та ін.). Тому навіть в системах м'якого реального часу Windows NT може бути використана тільки при виконанні цілого ряду рекомендацій і обмежень.

Розробники розширень пішли двома шляхами:

1. Використовували ядра класичних операційних систем реального часу в якості доповнення до ядра Windows NT. Такі рішення фірм «LP Eleknroniks» і «Radisys». У першому випадку паралельно з Windows NT працює операційна система VxWorks, у другому випадку - InTime. Крім того, надається набір функцій для зв'язку додатків реального часу і додатків Windows NT.

Ось як, наприклад, це виглядає у «LP Elektroniks»: спочатку стандартним чином завантажується Windows NT, потім за допомогою спеціального завантажувача завантажується операційна система VxWorks, вона розподіляє під себе необхідну пам'ять Windows (що в подальшому дозволяє уникнути конфліктів пам'яті між двома ОС). Після цього повною «господинею» на комп'ютері вже стає VxWorks, віддаючи процесор ядру Windows NT тільки у випадках, коли в ньому немає потреби для додатків VxWorks. Як канал для синхронізації і обміну даними між Windows NT і VxWorks використовуються псевдодрайвери

TCP / IP в обох системах. Технологія використання двох систем на одному комп'ютері зрозуміла - роботу з об'єктом виконує додаток реального часу, передаючи потім результати роботи додатків Windows NT для обробки, передачі в мережу та інш.

2. Варіант розширень реального часу фірми «VenturCom» виглядає інакше: тут зроблена спроба «інтегрувати» реальний час в Windows NT шляхом дослідження причин затримок та зависань і усунення цих причин за допомогою підсистеми реального часу.

Рішення фірми «VenturCom» (RTX 4.2) базуються на модифікаціях рівня апаратних абстракцій Windows NT (HAL - Hardware Abstraction Layer) - програмного шару, через який драйвери взаємодіють з апаратурою. Модифікований HAL і додаткові функції (RTAPI) відповідають також за стабільність і надійність системи, забезпечуючи відстеження краху Windows NT, зависання додатків або блокування переривань. До складу RTX входить також підсистема реального часу RTSS, за допомогою якої Windows NT розширюється додатковим набором об'єктів (аналогічним стандартним, але з атрибутами реального часу). Серед нових об'єктів - нитки (потоки, процеси) реального часу, які управляються спеціальним планувальником реального часу (256 фіксованих пріоритетів, алгоритм - пріоритетний з витисненням). Побічним результатом RTX є можливість простого створення програм управління пристроями, тому що серед функцій RTAPI є і функції роботи з портами введення-виведення і фізичною пам'яттю. Пропозиції «VenturCom» характерні ще й тим, що вони надають абсолютно екзотичну для NT можливість, а саме - можливість конфігурації Windows NT і створення вбудованих конфігурацій (в т.ч. без дисків, клавіатури і монітора).

5.13. Контрольні запитання до розділу 5

1. Які специфічні властивості характерні для більшості розповсюджених СРЧ.
2. Якому стандарту відповідає інтерфейс прикладних програм, що забезпечує їх функціонування в режимі реального часу.
3. Який механізм планування використовують розповсюджені СРЧ.
4. Який механізм обробки переривань застосований у сучасних СРЧ.
5. Які схеми розробки ПЗ найбільш часто застосовані у існуючих СРЧ.
6. Які переваги і недоліки розробки ПЗ СРЧ під конкретну модель контролера чи конкретне завдання.
7. За якими напрямками відбувається пристосування ОС Linux до вимог реального часу.
8. Яка мотивація використання розширень реального часу для ОС Windows NT при створенні СРЧ.
9. Чи пристосована ОС Windows NT для створення ПЗ жорсткого РЧ.

Розділ 6. Особливості програмування у реальному часі

6.1. Послідовне програмування та програмування задач реального часу

Програма представляє собою опис об'єктів - констант і змінних - і операцій, що здійснюються над ними. Таким чином, програма - це чиста інформація. Її можна записати на будь - який носій, наприклад на папір або на дискету. Програми можна створювати і аналізувати на декількох рівнях абстракції (деталізації) за допомогою відповідних прийомів формального опису змінних і операцій, які виконуються на кожному рівні. На самому нижньому рівні використовуються безпосереднє опис - для кожної змінної вказується її розмір і адреса в пам'яті. На більш високих рівнях змінні мають абстрактні імена, а операції згруповані в функції або процедури. Програміст, що працює на високому рівні абстракції, не повинен думати про те, за якими реальним адресами пам'яті зберігаються змінні, і про машинні команди, що генеруються компілятором.

Послідовне програмування (sequential programming) є найбільш поширеним способом написання програм. Поняття "послідовне" має на увазі, що оператори програми виконуються у відомій послідовності один за одним. Метою послідовної програми є перетворення вхідних даних, заданих в певній формі, у вихідні дані, що мають іншу форму, відповідно до деякого алгоритму - методом вирішення .

Таким чином, послідовна програма працює як фільтр для вхідних даних. Її результат і характеристики повністю визначаються вхідними даними і алгоритмом їх обробки, при цьому часові показники грають, як правило, другорядну роль. На результат не впливають ні інструментальні (мова програмування), ні апаратні (швидкодія ЦП) засоби: від перших залежать зусилля і час, витрачені на розробку і характеристики виконуваного коду, а від других - швидкість виконання програми, але в будь-якому випадку вихідні дані

будуть однаковими.

Програмування в реальному часі (real-time programming) відрізняється від послідовного програмування - розробник програми повинен постійно мати на увазі середовище, в якому працює програма, будь то контролер мікрохвильової печі або пристрій управління маніпулятором робота. У системах реального часу зовнішні сигнали, як правило, вимагають негайної реакції процесора. По суті, однією з найбільш важливих особливостей систем реального часу є час реакції на вхідні сигнали, який повинно задовольняти заданим обмеженням. Спеціальні вимоги до програмування в реальному часі, зокрема необхідність швидкого реагування на зовнішні запити, не можна адекватно реалізувати за допомогою звичайних прийомів послідовного програмування. Насильницьке послідовне розташування блоків програми, які повинні виконуватися паралельно, призводить до неприродної заплутаності результуючого коду і змушує пов'язувати між собою функції, які, по суті, є самостійними. У більшості випадків застосування звичайних прийомів послідовного програмування не дозволяє побудувати систему реального часу. У таких системах незалежні програмні модулі або завдання повинні бути активними одночасно, тобто працювати паралельно, при цьому кожна задача виконує свої специфічні функції. Така техніка відома під назвою паралельного програмування (concurrent programming). У назві робиться наголос на взаємодію між окремими програмними модулями. Паралельне виконання може здійснюватися на одній або декількох ЕОМ, пов'язаних розподіленою мережею. Програмування в реальному часі являє собою розділ мультипрограмування, який присвячений не тільки розробці взаємопов'язаних паралельних процесів, а й часовим характеристикам системи, яка взаємодіє із зовнішнім світом.

Між програмами реального часу і звичайними послідовними програмами, з чітко визначеними входом і виходом, є істотні відмінності. Перерахуємо відмінності програм реального часу від послідовних програм:

1. Логіка виконання програми визначається зовнішніми подіями.
2. Програма працює не тільки з даними, але і з сигналами, які надходять із

зовнішнього світу, наприклад, від датчиків.

3. Логіка розвитку програми може явно залежати від часу.

4. Жорсткі часові обмеження. Неможливість обчислити результат за певний час може виявитися такою ж помилкою, як і невірний результат ("правильну відповідь, отриману пізно - це невірна відповідь").

5. Результат виконання програми залежить від загального стану системи, і його не можна передбачити заздалегідь.

6. Програма, як правило, працює в багатозадачному режимі. Відповідно, необхідні процедури синхронізації і обміну даними між процесами.

7. Виконання програми не закінчується після вичерпання вхідних даних - вона завжди чекає надходження нових даних.

Важливість фактору часу не слід розуміти як вимога високої швидкості виконання програми. Швидкість виконання програми реального часу повинна бути достатньою для того, щоб в рамках встановлених обмежень реагувати на вхідні дані і сигнали і виробляти відповідні вихідні величини. "Повільна" система реального часу може чудово керувати повільним процесом. Тому швидкість виконання програм реального часу необхідно розглядати щодо керованого процесу або необхідної швидкості. Типові додатки автоматизації виробничих процесів вимагають гарантований час відповіді близько 1 мс, а в окремих випадках - близько 0.1 мс. При програмуванні в реальному часі особливо важливими є ефективність і час реакції програм. Відповідно, розробка програм тісно пов'язана з параметрами операційної системи, а в розподілених системах - і локальної мережі.

Особливості програмування в реальному часі вимагають спеціальної техніки і методів, що не використовуються при послідовному програмуванні, які відносяться до впливу на виконання програми зовнішнього середовища і часових параметрів. Найбільш важливими з них є перехоплення переривань, обробка виняткових (позаштатних) ситуацій і безпосереднє використання функцій операційної системи (виклики ядра з прикладної програми, минаючи стандартні засоби). Крім цього при програмуванні в реальному часі

використовуються методика мультипрограмування і модель "клієнт-сервер", оскільки окремий процес або потік зазвичай виконують тільки деяку самостійну частину всього завдання.

6.2. Середовище програмування

Розглянемо середовище, у якому виконуються програми. Середовище виконання може варіюватися від міні - персональних і одноплатних мікрокомп'ютерів і локальних шин, пов'язаних з навколишнім середовищем через апаратні інтерфейси, до розподілених систем "клієнт-сервер" з централізованими базами даних та доступом до системи високопродуктивних графічних робочих станцій. У комплексній системі управління промисловими і технологічними процесами може одночасно використовуватися все перераховане обладнання.

Різноманітність апаратного середовища відбивається і в програмному забезпеченні, яке включає в себе як програми, записані в ПЗУ, так і комплексні операційні системи, що забезпечують розробку і виконання програм. У великих системах створення і виконання програм здійснюються на одній і тій же ЕОМ, а в деяких випадках навіть в один час. Невеличкі системи можуть не мати засобів розробки, і програми для них повинні створюватися на більш потужних ЕОМ з подальшим завантаженням в виконуючу систему. Те ж стосується і мікропрограм, що "зашиті" в ПЗУ обладнання виробником (firmware), - вони розробляються на ЕОМ, відмінною від тієї, на якій виконуються.

Першим завданням програміста є ознайомлення з програмним середовищем і доступними інструментальними засобами. Проблеми, з якими доводиться стикатися, починаються, наприклад, з типу представлення даних в апаратурі і програмах, оскільки в одних системах застосовується прямий, а в інших - інверсний порядок зберігання біт або байт в слові (молодші байти зберігаються в старших адресах). Таких тонкощів дуже багато, і досвідчений програміст має

знати, як відокремити загальну структуру даних і код від технічних деталей реалізації в конкретному апаратному середовищі.

Важливо якомога раніше з'ясувати функції, що забезпечуються наявним середовищем, і можливі альтернативи. Наприклад, мікропроцесор Motorola 68000 має в своєму наборі команд інструкцію “test_and_set”, і тому зв'язок між завданнями може здійснюватися через загальні області пам'яті. Операційна система VAX / VMS підтримує поштові скриньки і синхронізувати процеси можна за допомогою механізму передачі повідомлень.

В UNIX та інших операційних системах зв'язок між процесами найзручніше здійснювати через канали. При розробці програм для середовища UNIX слід прагнути, з одного боку, максимально ефективно використовувати її особливості, наприклад стандартну обробку вхідних і вихідних даних, а з іншого - забезпечити переносимість між різними версіями UNIX.

Через те, що багатозадачні системи і системи реального часу розробляються колективами програмістів, необхідно з самого початку домагатися ясності, які методи і прийоми будуть використовуються.

Структурування апаратних і програмних ресурсів, тобто привласнення адрес на шині і пріоритетів переривань для інтерфейсних пристроїв, має важливе значення. Неправильний порядок розподілу ресурсів може призвести до тупикових ситуацій. Визначення апаратних адрес і відносних пріоритетів переривань не залежить від розроблюваної програми. Тому воно повинно бути вироблено на ранній стадії і зафіксовано в технічному завданні. Його не слід відкладати до моменту безпосереднього кодування, так як в цьому випадку неминучі конфлікти між програмними модулями і виникає ризик тупикових ситуацій.

Правильним практичним вирішенням цієї проблеми є використання в програмі тільки логічних імен для фізичного обладнання та його параметрів і таблиць відповідності між ними і реальними фізичними пристроями. При цьому зміна адреси шини або пріоритету пристрою вимагає не модифікації, а в гіршому випадку тільки нової компіляції програми. Розумно також

використовувати структуровану і організаційно оформлену угоду про найменування системних ресурсів і програмних змінних. Те саме можна сказати і про найменування і визначення адрес віддалених пристроїв у розподілених системах.

Програми слід будувати на засадах, що застосовуються в операційних системах, - на основі модульної і багаторівневої структури, оскільки це істотно спрощує розробку складних систем. Повинна бути визначена специфікація окремих модулів, починаючи з інтерфейсів між апаратними та програмними компонентами системи. До основної інформації про інтерфейси відноситься і структура повідомлень, якими будуть обмінюватися програмні модулі. Це не означає, що зміни у визначенні інтерфейсів не можуть вводитися після початку розробки програми. Але чим пізніше вони вносяться, тим більше витрат зажадає зміна коду, тестування і т. д. З іншого боку, слід бути готовим до того, що деякі зміни специфікацій все одно будуть відбуватися в процесі розробки програми, оскільки просування в роботі дозволяє краще побачити проблему.

Треба брати до уваги ефективність реалізації функцій операційної системи. Не можна вважати, що спосіб, яким в операційній системі реалізовані ті чи інші послуги, даний раз і назавжди. Для перевірки того, наскільки добре задовольняються часові обмеження, бажано провести оцінку, наприклад за допомогою еталонних тестових програм. Якщо результати тестів неприйнятні, то одним з рішень може бути розробка програм, що заміщають відповідні стандартні модулі операційної системи. Таке рішення вимагає дуже обережного і диференційованого підходу, зокрема заміщення може виконуватися не завжди, а тільки для певних процесів.

6.3. Структура програми реального часу

Розробка програми реального часу починається з аналізу та опису завдання. Функції системи поділяються на прості частини, з кожною з яких зв'язується

програмний модуль. Наприклад, завдання для управління рухом маніпулятора робота можна організувати таким чином:

- зчитати з диску опис траєкторій маніпулятора;
- розрахувати наступне положення маніпулятора (опорне значення);
- зчитати за допомогою датчиків поточний стан;
- розрахувати необхідний сигнал управління;
- виконати дію по управлінню;
- перевірити, чи опорне значення і поточний стан збігаються в межах заданої точності;
- отримати дані від оператора;
- зупинити робота в разі нештатної ситуації (наприклад, сигнал переривання від аварійної кнопки).

Принциповою особливістю програм реального часу є постійна готовність і відсутність умов нормального, а не аварійного завершення. Якщо програма не виконується і не обробляє дані, вона залишається в режимі очікування переривання / події або закінчення деякого інтервалу часу. Програми реального часу - це послідовний код, що виконується в нескінченному циклі.

В якомусь місці програми є оператор, який призупиняє виконання до настання зовнішнього події або закінчення інтервалу часу. Зазвичай програма структурується таким чином, що оператор “end” ніколи не досягається:

while true do (* нескінченний цикл *)
begin (* процедура обробки *)
wait event at # 2,28 (* зовнішнє переривання *) (* код обробки *) ... end;
(*Процедура обробки *)
end. (* Вихід з програми; ніколи не досягається *)

При розробці кожного програмного модуля повинні бути чітко виділені області, в яких відбувається звернення до захищених ресурсів (критичні секції). Вхід і вихід з цих областей координується будь-яким методом синхронізації або міжпрограмних комунікацій, наприклад за допомогою семафорів. У загальному випадку, якщо процес перебуває в критичній секції, можна вважати, що дані, з

якими він працює, не змінюються будь-яким іншим процесом. Переривання виконання процесу не повинно впливати на захищені ресурси. Це знижує ризик системних помилок.

Аналогічних заходів необхідно дотримуватися і для потоків, породжуваних як дочірніх процесів головного процесу. Різні потоки можуть використовувати загальні змінні породившого їх процесу, і тому програміст повинен вирішити, захищати ці змінні чи ні.

Для гарантії живучості програми нештатні ситуації, які можуть блокувати або аварійно завершити процес, повинні своєчасно розпізнаватися і виправлятися, якщо це можливо, в рамках самої програми.

У системах реального часу різні процеси можуть звертатися до загальних підпрограм. При простому вирішенні ці підпрограми зв'язуються з відповідними модулями після компіляції. При цьому в пам'яті зберігається кілька копій однієї підпрограми. При іншому підході в пам'ять завантажується лише одна копія підпрограми, але доступ до неї можливий з різних програм. Такі підпрограми повинні бути повторно входженні - реентерабельні (reentrant), тобто допускати багаторазові виклики і припинення виконання, які не впливають один на одного. Ці програми повинні використовувати тільки регістри процесора або пам'ять викликаючих процесів, тобто не мати локальних змінних. В результаті реентерабельний модуль, що розділяється декількома процесами, можна перервати в будь-який час і продовжити з іншої точки програми, оскільки він працює зі стеком процесу, що його викликав. Таким чином, реентерабельна процедура може виявитися одночасно в контексті кількох різних процесів.

Ефективність виконання є одним з найбільш важливих параметрів систем реального часу. Процеси повинні виконуватися швидко, і часто доводиться шукати компроміс між ясністю і структурованістю програми і її швидкодією. Життєвий досвід показує, що якщо для досягнення мети потрібно чимось пожертвувати, то це зазвичай і робиться. Не завжди виникає суперечність між структурністю і ефективністю, але якщо перше повинно бути принесено в

жертву другого, необхідно повністю документувати всі прийняті рішення, інакше істотно ускладнюється подальший супровід програми.

6.4. Паралельне програмування, мультипрограмування і багатозадачність

Програмування в реальному часі вимагає одночасного виконання декількох процесів або завдань на одній ЕОМ. Ці процеси використовують спільно ресурси системи, але більш - менш незалежно один від одного.

Мультипрограмування (multiprogramming) або багатозадачність (multitasking) є способом одночасного виконання декількох процесів. Цього ефекту можна домогтися як для одного, так і для декількох процесорів: процеси виконуються або на одному, або на декількох пов'язаних між собою процесорах. Насправді, багато сучасних обчислювальних систем складаються з декількох процесорів, пов'язаних між собою або мережею передачі даних, або загальною шиною.

Для запису паралельних процесів можна використовувати наступну нотацію:

cobegin

x: = 1; x: = 2; x: = 3;

coend;

write (x);

Виконання команд між ключовими словами **cobegin** і **coend** відбувається паралельно. Пара операторних дужок **cobegin-coend** призводить до генерації потоків в рамках багатозадачної системи. Оператор **cobegin** не накладає умов на відносний порядок виконання окремих процесів, а оператор **coend** досягається тільки тоді, коли всі процеси всередині блоку завершені. Якби виконання було послідовним, то остаточне значення змінної **x** мало було б дорівнювати 3. Для паралельних процесів кінцевий результат однозначно передбачити не можна; завдання виконуються, принаймні, з зовнішньої точки

зору, у випадковій послідовності. Тому остаточне значення **x** в наведеному прикладі може бути як 1, так і 2 або 3.

Іноді в технічній літературі термін "паралельне програмування" використовується як синонім мультипрограмування. Однак ці поняття не однакові за змістом. Паралельне програмування - це абстрактний процес розробки програм, який потенційно може виконуватися паралельно, незалежно від програмно-апаратного середовища. Іншими словами, передбачається, що кожна задача реалізується на власному віртуальному процесорі. З іншого боку, мультипрограмування є практичним способом виконання декількох програм на одному центральному процесорі або в розподіленій обчислювальній системі. Паралельне програмування більш трудомістке, ніж послідовне, оскільки здатність людини стежити за розвитком пов'язаних процесів, і досліджувати їх взаємодію, обмежена.

Програмування в реальному часі засновано на паралельному програмуванні і включає в себе техніку підвищення ефективності і швидкості виконання програм - управління перериваннями, обробку винятків і безпосереднього використання ресурсів операційної системи. Крім того, програми реального часу вимагають спеціальних методів тестування.

6.5. Вимоги до мов програмування реального часу

Основними критеріями при виборі мови для розробки програми реального часу є:

1. Отримання найвищої продуктивності додатка реального часу. З цієї вимоги випливає, що мова повинна бути компилюємого (як C, C ++), а не інтерпретуємого (як Java) типу, і для нього повинен існувати компілятор з високим ступенем оптимізації коду. Для сучасних процесорів якість компілятора особливо важлива, оскільки для них оптимізація може прискорювати роботу програми в кілька разів у порівнянні з не оптимізованим варіантом, причому, часто оптимізуючий компілятор може породити код

швидший, ніж написаний на асемблері. Технології оптимізації розвиваються досить повільно і часто потрібні роки на розробку високоефективного компілятора. Тому зазвичай для старих, з більш простою структурою мов є більш якісні компілятори, а ніж для досить молодих, і складно влаштованих мов.

2. Отримання доступу до ресурсів обладнання за допомогою мовних конструкцій, або за допомогою наявних для вибраної мови бібліотечних функцій.

3. Можливість виклику процедур, написаних іншою мовою, наприклад, на мові асемблера. З цієї вимоги випливає, що послідовність виклику підпрограм (механізм іменування об'єктів, передачі аргументів і отримання значення, що повертається) повинна бути документально підтверджена для вибраної мови.

4. Переносимість додатків, під яким зазвичай розуміють, можливість скомпілювати додаток іншим компілятором, що є на тій же платформі, так і можливість його скомпілювати на іншій платформі і / або іншої операційної системи.

5. Підтримка об'єктно - орієнтованого підходу стала останнім часом необхідністю, часто виходячи зі списку вимог на перше місце. Це пояснює використання іноді мови Java разом з ОС РЧ.

Програмування в реальному часі вимагає спеціальних засобів, які не завжди зустрічаються в звичайних мовах послідовного програмування. Мова або операційна система для програмування в реальному часі повинні надавати такі можливості:

- опис паралельних процесів;

- перемикання процесів на основі динамічних пріоритетів, які можуть змінюватися, в тому числі і прикладними процесами;

- синхронізацію процесів;

- обмін даними між процесами;

- функції, пов'язані з годинником та таймером, абсолютний і відносний час очікування;

- прямий доступ до зовнішніх апаратних портів;

- обробку переривань;

- обробку вийнятків.

Мало які мови забезпечують всі ці можливості. Більшість мов має лише частину з них, хоча для певних програм цього виявляється достатньо. Деякі компанії розробили спеціальні мови для підтримки своїх власних апаратних засобів. Ці мови не претендують на універсальність і орієнтовані швидше на конкретні ЕОМ і їх інтерфейси. Зазвичай вони базуються на існуючих мовах - FORTRAN, BASIC - з розширеннями, що включають функції реального часу, про що свідчить їх назви типу "Process BASIC" і "Real-time FORTRAN". Деякі мови не підтримують програмування в реальному часі в строгому сенсі, але вони легко розширюються, наприклад С та С ++.

У 1970-ті роки широку підтримку отримала концепція єдиної переносимої багатоцільової мови програмування. В результаті була розроблена мова ADA. Її головна ідея полягає в тому, що навколишнє середовище програмування, тобто мова, повинна бути повністю відокремлена від апаратних засобів. Програміст не повинен стикатися з деталями машинного рівня, а працювати тільки в термінах абстрактних структур і типів даних.

Досвід показав не реалістичність такого підходу. Універсальні, сильно типізовані мови програмування гарантують певний рівень надійності програми, але в той же час обмежують гнучкість. Швидкий розвиток технічних засобів, висуває нові вимоги, які не могли бути передбачені в існуючих мовах, і багато програмістів відчувають ці обмеження, використовуючи, не самі сучасні мови програмування. Ціна надійності мови - складність і громіздкість, а генерується при цьому компілятором код - надлишковий і малоефективний. Відкрита мова типу С, заснована на обмеженій кількості базових ідей, має більшу гнучкість і надає досвідченому програмісту більше можливостей. Не існує найкращої мови, для кожної програми і середовища, тому необхідно підбирати свої засоби і при цьому враховувати кваліфікацію і переваги розробників.

6.6. Контрольні запитання до розділу 6

1. Які суттєві відмінності послідовного програмування від програмування задач реального часу.
2. Охарактеризуйте особливості середовища, у якому виконуються програми реального часу.
3. Як впливає фактор часу на виконання програм реального часу.
4. Чи припустиме аварійне завершення програми реального часу.
5. Які вимоги до мови програмування задач реального часу.
7. Які спеціальні можливості мають надавати мова програмування або операційна система для програмування задач реального часу.
8. Яким основним критеріям має відповідати мова програмування задач реального часу.

Розділ 7. Асинхронна і синхронна обробки даних

7.1. Обробка переривань і виключень

Системи реального часу з'єднані із зовнішнім середовищем (фізичний процес) через апаратні інтерфейси. Доступ до інтерфейсів і зовнішнім даним здійснюється або за опитуванням, або по перериванню.

При опитуванні програма повинна циклічно послідовно перевіряти всі вхідні порти на наявність у них нових даних, які потім зчитуються і обробляються. Черговість і частота опитування визначають час реакції системи реального часу на вхідні сигнали. Опитування є простим, але неефективним методом через повторюваність перевірок вхідних портів.

Отримання даних по перериванню відбувається інакше. Інтерфейсний пристрій, що одержав нові дані, привертає увагу центрального процесора, посилаючи йому сигнал переривання через системну шину. По відношенню до поточного процесу переривання є асинхронними подіями, які вимагають негайної реакції. Отримавши сигнал переривання, процесор зупиняє виконання поточного процесу, зберігає в стеку його контекст, зчитує з таблиці адресу програми обробки переривання і передає їй управління. Ця програма називається обробником переривання. Переривання відбувається в довільній точці потоку команд програми, що програміст не може прогнозувати.

Переривання мають деяку подібність із процедурою в тому, що в обох випадках виконується деяка підпрограма, що обробляє спеціальну ситуацію, а потім триває виконання основної гілки програми.

Залежно від джерела, переривання поділяються на три великих класи:

-зовнішні;

-внутрішні;

-програмні.

Зовнішні переривання виникають у результаті дій користувача або в результаті надходження сигналів від апаратних пристроїв. Даний клас

переривань є **асинхронним** стосовно потоку інструкцій що переривається програми. Апаратура процесора працює так, що асинхронні переривання виникають між виконанням двох сусідніх інструкцій, при цьому система після обробки переривання продовжує виконання процесу, уже починаючи з наступної інструкції.

Внутрішні переривання (**виключення)**, відбуваються **синхронно** виконанню програми з появою аварійної ситуації в ході виконання деякої інструкції програми. Прикладами є ділення на нуль, помилки захисту пам'яті, звертання до неіснуючої адреси. Переривання виникають усередині виконання команди.

Програмні переривання відрізняються від попередніх двох класів тим, що вони по своїй суті не є «щирими» перериваннями. Програмне переривання виникає при виконанні особливої команди процесора, виконання якої імітує переривання, тобто перехід на нову послідовність інструкцій по ступені важливості й терміновості. Про переривання, що мають однакове значення пріоритету, говорять, що вони ставляться до **одного рівня пріоритету** переривань.

Процедури, викликувані по перериваннях, звичайно називають **оброблювачами переривань,** або **процедурами обслуговування переривань**.

Апаратні переривання обробляються драйверами відповідних зовнішніх пристроїв, внутрішні переривання - спеціальними модулями ядра, а програмні переривання - процедурами ОС, що обслуговують системні виклики.

Крім цих модулів в ОС може перебувати так званий диспетчер переривань, що координує роботу окремих оброблювачів переривань.

Узагальнено послідовність дій апаратних і програмних засобів по обробці переривання можна описати в такий спосіб:

1. При виникненні сигналу (для апаратних переривань) або умови (для внутрішніх переривань) переривання відбувається первинне апаратне розпізнавання типу переривання. Якщо переривання даного типу в даний

момент заборонені (пріоритетною схемою або механізмом маскування), то процесор продовжує підтримувати природний хід виконання команд.

У противному випадку залежно від інформації, що надійшла в процесор, відбувається автоматичний виклик процедури обробки переривання, адреса якої перебуває в спеціальній таблиці ОС, розташованої або в регістрах процесора, або в певному місці оперативної пам'яті.

2. Автоматично зберігається деяка частина контексту перерваного потоку, що дозволить ядру відновити виконання потоку процесу після обробки переривання. У цю підмножину включаються значення лічильника команд слова стану машини, що зберігає ознаки основних режимів роботи процесора (приклад слова - регістр EFLAGS в Intel Pentium), а також декількох регістрів загального призначення, які потрібні програмі обробки переривання. Може бути збережений і повний контекст процесу, якщо ОС обслуговує дане переривання зі зміною процесу.

3. Одночасно із завантаженням адреси процедури обробки переривань у лічильник команд може автоматично виконуватися завантаження нового значення слова стану машини, що визначає режими роботи процесора при обробці переривання, у тому числі роботу в привілейованому режимі.

4. Тимчасово забороняються переривання даного типу, щоб не утворилася черга вкладених друг у друга потоків однієї й тієї ж процедури. Деталі виконання цієї операції залежать від особливостей апаратної платформи, наприклад може використовуватися механізм маскування переривань.

5. Після того, як переривання оброблене ядром ОС, перерваний контекст відновлюється, і робота потоку відновляється з перерваного місця.

Частина контексту відновлюється апаратно по команді повернення з переривань (наприклад, адреса наступної команди й слово стану машини), а частина - програмним способом, за допомогою явних команд добування даних зі стека. При поверненні з переривання блокування повторних переривань даного типу знімається.

Програмні переривання.

Програмне переривання реалізує один зі способів переходу на підпрограму за допомогою спеціальної інструкції процесора, такий як INT у процесорах Intel Pentium, trap у процесорах Motorola, і т.п.

При виконанні команди програмного переривання процесор відпрацьовує ту ж послідовність дій, що й при виникненні зовнішнього або внутрішнього переривання, але тільки відбувається це в передбачуваній точці програми - там, де програміст помістив дану команду.

Практично всі сучасні процесори мають у системі команд інструкції програмних переривань.

Програмні переривання часто використовуються для виконання обмеженої кількості викликів функцій ядра операційної системи, тобто системних викликів.

Диспетчеризація й пріоритезація переривань в ОС

Переривання виконують корисну для обчислювальної системи функцію - дозволяють реагувати на асинхронні стосовно обчислювального процесу події.

Переривання створюють додаткові труднощі для ОС в організації обчислювального процесу.

Для впорядкування роботи оброблювачів переривань в ОС застосовується той же механізм, що й для впорядкування роботи користувальницьких процесів - механізм пріоритетних черг. Всі джерела переривань діляться на кілька класів, причому кожному класу присвоюється пріоритет. В ОС виділяється програмний модуль, що займається диспетчеризацією оброблювачів переривань. Цей модуль у різних ОС називається по-різному, ми на самому початку визначимо його, як *диспетчер переривань.*

При виникненні переривання диспетчер переривань викликається першим. Він забороняє на якийсь час всі переривання і з'ясовує причину переривання. Потім порівнюється пріоритет джерела переривання з поточним пріоритетом потоку команд, виконуваних процесором. Якщо пріоритет нового запиту вище

поточного, то виконання поточного потоку припиняється й виконується обробка переривання. У противному випадку запит ставиться в чергу.

Процедури обробки переривань і поточний процес

Важливою особливістю процедур, виконуваних по запитах переривань, є те, що вони виконують роботу, найчастіше ніяк не пов'язану з поточним процесом.

Наприклад, драйвер диска може одержати керування після того, як контролер диска записав у відповідні сектори інформацію, отриману від процесу А, але цей момент часу, не збіжиться з періодом чергової ітерації виконання процесу А або його потоку.

У найбільш типовому випадку процес А буде перебувати в стані очікування завершення операції вводу-виводу (при синхронному режимі виконання цієї операції) і драйвер диска прерве який - небудь інший процес.

У деяких випадках взагалі важко однозначно визначити, для якого процесу виконує роботу той або інший програмний модуль ОС, наприклад планувальник потоків. Тому для такого роду процедур вводяться обмеження - вони не мають права використовувати ресурси (пам'ять, відкриті файли й т.п.), з якими працює поточний процес.

Процедури обробки переривань працюють із ресурсами, які були виділені їм при ініціалізації відповідного драйвера або ініціалізації самої операційної системи. Ці ресурси належать ОС, а не конкретному процесу. Так пам'ять драйверам виділяється із системної області. Тому звичайно говорять, що процедури обробки переривання працюють поза контекстом процесу.

Диспетчеризація переривань є важливою функцією ОС, і ця функція реалізована практично у всіх мультипрограмних ОС. Як правило, в ОС реалізується дворівневий механізм планування робіт. Верхній рівень планування виконується диспетчером переривань, що розподіляє процесорний час між потоком вступників запитів на переривання різних типів - зовнішніх, внутрішніх і програмних. процесорний час, що залишився, розподіляється іншим диспетчером -

диспетчером потоків, на підставі дисциплін квантування й інших, які ми розглядали.

Системні виклики

Системний виклик дозволяє додатку звернутися до ОС із проханням виконати ту або іншу дію, що оформлена як процедура (або набір процедур) кодового сегмента ОС.

Для прикладного програміста ОС виглядає як якась бібліотека, що реалізує корисні функції, що полегшують керування прикладним завданням або виконання дій, заборонених у користувальницькому режимі, наприклад обмін даними із пристроєм вводу - виводу.

Реалізація системних викликів повинна задовольняти наступним вимогам:

- забезпечувати перемикання в привілейований режим;
- мати високу швидкість виклику процедур ОС;
- забезпечувати однакове звертання до системних викликів для всіх апаратних платформ, на яких працюють ОС;
- допускати легке розширення набору системних викликів;
- забезпечувати контроль із боку ОС за коректним використанням системних викликів.

У більшості ОС системні виклики обслуговуються за централізованою схемою, заснованої на існуванні диспетчера системних викликів.

При будь - якому системному виклику додаток виконує програмне переривання з певним і єдиним номером вектора. Перед виконанням програмного переривання додаток передає ОС номер системного виклику. Спосіб передачі залежить від реалізації. Наприклад, номер можна помістити в певний регістр процесора або передати через стек. Також деяким способом передаються аргументи системного виклику, вони можуть міститися як у регістрі загального призначення, так і передаватися через стек або масив оперативної пам'яті.

Після завершення роботи системного виклику керування повертається диспетчеру, при цьому він одержує також код завершення цього виклику. Диспетчер відновлює регістри процесора, поміщає в певний регістр код повернення й виконує інструкцію повернення з переривання, що відновлює непривілейований режим роботи процесора.

Описаний табличний спосіб організації системних викликів прийнятий практично у всіх операційних системах. Він дозволяє легко модифікувати склад системних викликів, просто додавши в таблицю нову адресу й розширивши діапазон припустимих номерів викликів. ОС може виконувати системні виклики в **синхронному** або **асинхронному** режимах (Рис. 7.1).

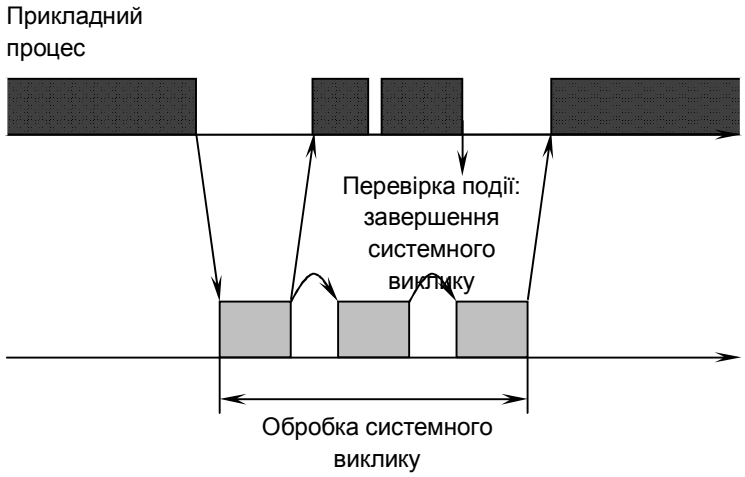
Синхронний системний виклик означає, що процес, що зробив такий виклик, припиняється доти, поки системний виклик не виконає всю необхідну роботу. Після цього планувальник переводить процес у стан готовності.

Асинхронний системний виклик не приводить до переводу процесу в режим очікування після виконання деяких початкових системних дій, наприклад запуску операції виводу-виводу, керування повертається прикладному процесу. Більшість системних викликів в ОС є синхронними.

Інший варіант обробки переривань полягає в тому, що планувальник вибирає з черги очікування цієї події або переривання наступний процес і переводить його в чергу готових процесів.

Коли процесор передає управління оброблювачу переривань, він зазвичай зберігає тільки лічильник команд і покажчик на стек з поточною діяльністю. Оброблювач переривань повинен зберегти в тимчасових буферах або в стеку всі регістри, які він збирається використовувати, і відновити їх в кінці. Ця операція критична до часу і, як правило, вимагає заборони переривань для того, щоб уникнути перемикання процесів під час її виконання.

а) Асинхронний системний виклик



б) Синхронний системний виклик

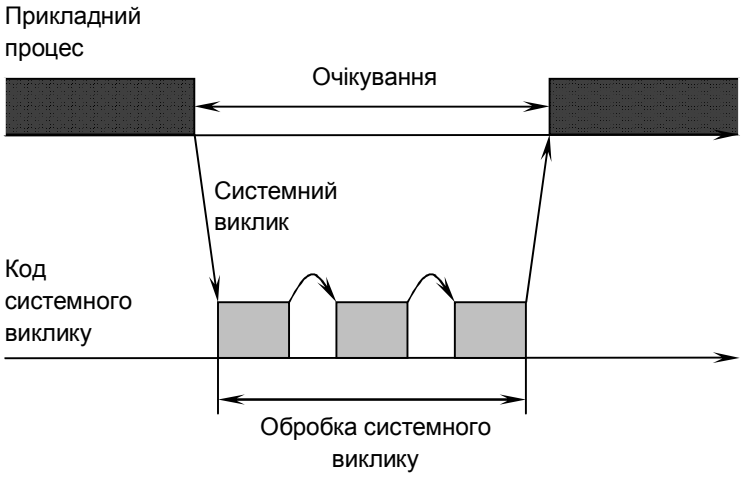


Рис.7.1. Системні виклики

При управлінні перериваннями час реакції має бути якомога менше. Він являє собою суму часу, необхідного процесору, щоб зреагувати на переривання (латентність переривання), і часу, необхідного на перемикання контексту до запуску обробника переривань. Типове завантаження системи також грає певну роль. Якщо система повинна обслуговувати багато одночасних переривань, знову надходжувані переривання чекатимуть у черзі, поки процесор не звільниться.

Реакція на виключення (exceptions) схожа на обробку переривань. **Виключеннями** називаються нештатні ситуації, коли процесор не може

правильно виконати команду. Прикладом виключення є поділ на нуль або звернення за неіснуючою адресою. В англомовній літературі для різних видів винятків застосовуються терміни **trap, fault, abort** (не плутати з "взаємним винятком" - **mutual exclusion**).

Зазвичай операційна система обробляє виключення, припиняючи поточний процес, і виводить повідомлення, чітко описує ситуацію, на пристрій відображення, звичайно монітор або принтер. Прийнятна при послідовній інтерактивній багатокористувацькій обробці, раптова зупинка процесу в системах реального часу повинна бути абсолютно виключена. Не можна допустити, щоб керовані мікропроцесором автопілот літака або автоматична гальмівна система автомобіля (Automatic Braking System - ABS), раптово припинили роботу через ділення на нуль. У системах реального часу всі можливі винятки повинні аналізуватися заздалегідь з визначенням відповідних процедур обробки.

Складною проблемою при обробці виключень є перевірка, того що виключення не виникне знову після того, як воно було оброблено. Або іншими словами, обробка виключень повинна займатися причиною, а не симптомами аномальної ситуації. Якщо виняток оброблено некоректно, він може виникнути знову, змушуючи процесор знову і знову переходити до модулю обробки. Наприклад, обробник ділення на нуль повинен перевіряти і змінювати операнди, а не просто відновлювати виконання з місця, що передує помилці, що призведе до нескінченного циклу.

Фактичні адреси програмних модулів відомі тільки після їх завантаження. При запуску системи в таблицю обробки переривань записуються адреси пам'яті, куди завантажуються обробники, які потім викликаються по посиланнях з цієї таблиці.

7.2. Програмування операцій очікування

Процес реального часу може явно чекати закінчення деякого інтервалу (відносний час) або настання заданого моменту (абсолютний час). Відповідні функції зазвичай мають наступний формат:

wait (n) або **wait until (час),**

де n - інтервал в секундах або мілісекундах, а змінна "час" має формат години, хвилини, секунди, мілісекунди.

Коли виконується одна з цих функцій, операційна система поміщає процес в чергу очікування. Після закінчення / настання заданого часу процес переводиться в чергу готових процесів.

Поширений, але не кращий метод організації часової затримки - цикл, контроль системного часу в циклі зайнятого очікування:

repeat (* холостий хід *)

until (time = 12:00:00);

Як правило, подібні активні цикли очікування є марнування процесорного часу, і їх слід уникати. Однак є винятки. В системі, де аналого - цифрове перетворення займає 20 мкс, а операція перемикання процесів - 10 мкс, більш економно організувати очікування на 20 мкс. перед тим, як вводити нові дані, ніж починати процедуру перемикання процесів, неявно вважаючи "гарною" операцією очікування. Кожен випадок вимагає індивідуального підходу - для цього зазвичай потрібне добре знання системи і розвинене чуття.

Важливою особливістю процесів, що запускаються періодично, наприклад, фільтрація і алгоритми регулювання, - є накопичення помилки часу. Це пов'язано з тим, що процес з черги очікування події знову потрапляє в чергу, але вже готових процесів і повинен чекати деякий випадковий інтервал часу, перш ніж отримає управління. Необхідний і фактичний час пробудження процесу не збігаються. Помилки очікування накопичуються, якщо цей час розраховується так: **новий час пробудження = часу початку очікування + інтервал**. За таким алгоритмом працює холостий цикл, наприклад, "чекати 10 секунд". Накопичена часова помилка є сумою часу, проведеного в черзі, і часу, необхідного для безпосереднього виконання. Правильне рішення виходить,

якщо відлік ведеться від моменту попереднього пробудження.: **новий час пробудження = час попереднього пробудження + інтервал.**

Таким чином, відносний час перетворюється в абсолютний. На практиці необхідні дві команди:

wait until (ref_time);

ref_time: = ref_time + 10 seconds;

7.3. Внутрішні підпрограми операційної системи

Типовою ситуацією при програмуванні в реальному часі є безпосереднє звернення до підпрограм операційної системи через те, що в використовуваній мові програмування відсутній еквівалентний засіб. Звернення до функцій операційної системи також необхідні при роботі в мережевому і розподіленому середовищі. Операційна система відповідає за все обслуговування прикладних задач, включаючи файлові і мережеві операції. Просте звернення до операційної системи може призвести до складної послідовності дій при доступу до віддаленої бази даних, включаючи всі супутні перевірки і операції управління, які позбавляють прикладну програму від зайвих деталей. Інтерфейс операційної системи робить виконання таких операцій більш прозорим і спрощує написання складних програм.

Багато мов програмування високого рівня, наприклад С, забезпечують інтерфейс з операційною системою для безпосереднього виклику її модулів з виконуваних процесів. Існують різні види програмних інтерфейсів з операційною системою: безпосередні виклики, примітиви і доступ через бібліотечні модулі.

Безпосередні (системні) виклики здійснюються за допомогою конструкції мови високого рівня, яка передає управління підпрограмі, яка є частиною операційної системи. Необхідні параметри передаються списком, як при звичайному зверненні до підпрограми. Після завершення системної процедури результат повертається викликаючий програмі.

Так як в багатозадачному середовищі системні програми і примітиви можуть викликатися одночасно різними процесами, їх код завжди реентерабельний. Це дозволяє уникнути конфліктів при перериванні системної програми іншим запитом, що вимагає ту ж послугу з іншого контексту.

У деяких випадках для доступу до внутрішніх ресурсів операційної системи можна використовувати бібліотечні модулі. Ці модулі вже попередньо відкомпільовані, і їх залишається тільки пов'язати з основною програмою. Необхідно перевірити по документації системи необхідні параметри, а також механізми їх передачі та редагування зв'язків в мові високого рівня.

7.4. Пріоритети процесів і продуктивність системи

Багатозадачна операційна система реального часу повинна допускати призначення пріоритетів виконуваним процесам. Зазвичай пріоритети є динамічними, що означає, що під час виконання вони можуть змінюватися як самими процесами, так і операційною системою. Зазвичай існують певні обмеження і механізми контролю, які визначають, хто і як може змінювати пріоритети. Призначення пріоритетів чинить серйозний вплив на роботу системи в цілому.

Найбільш важливі процеси або процеси, час реакції яких жорстко обмежений, отримують більш високий пріоритет. До останніх відносяться обробники переривань. Завдання, які виконують менш важливі дії, наприклад друк, отримують більш низький пріоритет. Очевидно, що необхідно звертати увагу на угоди, які використовуються в системі щодо того, чи пов'язаний вищий пріоритет з більшим чи меншим числом. Пріоритети мають відносне значення і впливають тільки тоді, коли існують процеси з різними пріоритетами.

У системах реального часу реакція на переривання відокремлена від обчислень, що вимагає значних ресурсів процесора. Як тільки відбувається подія або переривання, його обробник негайно включається в чергу готових процесів. Програми обробників переривань зазвичай компактні, так як вони

повинні забезпечувати швидку реакцію, наприклад введення нових даних, і передавати управління більш складним процесам, вони інтенсивно споживають ресурси процесора, які виконуються з більш низьким пріоритетом.

У прикладі системи управління маніпулятором робота одна задача, яку можна побудувати як обробник переривань, чекає надходження від датчика нових даних про поточний стан маніпулятора. Коли надходить переривання від датчика - є нові дані, - це завдання має відразу отримати управління. Потім вона передає дані про стан програми їх обробки, що вимагає великих обчислювальних ресурсів. Ця програма не відповідає за обробку переривань і може використовувати більше часу для обчислень.

Продуктивність системи реального часу значно складніше піддається оцінці, ніж систем, що використовують звичайні послідовні програми. Якщо звичайна послідовна програма виконується на конкретному процесорі з відомою швидкістю, то програма реального часу залежить від поведінки навколишнього середовища, тобто керованих технічних процесів. Загальна продуктивність системи повинна бути достатньою для того, щоб виконувати всі операції і видавати результати за встановлений час. Іншими словами, система реального часу завжди повинна бути готова до максимального навантаження, яке може створити технічний процес.

7.5. Контрольні запитання до розділу 7

1. Яким способом може бути організовано доступ до зовнішніх даних у СРЧ.

2. У чому полягає проблема реалізації “виключень”.

3. Які особливості програмування операцій очікування при створенні ПЗ.

4. Як реалізувати безпосереднє звернення до підпрограм ОС при створенні ПЗ РЧ.

5. Чи допускає ОС РЧ призначати пріоритети виконуваним програмам.

6. Чи відокремлена реалізація обчислень від реакції на переривання у ПЗ РЧ.

7. Від яких факторів залежить термін виконання задачі реального часу.

8. Прийняття яких рішень може суттєво вплинути на час відгуку прикладної задачі ПЗ РЧ.

Розділ 8. Визначення часу виконання програм

8.1. Особливості планування

Більшістю розробників стверджується, що планувальник в СРЧ має дві складові: планувальник періоду розробки (off-line) та планувальник періоду виконання (on-line, run-time).

Розглянемо планувальник періоду розробки. На етапі розробки СРЧ у розпорядження розробника має бути частина інформації про прикладні задачі та особливості їх взаємодії. Планування періоду розробки полягає в обробці інформації до початку роботи системи. Планувальник періоду виконання є складовою системи і починає діяти при її запуску. Він використовує інформацію, яка отримана у результаті періоду розробки.

Планування періоду розробки включає аналіз можливості виконання (schedulability analysis, feasibility analysis), необхідність в якому виникає, коли обмеження на час виконання задач системи чітко сформульовані і порушення цього часу може призвести до виходу системи з ладу.

Успішне завершення аналізу дає гарантію того, що за різних обставин виконання всіх задач буде завершено у заданий час.

При розподіленні робіт по періодам розробки розрізняють два типи планувальників: статичні та динамічні.

Статичне планування використовують тоді, коли ще до початку роботи системи є повна інформація про задачі, що вирішуються: час виконання, структура та часові характеристики взаємодії задач. У цьому випадку, на етапі розробки системи можна побудувати статичний графік, який має повну інформацію на період виконання, і планувальник періоду виконання вироджується у простий інтерпретатор статичного графіку.

Алгоритми планування для простих систем, де питання ефективності не є визначальним, не використовують системи пріоритетів. Однак, у більшості випадків, використовують алгоритми планування, що засновані на пріоритетах задач. Всі готові задачі впорядковуються по значеннях пріоритету і управління завжди отримує задача з найбільшим пріоритетом. Пріоритети задач визначають з урахуванням їх часових характеристик. Пріоритети можуть бути

фіксовані або динамічні. По способу призначення пріоритетів задачам алгоритми планування поділяють на алгоритми планування з фіксованими або динамічними пріоритетами.

У першому випадку, пріоритети задачам призначаються при їх включенні у склад системи при розробці та залишаються незмінними на весь період роботи системи.

У другому випадку, пріоритети задачам визначають заново при кожному випадку їх виникнення. При цьому значення пріоритету може змінюватись у часі.

Алгоритми, що використовують динамічні пріоритети, більш ефективні, але досить складні у реалізації. Алгоритми з фіксованими пріоритетами менш ефективні, але прості в реалізації.

Пріоритетні планувальники поділяються на витисняючі і не витисняючі. Витисняючі - при появі більш пріоритетної задачі миттєво передають їй процесор, а не витисняючі - дозволяють завершити менш пріоритетні задачі, що до того виконувались.

Витисняючі планувальники забезпечують більшу ефективність планування і тому в ОС СРЧ набагато частіше використовуються .

Класичні алгоритми призначення пріоритетів задачам використовують тільки унікальні пріоритети кожній задачі. Однак, у ряді випадків бажаним є призначення ряду задач одного пріоритету. У такому випадку найбільш простим та ефективним методом розв’язання протиріч між задачами одного пріоритету є обслуговування задач одного пріоритету у порядку надходження (FIFO) Тобто, загальний алгоритм планування може бути віднесено до типу MQS (Multilevel Queue Scheduling), а алгоритм кожної черги - складової до типу FIFO. Іноді рівно пріоритетні задачі виконуються також і у циклічному порядку (Round-Robin), а час, що відведений до кожної задачі, лімітується.

8.2 Методи визначення часу виконання програм

Оцінка часу виконання програм на стадії проектування програмного забезпечення спеціалізованих комп'ютерних систем завжди викликала значний інтерес у розробників.

Вирішення поставленої задачі можна розбити на декілька етапів:

- дослідження та обґрунтування методів визначення часу виконання окремої програми у монопольному режимі;
- дослідження та обґрунтування методів визначення часу виконання програм у СРЧ, що використовує алгоритми планування, засновані на пріоритетах задач.

8.2.1. Прогнозування часу виконання програми

Підходи до прогнозування та визначення часу виконання програми різні. Вважається, що час виконання певної дії програми є деякою константою. Наприклад, це елементарні дії - складові (оператори) мови високого рівня, такі як: «додати», «поділити», «присвоїти».

Знаючи час виконання певної дії, можна обчислити час виконання деякої послідовності дій.

Гіпотеза про постійний час виконання цих дій, однак, не витримує критики. Цей час може змінюватись у залежності від типу адресації, стану процесора на даний час, організації пам'яті процесора тощо.

Іноді підраховують мінімальний та максимально можливий час і дещо покращують рівень прогнозу.

Такі обчислення називають аналітичними. Подальше покращення аналітичних обчислень можливе, якщо додатково промоделювати роботу різних пристроїв процесора або навіть побудувати емулятор обчислювача. Частіше використовують лише статичну імітацію, яка зводиться до побудови таблиць зайнятості пристроїв процесора при виконанні програми.

Іноді, в інструментальних системах, призначених для оцінки часу виконання програм, процесор та метод відображення програми наперед

визначені. Такі системи, зазвичай, дають достатньо високий рівень якості прогнозу, оскільки дають можливість наперед врахувати як властивості мови, так і процесора. Ці системи називають спеціалізованими.

У подальшому будуть розглядатися тільки спеціалізовані системи, у яких програми представлені мовою високого рівня, а тип процесора наперед обраний. Це дає можливість позбутися неприємних процедур емуляції або врахування багатьох особливостей архітектури процесора.

Часто використовують термін «лінійний (базовий) блок програми». Надамо йому формальне визначення: лінійним блоком (ЛБ) програми будемо вважати фрагмент, що створений послідовними діями програми, якому притаманні наступні властивості:

- виконання фрагменту починається з його першої дії;
- виконуються послідовно усі зазначені дії до останньої.

Лінійна структура всієї програми - досить рідкісний випадок. Як правило, вона складається, як з лінійних (базових) блоків, так і з циклів, умовних, безумовних переходів та викликів процедур і функцій. Така структура програми визначає досить складну структуру графа передачі керування між базовими блоками. Цей граф фактично визначає множину всіх шляхів (історій виконання) від початку до завершення програм. Кожна з таких історій відповідає певним даним, тобто залежить від них.

Поводження програми - це, власне, реалізація сукупності всіх можливих історій виконання.

Відповідно до мети прогнозу оцінки часу виконання програми її можна виконати :

- для кожної з можливих історій з осередненням результатів;
- для деякої сукупності історій;
- для конкретно заданої історії.

Вибір мети прогнозу тісно пов’язано з пошуком репрезентативних наборів даних, які б відповідали цьому.

Це пов’язано зі статичним і статистичним аналізом вхідних даних, що часто досить ускладнює задачу. Оцінка часу виконання програми для кожної з можливих історій з осередненням результатів необхідно виконати у випадку, коли цікавить прогноз повного часу виконання програми у багатопрограмній системі, що використовує пріоритети задач.

У цьому випадку можна піти двома шляхами:

- виконати вимірювання часу виконання у монопольному режимі для всіх можливих вхідних наборів даних і осереднити результат;
- оцінити час виконання окремих базових блоків програми та визначити матрицю ймовірностей переходу між окремими блоками P, розглядаючи процес виконання програми, як випадковий марківский процес з дискретними станами та дискретним часом .

Дещо інший метод може бути досить ефективно використаний у випадку, коли програма реалізується за допомогою бібліотеки програмних модулів, часові характеристики як их наперед відомі .

Задача **прогнозування** часу виконання окремої програми буде пов’язана зі здатністю вирішення трьох задач:

- виміру часу виконання прикладної програми;
- виміру часу виконання заданого фрагменту прикладної програми;
- прогнозування часу виконання програми на моделі випадкового марківського процесу з дискретними станами та дискретним часом.

Вхідні дані та текст програми задано мовою високого рівня.

8.2.2. Методика виміру часу виконання прикладної програми

Відомо, що на час виконання програми впливають різни фактори :

- особливості самої програми;
- архітектура та конфігурація комп’ютера;
- операційна система;
- сумісно працюючі процеси;
- стан комп’ютера на момент старту програми;

- вплив вимірювача.

Максимально зменшити вплив факторів, які заважають, не завжди просто. Однак існує цілий ряд прийомів, які дозволяють це зробити:

- слід обрати той комп'ютер, на якому у подальшому передбачається реалізація системи;
- програму слід запускати багаторазово у циклі, та поділити загально отриманий час на число ітерацій у циклі або обрати мінімальний час виконання програми, бо він буде найбільш точним;
- виключити з стадії виміру стадії ініціалізації та завершення фрагменту коду;
- зменшити вплив коду виміру часу, для чого код виміру часу має викликатись тільки на початку та по завершенні фрагменту коду;
- виконати прив'язку поточного потоку до одного з ядер процесора (якщо він багатоядерний), для цього встановити для поточного процесу найвищий пріоритет;
- для виміру часу доцільно використовувати функції, що отримують поточне значення високочастотного лічильника та перетворюють його в одиниці часу (кількість тиків у секунду).

Розглянемо час виміру прикладної програми в операційній системі Windows. Фірмою Microsoft, щоб раз і назавжди поставити крапку у проблемі виміру часу виконання програм, було введено таймер QueryPerformanceCounter( ). Для цього використаємо функцію QueryPerformanceCounter( ), яка отримує поточне значення високочастотного лічильника. За допомогою функції QueryPerformanceFrequency( ) переведемо значення лічильника в одиниці часу. Ця функція повертає кількість тиків лічильника у секунди. Для більшої релевантності замірів виконаємо прив'язку поточного потоку до одного з ядер процесора. Встановимо для поточного процесу найвищій пріоритет .

Як приклад, у якості прикладної програми використаємо реалізацію алгоритму сортування Шелла. Лістинг програми мовою С наведено на рис. 8.1.

```
double tmp;
int d = SIZE / 2;
bool b1_cond = d > 0;
while (b1_cond) {
	int i = d;
	bool b2_cond = i < SIZE;
	while (b2_cond) {
		int j = i;
		bool b3_cond = j >= d && a[j - d] > a[j];
		while(b3_cond){
			tmp = a[j];
			a[j] = a[j - d];
			a[j - d] = tmp;
			j -= d;
			b3_cond = j >= d && a[j - d] > a[j];
		}
		i++;
		b2_cond = i < SIZE;
	}
	d /= 2;
	b1_cond = d > 0;
}
```

Рис. 8.1. Лістинг програми алгоритму Шелла мовою С

Всю логіку замірів часу винесемо в окремий клас *TimeMeasure*. При конструюванні класу розглянемо параметри, які передбачають не тільки вимір часу задачі, як певної одиниці, але зможуть бути застосовані і при вимірі певної кількості лінійних фрагментів - складових прикладної задачі: *TimeMeasure tm*(n, m, *TimeModifier*), де

n - кількість фрагментів;

m - максимальна кількість замірів для кожного фрагменту, якщо він зустрічається декілька разів при вирішенні задачі виміру згідно алгоритму;

TimeModifier - дільник частоти, який використовується для зміни одиниці виміру (1 - секунди, 1000 - мсек, тощо).

Для заміру часу у класі *TimeMeasure* передбачено дві функції *Start* та *Save*. Функція *Start*( ) розпочинає відлік часу виконання. Її виклик треба вставляти на початку фрагменту, що досліджується (у тому числі і задачі). Функція *Save*( ) фіксує час виконання заданого фрагменту. Ця функція приймає значення номеру фрагменту.

Створений клас *TimeMeasure* доцільно розмістити у окремому файлі «*time _ measure. h.*», який можна буде підключати до програми, що досліджується. Для такого підключення цей файл слід скопіювати у ту саму директорію (папку), де знаходиться файл з досліджуваною програмою. Після цього, в програмі слід використати директиву *#include"time _ measure. h* .

На рис. 8.2 наведено лістинг файлу *time _ measure. h*.

```
#ifndef _TIME_MEASURE_H_
#define _TIME_MEASURE_H_

#include <Windows.h>
#include <iostream>
#include <ctime>
#include <vector>
#include <limits>

class TimeMeasure {
    std::vector<std::vector<double> > ticks;
    __int64 freq;
    __int64 tmp_start;
    int max_size;

public:
    //n - number of blocks
    //m - number of measures for every block
    TimeMeasure(int n, int m, int time_modifier) : max_size(m) {
     for (int i = 0; i < n; ++i) {
          std::vector<double> tmp_vec;
          ticks.push_back(tmp_vec);
     }

     QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
     freq /= time_modifier;

     SetThreadAffinityMask(GetCurrentThread(), 1);
     SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);
    }

    ~TimeMeasure() {
     int block_num = 0;
     for     (std::vector<std::vector<double>     >::iterator     it     =
ticks.begin(); it != ticks.end(); it++) {
          std::cout << "Block #" << block_num << std::endl;
    double avg = 0;
          for  (std::vector<double>::iterator  inner_it  =  it->begin();
inner_it != it->end(); inner_it++) {
               std::cout << *inner_it << "  ";
               avg += *inner_it;
          }
```

```
            std::cout << std::endl << "Average time: " << avg / it->size()
<< std::endl << std::endl;
            ++block_num;
      }
     }

     void Start() {
      QueryPerformanceCounter((LARGE_INTEGER *)&tmp_start);
     }

     void Save(int n) {
      if (ticks[n].size() >= max_size) return;

      __int64 end;
      QueryPerformanceCounter((LARGE_INTEGER *)&end);
      double new_time = (double)(end - tmp_start) / freq;
      ticks[n].push_back(new_time);
     }
};
#endif
```

Рис. 8.2. Лістинг файлу *time _ measure. h*.

Таким чином, обрано метод та створено універсальний засіб виміру часу виконання прикладної програми або її окремих лінійних фрагментів (базових блоків). Як приклад, лістинг програми алгоритму Шелла мовою С після додавання класу *TimeMeasure* та використання його методів для заміру часу наведено на рис. 8.3.

Виклики функцій *Start* ( ) і *Save* ( ) вставлені в тіло програми, як маркери, які відокремлюють оператори, сумарний час використання яких досліджується, а для передачі параметрів об'єкту класу *TimeMeasure* використовується *TimeMeasure tm* (*n*, *m*, *TimeModifier*).

```
#include <iostream>
#include <cstdlib>

#include "time_measure.h"

#define SIZE 1000

void shell_sort(double* a)
{
    TimeMeasure tm(1, 1, 1000);
    /*block 0*/
    tm.Start();
    double tmp;
    int d = SIZE / 2;
    bool b1_cond = d > 0;
```

```
     /*block 0 end*/
     while (b1_cond) {

      /*block 1*/

      int i = d;
      bool b2_cond = i < SIZE;

      /*block 1 end*/
      while (b2_cond) {

            /*block 2*/

int j = i;
            bool b3_cond = j >= d && a[j - d] > a[j];

            /*block 2 end*/
            while(b3_cond){
                  /*block 3*/
                  tmp = a[j];
                  a[j] = a[j - d];
                  a[j - d] = tmp;
                  j -= d;
                  b3_cond = j >= d && a[j - d] > a[j];

                  /*block 3 end*/
            }
            /*block 4*/

            i++;
            b2_cond = i < SIZE;

            /*block 4 end*/
       }
       /*block 5*/

       d /= 2;
       b1_cond = d > 0;

       /*block 5 end*/
     }
     tm.Save(0);
}
void fill_random(double* a)
{
     srand(time(NULL));
     for (int i = 0; i < SIZE; ++i) {
       a[i] = (double)rand() / RAND_MAX;
     }
}
int main()
{
     double a[SIZE];
     double a1[SIZE];
     fill_random(a);
     for(int i = 0; i < SIZE; i++)
       a1[i] = a[i];
     //i раз запуск shell_sort
```

```
    for(int i = 0; i < 1; i++){
     shell_sort(a);
     std::cout<<"------------------------------\n";
     for(int i = 0; i < SIZE; i++)
          a[i] = a1[i];
    }
    std::cout << "Press any key..." << std::endl;
    getchar();

}
```

Рис. 8.3. Лістинг програми алгоритму Шелла після додавання класу *TimeMeasure*

Результат роботи програми, що представлена на рис. 8.3, наведено на рис. 8.4 (а, б). На рис. 8.4 представлені результати: а) у мсек: *TimeMeasure tm* (1, 1, 1000); б) у мксек: *TimeMeasure tm* (1, 1, 1000000).

Block #0

0.217047

Average time: 0.217047

Press any key...

а)

Block #0

219.19

Average time: 219.19

Press any key...

б)

Рис. 8.4. Результат роботи програми у мсек (а) та результат роботи програми у мксек (б)

8.2.3. Методика виміру часу лінійних блоків прикладної програми

З метою виділення лінійних (базових) блоків прикладної програми розглянемо схему програми, що відповідає лістингу, представленому на рис. 8.1. Ця схема буде мати вигляд, як на рис. 8.5.

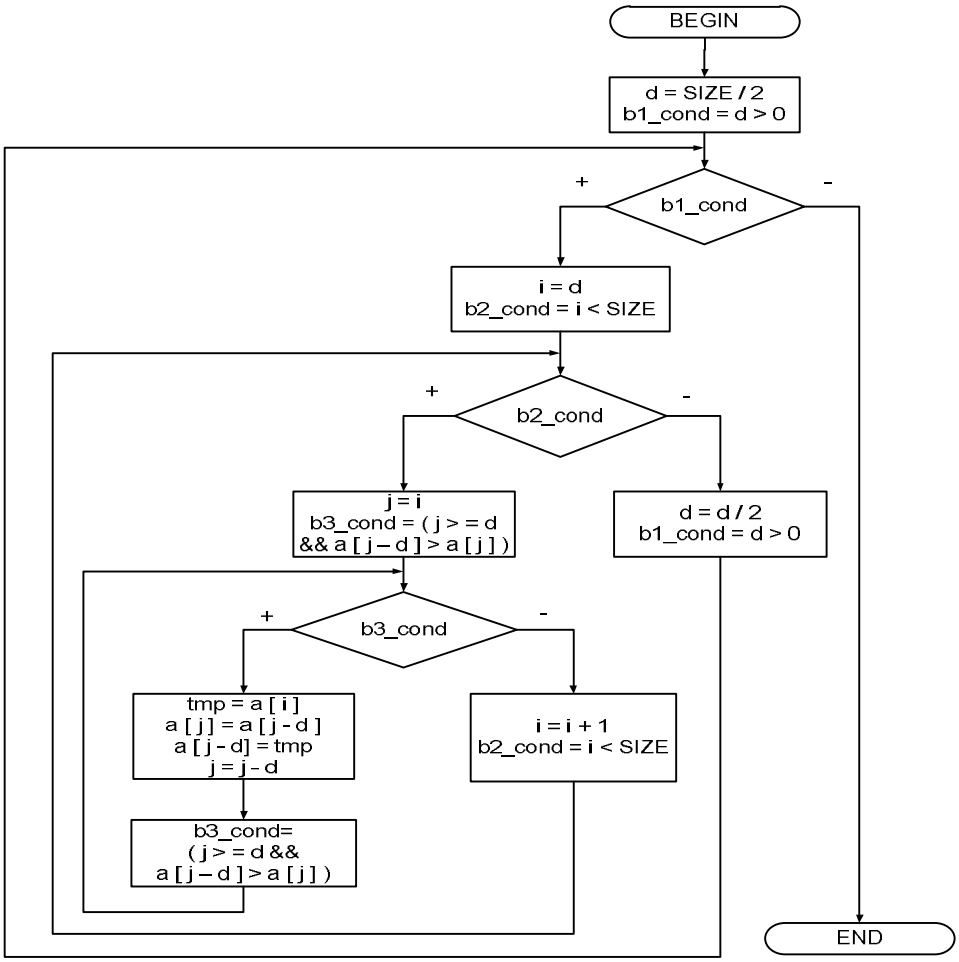


Рис. 8.5. Схема алгоритму програми

За отриманою схемою створимо граф переходів програми, замінивши окремі блоки програми на вершини графу. Лінійні блоки, час виконання яких буде у подальшому виміряно, пронумеровано цифрами, починаючи з нуля, а вершини графу, що відповідають переходам, позначено парою - < с, цифра >. Всього на графі налічується 7 лінійних блоків з номерами від 0 до 6 та три умовні переходи C_0, ... , C_2. Граф переходів представлено на рис. 8.6.

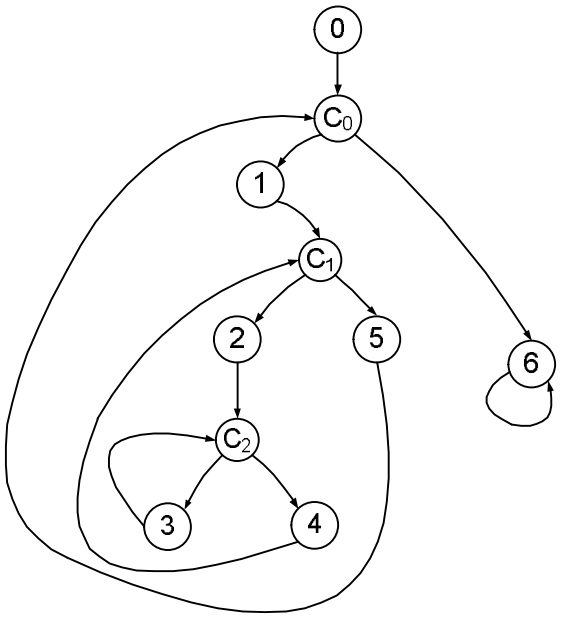


Рис. 8.6. Граф переходів програми

Виклики функцій *Start*( ) і *Save*( ) слугують певними маркерами, які визначають початок та кінець блоків програми, час яких вимірюється.

Якщо під час роботи програми деякі лінійні блоки повторюються декілька разів (наприклад, N разів), то результати виміру друкуються n разів при $N \geq m$, або N разів при $m > N$ та вираховується середньоарифметичне за вимірюваннями часу виконання блоку.

Передача параметрів об'єкту класу *TimeModifier* виконується як і у попередньому випадку.

Лістинг програми для виміру часу виконання програми окремих блоків після додавання класу *TimeMeasure* та використання його методів для заміру часу наведено на рис. 8.7.

```
#include <iostream>
#include <cstdlib>

#include "time_measure.h"

#define SIZE 10000
void shell_sort(double* a)
{
      TimeMeasure tm(6, 10, 1000);

      /*block 0*/
```

```
        tm.Start();
        double tmp;
        int d = SIZE / 2;
        bool b1_cond = d > 0;
        tm.Save(0);
        /*block 0 end*/
        while (b1_cond) {

          /*block 1*/
          tm.Start();
          int i = d;
          bool b2_cond = i < SIZE;
          tm.Save(1);
          /*block 1 end*/
          while (b2_cond) {
        /*block 2*/
                tm.Start();
                int j = i;
                bool b3_cond = j >= d && a[j - d] > a[j];
                tm.Save(2);
                /*block 2 end*/
                while(b3_cond){

                        /*block 3*/
                        tm.Start();
                        tmp = a[j];
                        a[j] = a[j - d];
                        a[j - d] = tmp;
                        j -= d;
                        b3_cond = j >= d && a[j - d] > a[j];
                        tm.Save(3);
                        /*block 3 end*/
                }
                /*block 4*/
                tm.Start();
                i++;
                b2_cond = i < SIZE;
                tm.Save(4);
                /*block 4 end*/
        }
/*block 5*/
        tm.Start();
        d /= 2;
        b1_cond = d > 0;
        tm.Save(5);
        /*block 5 end*/
        }
}
```

```
/*block 5*/
          tm.Start();
          d /= 2;
          b1_cond = d > 0;
          tm.Save(5);
          /*block 5 end*/
        }
}
void fill_random(double* a)
{
        srand(time(NULL));
        for (int i = 0; i < SIZE; ++i) {
          a[i] = (double)rand() / RAND_MAX;
        }
}

int main()
{
        double a[SIZE];
        double a1[SIZE];
        fill_random(a);
        for(int i = 0; i < SIZE; i++)
          a1[i] = a[i];

        //i раз запуск shell_sort
        for(int i = 0; i < 1; i++){
          shell_sort(a);
          std::cout<<"------------------------------\n";
          for(int i = 0; i < SIZE; i++)
                  a[i] = a1[i];
        }
        std::cout << "Press any key..." << std::endl;
        getchar();

}
```

Рис. 8.7. Лістинг програми для виміру часу виконання програми окремих блоків після додавання класу *TimeMeasure*

Результати роботи програми наведено на рис. 8.8, де результати виміру часу виконання блоків розраховано в мілісекундах (а), тобто *TimeMeasure tm* (6, 10, 1000), та в мікросекундах (б), тобто *TimeMeasure tm* (6, 10, 1000000).

Block #0

0.000514992

Average time: 0.000514992

Block #1

0.000304995 0.000299995 0.000299995 0.000304995 0.000299995 0.000314995 0.000299995 0.000299995 0.000309995 0.000299995

Average time: 0.000303495

Block #2

0.000319995 0.000314995 0.000314995 0.000304995 0.000309995 0.000309995 0.000304995 0.000354995 0.000309995 0.000294995

Block #3

0.000329995 0.000294995 0.000299995 0.000294995 0.000334995 0.000304995 0.000299995 0.000299995 0.000299995 0.000304995

Average time: 0.000306495

Block #4

0.000304995 0.000299995 0.000304995 0.000314995 0.000299995 0.000344995 0.000304995 0.000294995 0.000299995 0.000304995

Average time: 0.000307495

Block #5

0.000304995 0.000299995 0.000299995 0.000299995 0.000314995 0.000299995 0.000299995 0.000304995 0.000309995 0.000304995

Average time: 0.000303995

Press any key...

a)

Block #0

0.535

Average time: 0.535

Block #1

0.3 0.305 0.3 0.335 0.305 0.32 0.3 0.295 0.295 0.295

Average time: 0.305

Block #2

0.315 0.315 0.31 0.3 0.36 0.305 0.3 0.305 0.31 0.305

Average time: 0.3125

Block #3

0.325 0.315 0.295 0.305 0.335 0.295 0.295 0.295 0.3 0.295

Average time: 0.3055

Block #4

0.3 0.295 0.305 0.345 0.305 0.295 0.3 0.29 0.29 0.29

Average time: 0.3015

Block #5

0.3 0.3 0.3 0.3 0.3 0.3 0.295 0.295 0.295 0.3

Average time: 0.2985

Press any key...

б)

Рис. 8.8. Результати роботи програми в мілісекундах (а) та в мікросекундах (б)

8.2.4. Метод прогнозування часу виконання програм за допомогою марківського ланцюга

У цьому розділі розглянуто моделювання процесу виконання програми за допомогою моделі марківського ланцюга з дискретними станами та дискретним часом. Відповідний марківський процес представлено у вигляді графу переходу, де дискретним станам у відповідність були поставлені вершини графу, а переходам зі стану в стан - орієнтовані ребра, на яких позначались імовірності можливих переходів зі стану в стан. На кожному графі виділялись дві особливі вершини: вершина, що відповідала початковому стану процесу та вершина поглинаючого стану, в якій випадковий процес має завершитись.

Ототодження програми випадковому процесу виконано за рахунок того, що кожному стану випадкового процесу ставився у відповідність лінійний блок програми, а переходам - умовні та безумовні переходи програми. Моменти часу, в які процес стрибком переходить зі стану в стан, визначено часом знаходження у попередньому стані, якому відповідав у свою чергу час виконання того чи іншого лінійного блоку.

За допомогою моделі програми у вигляді поглинаючого марківського ланцюга можна обчислити кількість звертань до окремих блоків програми, середній час виконання програми, середню кількість кроків до переходу у поглинаючий стан, що свідчить про завершення програми.

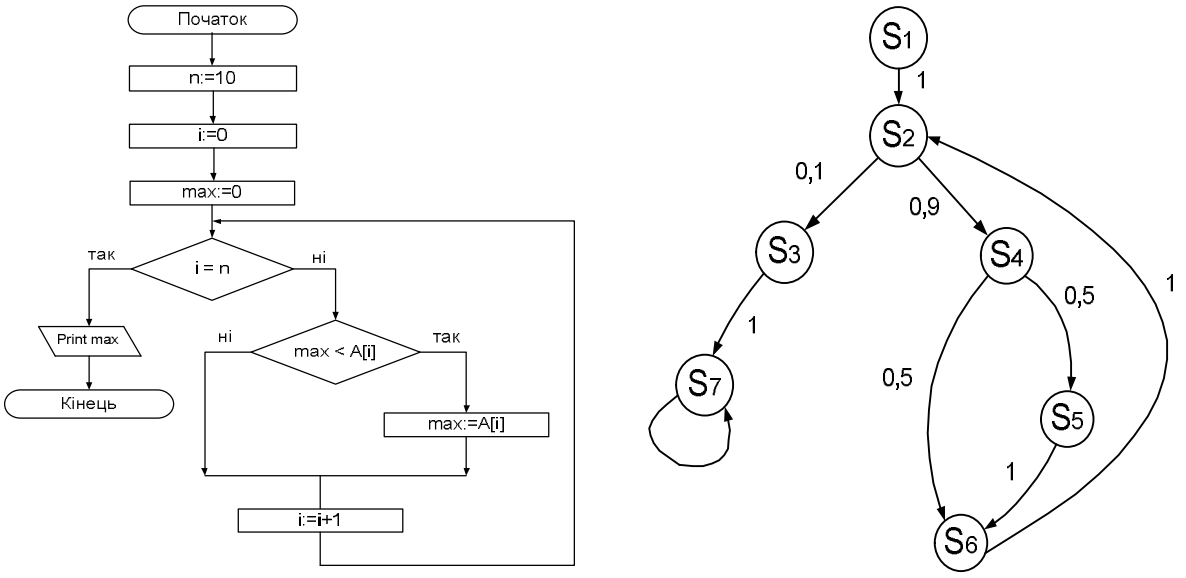
Для визначення вказаних характеристик різними авторами застосовано методи, які базуються на побудові фундаментальної матриці та імітаційного моделювання марківського ланцюга. Однак, у загальному випадку, коли кількість станів марківського ланцюга перевищує кілька десятків, ці методи важко застосовувати через труднощі обчислювання.

Далі розглянемо метод, що базується на математичному моделюванні поглинаючого марківського ланцюга, який вільний від вказаних недоліків .

Опис математичної моделі. Поставимо у відповідність схемі алгоритму програми орієнтований граф $G(i)$, у якому вершини i будуть співставленні блокам і розгалуженням, а орієнтованим ребрам - шляхи переходу алгоритму між окремими блоками. Припустимо, що процесу виконання програми будуть відповідати переходи з однієї вершини до іншої, а знаходженню у певній вершині - виконанню деякої команди або блоку програми.

Як приклад, на рис. 8.9 наведено схему алгоритму програми пошуку максимального елементу в одновимірному масиві (а) та відповідний граф $G(i)$ (б).

$G(i)$ (б).



а) б)

Рис. 8.9. Схема алгоритму програми (а) та її представлення графом (б)

Граф $G(i)$ може бути побудований для алгоритмів будь-якого рівня: рівень команд, модулів, процедур, операторів мови проектування програм PDL. Використовуючи таку модель, розглянуто виконання програми як існування деякого випадкового процесу $S(t)$ з дискретними станами і з дискретним

часом, якому відповідає граф $G(i)$. Зважаючи на особливості виконання програм, переходи процесу $S(t)$ зі стану i в стан j не залежать від того, яким чином він потрапив у стан i, а тільки залежить від перехідних ймовірностей P_{ij}, якими навантажують (розмічають) відповідні ребра графа $G(i)$.

З урахуванням цього, можна інтерпретувати процес виконання програми, як випадковий марківский процес з дискретними станами, які пов'язані у марківський ланцюг матрицею перехідних ймовірностей P.

Нагадаємо, що випадковий процес $S(t)$, який відбувається у системі, називається процесом з дискретним часом, якщо перехід з одного стану в інший відбувається тільки у наперед визначені моменти часу $t(1), t(2), \ldots$, що називаються кроками такого процесу. У проміжках між кроками система S зберігає свій стан. При цьому не виключається, що на деяких кроках система не буде змінювати свій стан. Як відомо , випадковий процес з дискретним часом можна представити випадковою послідовністю (по індексу k) таких подій

$$S_i(k);\ \overline{i=1,\ n};\ \overline{k=1,\ 2,\ \ldots}, \tag{8.1}$$

де n - кількість вершин графа $G(i)$.

Оскільки система $S(t)$ у довільний момент часу t може перебувати тільки в одному зі станів $S_1, \ldots, S_n$, то при кожному $k=1,\ 2,\ \ldots$ події $S_1(k), \ldots, S_n(k)$ несумісні і утворюють повну групу подій. Основними характеристиками марківських ланцюгів з дискретним часом є імовірність станів

$P_i(k) = P(S_i(k))$, $\overline{i=1,\ n};\ \overline{k=1,\ 2,\ \ldots}$

та імовірності переходів P_{ij}, що утворюють квадратну матрицю переходів $\mathbf{P}$ порядку n.

$$\mathbf{P} = \begin{bmatrix} P_{11},\ P_{12},\ \ldots,\ P_{1n} \\ P_{21},\ P_{22},\ \ldots,\ P_{2n} \\ P_{n1},\ P_{n2},\ \ldots,\ P_{nn} \end{bmatrix} \tag{8.2}$$

$$\sum_{i=1}^{n} P_i(k) = 1; \quad k = 1,\ 2,\ \ldots, \tag{8.3}$$

$$\sum_{j=1}^{n} P_{ij} = 1; \quad \overline{i = 1,\ n}. \tag{8.4}$$

Наявність на розміченому графі стрілок та визначень відповідних ймовірностей переходів вказує на відмінність їх від нуля. Відсутність стрілок говорить про те, що відповідні імовірності дорівнюють нулю. Якщо сума ймовірностей, що виходять з певної вершини менше 1, це свідчить про імовірність затримки на певному кроці P_{ii}, яка дорівнює

$$P_{ii} = 1 - \sum_{j=1}^{n} P_{ij}; \quad i \neq j \tag{8.5}$$

Якщо $P_{ii} = 1$, то процес потрапив в так званий поглинаючий стан, з якого він вже вийти не може, блукання по станам завершується.

Вектор-рядок станів у початковий момент дискретного часу $t(0) - P(0) = (P_1(0),\ P_2(0),\ \ldots,\ P_n(0))$ називають вектором початкового розподілення ймовірностей. Для наведеного прикладу (рис. 8.9(б)) він дорівнює $(1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0)$. Як відомо, для визначення вектора - рядка ймовірностей переходу від кроку k до кроку $k+1$ слід взяти добуток вектора-рядка переходу від стану $k-1$ до k на матрицю перехідних ймовірностей $\mathbf{P}$, тобто

$$P(k) = P(\mathrm{k}-1)\mathbf{P}, \tag{8.6}$$

у загальному вигляді

$$P(k) = P(0)\mathbf{P}^k.$$

Або,

$$P_i(k) = \sum_{j=1}^{n} P_j(k-1)P_{ji}; \quad \overline{i = 1,\ n}. \tag{8.7}$$

Тривалість часу між моментами переходу іноді розглядається як стала величина, тобто $t(i) - t(i-1) = const$. Пов'яжемо тривалість між моментами переходу з попереднім станом процесу. Якщо процес переходить зі стану S_i в

стан S_j, то такий перехід можливий після завершення проміжку часу τ_i - знаходження процесу в стані S_i. У такому разі розглядаємо випадковий процес з дискретним часом, переходи якого відбуваються через наперед визначені, але нерівномірні інтервали часу.

Визначимо величину проміжку часу $T(k)$ між сусідніми кроками. Ця величина буде дорівнювати

$$T(k) = t(k) - t(k-1) \tag{8.8}$$

або $T(k)$ буде відповідати формулі

$$T(k) = \sum_{i=1}^{n} P_i(k-1)\tau_i \tag{8.9}$$

при нерівномірній дискретизації та $T(k) = \tau_i = const$ при рівномірній при нерівномірній дискретизації та $T(k) = \tau_i = const$ при рівномірній дискретизації. Якщо $\tau_i = 1,$ то

$$T(k) = \sum_{i=1}^{n} P_i(k-1) = 1, \tag{8.10}$$

Якщо кроки нерівномірні за часом, то величина $T(k)$ визначається за формулою (8.9) і буде залежатиме від τ_i.

Кожний алгоритм або програма мають початок і кінець. Таким чином, ця необхідна умова відповідає структурі графа $G(i)$. Як відомо , представляючи алгоритм або програму у вигляді графу $G(i)$, можна наперед побудувати діаграму порядку G_1, оскільки вона представляє собою граф без контурів. При цьому, у графі G_1 можуть існувати тупикові вершини - вершини, з яких дуги не виходять за межі контуру. Підграфи, що відповідають тупиковим вершинам, називають ергодичними класами марківського ланцюга. Якщо ергодичний клас складається з однієї вершини, то такий стан називають поглинаючим. Наприклад, на рис. 8.10 наведено граф $G(i)$ (а) та його діаграма порядку (б).

З діаграми порядку видно, що у графі $G(i)$ існує два ергодичні класи:

$A_1 = \{3,\ 4\}$ *та* $A_2 = \{8\}$.

Клас A_2 є поглинаючим.

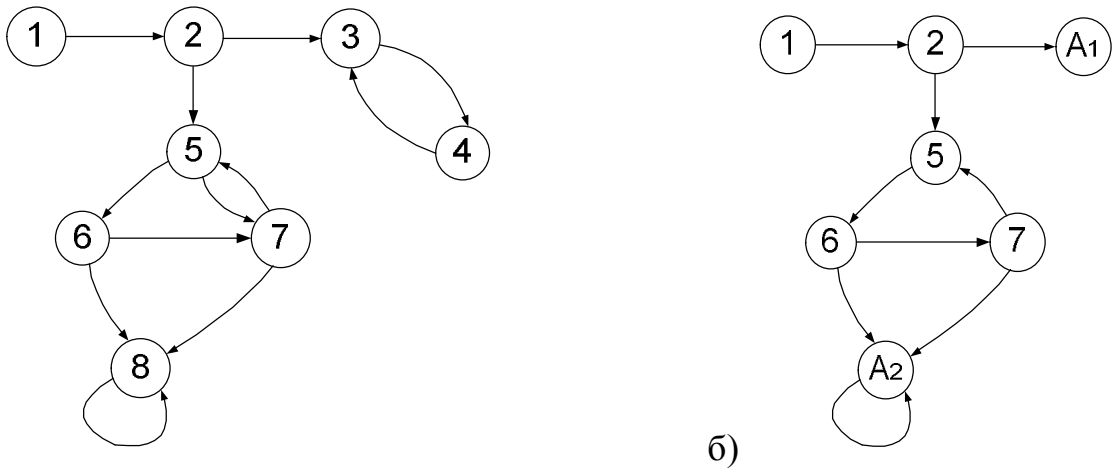


Рис. 8.10. Граф переходів $G(i)$ (а) та його діаграма порядку (б)

Під час блукання по графу з імовірністю 1, траєкторія марківського ланцюга опиниться в одному з ергодичних класів і вийти з нього не зможе. Якщо наведений граф $G(i)$ відповідає деякий конкретній програмі, то можна казати, що початку програми відповідає вершина 1, кінцю - вершина 8, ергодичний клас A_1 відповідає помилці - «зациклюванню» програми, яка має бути виправлена, а A_2 - поглинаючий (кінець програми).

Метод використання моделі програми. Використаємо метод обчислень, побудований безпосередньо на математичній моделі марківського ланцюга з дискретними станами та дискретним часом, який полягає у послідовному обчисленні граничних ймовірностей станів

$$P_i(k) \; при \;\; k \to \infty \, .$$

Критерієм завершення процедури обчислень служить:

$$P_n(k) \to 1, \; при \;\; k \to \infty \, , \qquad (8.11)$$

якщо P_n відповідає поглинаючому стану.

Поняття $k \to \infty$ є математичною абстракцією. Доведено, що для відповідних розрахунків достатньо обрати $P_n \; (k) \approx 0,99999$.

Позначимо середню кількість кроків до поглинання через K. Тоді, середній час виконання K кроків до завершення програми (потрапляння у поглинаючий стан) T можна обчислити, як:

$$T = \sum_{j=1}^{K} T(j). \tag{8.12}$$

Якщо поглинаючий стан винесено за «кінець» алгоритму, то час його виконання τ_n в розрахунках має дорівнювати 0. Тоді, з урахуванням (8.9) час T визначається, як:

$$T = \sum_{j=1}^{K} \sum_{i=1}^{n} P_i(j-1)\tau_i. \tag{8.13}$$

Припустимо, що $\tau_i = const = 1; \quad \overline{i = 1,\ n-1}; \quad \tau_n = 0$, тоді з урахуванням (8.9)

$$T = \sum_{j=1}^{K} T(j) = K. \tag{8.14}$$

Визначимо загальний час перебування системи у стані $S_i - T_i$.

На кожному кроці система знаходиться у стані S_i середній час $P_i(j-1)\ \tau_i$.

Тоді, за K кроків загальний час T_i визначається як:

$$T_i = \sum_{j=1}^{K} P_i(j-1)\tau_i \tag{8.15}$$

У середньому, кількість разів K_i потрапляння системи у стан S_i буде становити:

$$K_i = T_i \ / \ \tau_i \tag{8.16}$$

або

$$K_i = \sum_{j=1}^{K} P_i(j-1). \tag{8.17}$$

Тоді,

$$K = \sum_{i=1}^{n} K_i. \tag{8.18}$$

За результатами досліджень створена спеціальна програма обчислення вказаних характеристик. Наведемо приклад розрахунку параметрів T, K та

K_i програми, граф $G(i)$ якої наведено на рис. 8.9 (б). Матриця перехідних ймовірностей P_{ij} наведена на рис. 8.11.

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	0.1	0.9	0	0	0
3	0	0	0	0	0	0	1
4	0	0	0	0	0.5	0.5	0
5	0	0	0	0	0	1	0
6	0	1	0	0	0	0	0
7	0	0	0	0	0	0	1

Рис. 8.11. Матриця перехідних ймовірностей P_{ij}

Результати розрахунків для випадку $\tau_1=10$, $\tau_2=20$, $\tau_3=50$, $\tau_4=20$, $\tau_5=10$, $\tau_6=20$:

$T=665$; $K=34{,}5$; $K_1=1$; $K_2=10$; $K_3=1$; $K_4=9$; $K_5=4{,}5$; $K_6=9$.

Таким чином, запропоновано метод обчислення часових характеристик виконання програм. Перевагою цього методу є те, що основною операцією обчислень є операція добутку вектора на матрицю $n \times n$, яка достатньо просто реалізується для великих n, особливо зважаючи на те, що відповідні матриці сильно розріджені і для обчислень можна застосовувати спеціальні методи, рекомендовані сучасною літературою.

Для перевірки правильності методу оберемо програму, час виконання якої не залежить від вхідних даних, а тільки від параметрів задачі. Тоді час її виконання у цілях контролю можна визначити простим підрахунком кількості елементарних операцій, що має виконати програма, а потім зіставити їх з результатами, отриманими за запропонованим методом.

В якості такої програми для зіставлення результатів була обрана програма множення вектору-строки на матрицю. Схема алгоритму програми наведена на рис. 8.12. (Час виконання операції присвоювання дорівнює 3 умовним одиницям додавання - 8 у.о., множення - 15 у.о., умовного переходу - 0). Час виконання програми в умовних одиницях не залежить від конкретних даних, а

тільки від параметрів задачі - розміру матриці $N \times K$. На рис. 8.13 наведено граф переходів процесу, який може бути зіставлений алгоритму програми на рис. 8.12. Шляхом нескладного підрахунку можна встановити, що час знаходження у стані S_1 дорівнює 3 у.о., у стані S_2 - 6 у.о., у стані S_3 - 37 у.о., у стані S_4 - 11 у.о., у стані S_5 - 0. Нижче, у таблиці 8.1 наведені результати обчислення часу виконання програми за двома методами: прямого підрахунку часу операцій та згідно запропонованому методу.

Таблиця 8.1. Результати розрахунків

Розмір матриці $N \times K$	Час виконання за методом простого підрахунку	Час виконання за методом марківського ланцюга
10x10	3873	3872,96
20x20	15143	15142
30x30	33813	33811,32
20x1	1083	1082,99
10x1	543	543
1x1	57	57

Зробимо висновок: дані результатів практично співпадають.

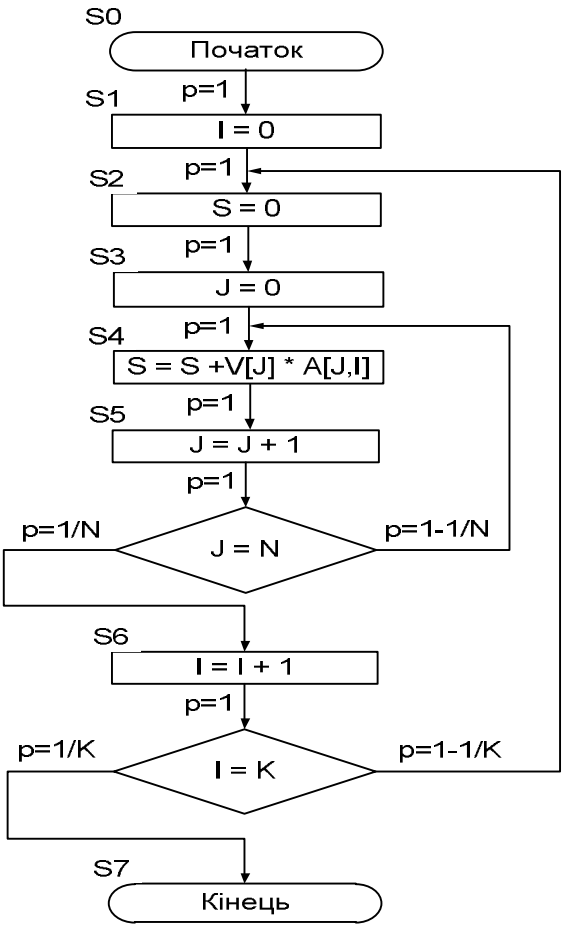


Рис. 8.12. Схема алгоритму програми

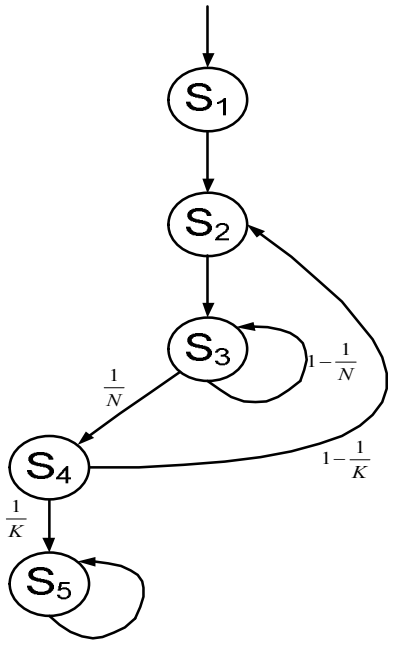


Рис. 8.13. Граф переходів процесу

8.3. Методи дослідження часових характеристик виконання комплексу програм

Встановлено, що програмній системі, яка не заснована на пріоритетах задач може бути поставлена у відповідність скінченна класична модель СМО «про простій верстатів», що названа базовою моделлю дослідження.

Граф випадкового процесу, що відповідає базовій моделі СМО, можна представити, як на рис. 8.14.

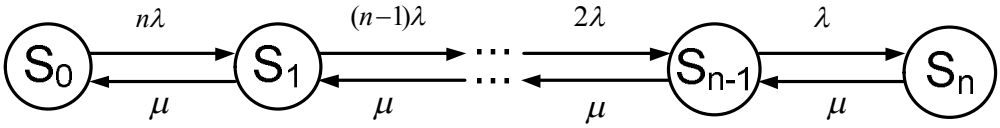


Рис. 8.14. Граф випадкового процесу S

Стани випадкового процесу пронумеровані від 0 до n, де n - кількість задач у системі. Номер стану відповідає кількості задач, що на даний час знаходиться у системі. Вважається, що обслуговування вимог виконується у порядку їх надходження.

Починаючи зі стану S_2 у системі виникає черга на обслуговування.

Ця модель СМО, за умови, що вхідні потоки вимог пуассонівські, а час обслуговування показниковий, добре вивчена у сучасній літературі.

Базова модель СМО відповідає програмній системі, що складається з n задач. Кожна i-та задача у довільний момент часу може виставити вимогу на обслуговування, при чому середня інтенсивність такого потоку вимог становить λ_i. Якщо процесор вільний, він надається задачі. На обслуговування задачі витрачається середній час $\nu_i = 1/\mu_i$. Якщо на момент виникнення вимоги процесор зайнятий, задача стає у чергу на виконання, на що витрачається додатковий час. Черговий запит обслуговування i-та задача може виставити тільки після обслуговування попереднього запиту. Загальний потік від джерел (задач) можна розглядати як суперпозицію або об’єднання вимог. Не важко

показати, що у випадку, коли всі потоки простіші, середня інтенсивність такого потоку буде дорівнювати сумарному потоку з інтенсивністю λ,

де $$\lambda = \sum_{i=1}^{n} \lambda_i \qquad (8.19)$$

Особливістю обраної скінченної моделі СМО є те, що у залежності від стану системи (кількості вимог, що знаходяться у черзі і очікують обслуговування), середня інтенсивність такого потоку буде зменшуватись, оскільки змінюється кількість джерел, що можуть виставити вимоги. Тому слід визначити, яке математичне очікування потоку вимог одного джерела базової моделі СМО λ слід обрати. У якості математичного очікування випадкової величини можна обрати її середньоарифметичне, тобто для пулу задач

$$\lambda = \frac{1}{n} \sum_{i=1}^{n} \lambda_i \qquad (8.20)$$

Відповідно, у якості математичного очікування часу обслуговування задач оберемо середньоарифметичне ν, тобто для пулу задач

$$\nu = \frac{1}{n} \sum_{i=1}^{n} \nu_i, \quad \mu = \frac{1}{\nu} \; . \qquad (8.21)$$

Розглянемо ще одну особливість, пов'язану з базовою моделлю, яка буде корисна при розгляді СРЧ, що засновані на пріоритетах задач.

Дисципліни обслуговування з абсолютним та відносним пріоритетами для такого типу джерел досить детально вивчені Н. Джейсуолом . Ним розглянуто моделі зі скінченними джерелами довільних об’ємів - N_1 та N_2, а також, деякі зауваження для узагальнення процесів з k класами пріоритетів.

Для вивчення згаданих моделей СМО, автором рекомендовано використання досить складних аналітичних викладок і формул, які дуже не просто застосовувати для отримання конкретних практичних результатів. Окрім того, ці дослідження виконані стосовно джерел з 2 класами пріоритетів, а

що стосується k класів, він обмежується, в основному, рекомендаціями до відповідного розширення дослідження.

Тому є доцільним і необхідним спростити дослідження, максимально врахувати раніше отримані результати та розробити алгоритм, що дозволив би оперативно виконувати необхідні обчислення для систем з k пріоритетами.

При вивченні систем з абсолютним пріоритетом отримано наступні результати .

1. У випадку абсолютного пріоритету з двома рівнями пріоритету, виконується обслуговування, так званої, ізольованої пріоритетної черги, тобто обслуговуються тільки пріоритетні вимоги, коли вони є. Таким чином, для вищого пріоритету задача дослідження просто зводиться до розгляду основної моделі зі скінченним джерелом.

2. У випадку k класів пріоритетів з абсолютним пріоритетом розмір черги та час очікування вимог i-го пріоритету не залежить від вимог пріоритетів j при $j > i$.

3. В результаті порівняння циклів обслуговування непріоритетних вимог при різних варіантах абсолютних пріоритетів зроблені наступні висновки: у випадку абсолютного пріоритету з дообслуговуванням, довжина циклу обслуговування повністю визначається тим часом обслуговування, який воно потребує при першому виборі каналом. При цьому, чим більше цей час, тим більшу кількість раз воно буде витіснено пріоритетними вимогами і тим більшим буде цикл обслуговування. У випадку з обслуговуванням заново, даний факт не має місця, оскільки при черговому виборі на обслуговування встановлюється нова реалізація часу обслуговування і цикл обслуговування мов би починається заново, тобто, у середньому цикл обслуговування для абсолютного пріоритету з повтором обслуговування більше, ніж для абсолютного пріоритету з обслуговуванням заново.

4. Оскільки розглядається дисципліна обслуговування з абсолютним пріоритетом, то вимоги класів $j+1, \ldots, k$ $(j \le k)$ ніяк не впливають на інтервал розгрузки каналу для j-го класу. Далі, якщо j класів поділити на дві групи, то

перша група об'єднує від 1 до $(j-1)$, а друга група - складається з вимог j-го класу. Якщо канал обслуговує вимоги другої групи (вимоги j-го класу) і з'являється вимога від першої групи, то обслуговування вимоги другої групи припиняється і не поновлюється, доки у системі не залишиться не обслугованих вимог першої групи. Виходить, що довжина цього часового інтервалу буде дорівнювати періоду зайнятості процесу з $j-1$ класами пріоритетів. Результат для j-го пріоритету можна отримати, якщо розглянути процес з двома класами пріоритетів при заміні вимог 1-го класу на

$$\Lambda_{j-1} = \sum_{i=1}^{j-1} \lambda_i \tag{8.22}$$

Якщо час обслуговування непріоритетної вимоги розподілено експоненційно:

$$(T_p)_{повт} > (T_p)_{заново} = (T_p)_{дообсл}; \tag{8.23}$$

коли час обслуговування сталий:

$$(T_p)_{повт} = (T_p)_{заново} > (T_p)_{дообсл}, \tag{8.24}$$

де $(T_p)_{повт}$ - час виконання непріоритетної вимоги при абсолютному пріоритеті з поновленням обслуговування (поновлюються обслуговування з новим часом обслуговування);

$(T_p)_{заново}$ - час виконання непріоритетної вимоги при абсолютному пріоритеті з обслуговуванням заново (повний старий час обслуговування);

$(T_p)_{дообсл}$ - при абсолютному пріоритеті з дообслуговуванням (поновлюється обслуговування з того стану, коли виникло переривання).

Відносний та змішаний пріоритети досить мало використовуються при створенні ПЗ СРЧ, тому багатопріоритетні системи ПЗ СРЧ будуються з використанням абсолютних пріоритетів.

Іноді вирішують задачу оптимального розподілу задач по пріоритетах. Для цього визначають функціонал вартості очікування для вимог різного класу та

систему обмежень, коли час обслуговування для вимог першого пріоритетного класу становить менше деякої величини ς_1, другого - ς_2 і для k-го класу - ς_k. Таку задачу, наприклад, вирішував Песталоцці методами динамічного програмування.

Виникає питання, яким чином можна визначити параметри λ та μ об'єднаної базової моделі для суми декількох базових моделей, що мають відповідні параметри.

Спочатку розглянемо дві базові моделі, які слід об'єднати в одну. Нехай перша базова модель характеризується трійкою $\{\lambda_1, \nu_1, n_1\}$, друга - $\{\lambda_2, \nu_2, n_2\}$, де λ_i, ν_i, n_i - відповідно параметри λ, ν, n базових моделей. Перенумеруємо задачі - складові базових моделей $\lambda^{(j)}$, $\nu^{(j)}$. Таким чином, що номери від 1 до n_1 будуть належати першій, а від n_{1+1} до $n_1 + n_2$ - другій базовій моделі, тоді отримаємо наступні вирази для середніх значень:

$$\lambda_1 = \frac{1}{n_1}\sum_{j=1}^{n_1}\lambda^{(j)}; \qquad \nu_1 = \frac{1}{n_1}\sum_{j=1}^{n_1}\nu^{(j)}; \tag{8.25}$$

$$\lambda_2 = \frac{1}{n_2}\sum_{j=n_1+1}^{n_1+n_2}\lambda^{(j)}; \quad \nu_2 = \frac{1}{n_2}\sum_{j=n_1+1}^{n_1+n_2}\nu^{(j)}. \tag{8.26}$$

Ліву та праву частини рівняння (8.25) помножимо на n_1, а рівняння (8.26) помножимо на n_2 та отримаємо:

$$n_1\lambda_1 = \sum_{j=1}^{n_1}\lambda^{(j)}; \qquad \mathrm{n}_1\nu_1 = \sum_{j=1}^{n_1}\nu^{(j)}; \tag{8.27}$$

$$n_2\lambda_2 = \sum_{j=n_1+1}^{n_1+n_2}\lambda^{(j)}; \qquad \mathrm{n}_2\nu_2 = \sum_{j=n_1+1}^{n_1+n_2}\nu^{(j)}. \tag{8.28}$$

Запишемо суми лівих і правих рівнянь (8.27) та (8.28), як:

$$n_1\lambda_1 + n_2\lambda_2 = \sum_{j=1}^{n_1+n_2}\lambda^{(j)}; \tag{8.29}$$

$$n_1\nu_1 + n_2\nu_2 = \sum_{j=1}^{n_1+n_2} \nu^{(j)}. \tag{8.30}$$

Ліві та праві частини (8.29) і (8.30) поділимо відповідно на $n_1 + n_2$, тоді:

$$\frac{n_1\lambda_1 + n_2\lambda_2}{n_1 + n_2} = \frac{1}{n_1 + n_2} \sum_{j=1}^{n_1+n_2} \lambda^{(j)} = \lambda; \tag{8.31}$$

$$\frac{n_1\nu_1 + n_2\nu_2}{n_1 + n_2} = \frac{1}{n_1 + n_2} \sum_{j=1}^{n_1+n_2} \nu^{(j)} = \nu = \frac{1}{\mu}. \tag{8.32}$$

Порівнюючи (8.20) з (8.31) та (8.21) з (8.32), визначимо середні значення λ та ν об'єднаної моделі, як:

$$\lambda = \frac{n_1\lambda_1 + n_2\lambda_2}{n_1 + n_2}; \tag{8.33}$$

$$\nu = \frac{n_1\nu_1 + n_2\nu_2}{n_1 + n_2}. \tag{8.34}$$

Відповідно, для обчислень j-го рівня пріоритету, обчислення середніх значень λ та ν для об'єднаної моделі від 1 до $j-1$ пріоритетів, виконаємо:

$$n = n_1 + n_2 + ... + n_{j-1}; \tag{8.35}$$

$$\lambda = \frac{n_1\lambda_1 + n_2\lambda_2 + ... + n_{j-1}\lambda_{j-1}}{n_1 + n_2 + ... + n_{j-1}}; \tag{8.36}$$

$$\frac{1}{\mu} = \nu = \frac{n_1\nu_1 + n_2\nu_2 + ... + n_{j-1}\nu_{j-1}}{n_1 + n_2 + ... + n_{j-1}}. \tag{8.37}$$

Всі стаціонарні характеристики базової моделі СМО можуть бути визначені через граничні імовірності станів P_i, як :

$$P_0 = (1 + n\rho + n(n-1)\rho^2 + n(n-1)(n-2)\rho^3 + ... + n(n-1)\ ...1\rho^n)^{-1};$$

$$P_1 = n\rho P_0; \tag{8.38}$$

$P_2 = n(n-1)\rho^2 P_0$;

……..

$P_n = n(n-1)(n-2)...1\rho^n P_0$,

де ρ - приведена інтенсивність, $\rho = \lambda / \mu$.

Позначимо середній час обслуговування вимог через T; середню пропускну спроможність вирішення задач через A (середня кількість задач в одиницю часу); середній час очікування у черзі - $T_{оч}$; середній час вирішення задачі, включаючи час перебування у черзі - T_p; середню кількість вимог, що вимагають обслуговування - ω. Згідно з можна представити, що

$$\omega = 1P_1 + 2P_2 + ... + nP_n \tag{8.39}$$

або $$\omega = n - \frac{1-P_0}{\rho}. \tag{8.40}$$

Звідси: $$T = \frac{1}{\mu}\omega$$

або $$T = \frac{1}{\mu}(n - \frac{1-P_0}{\rho}), \tag{8.41}$$

де P_0 - гранична імовірність відсутності вимог (канал вільний).

Величина A визначається як:

$$A = (1-P_0)\mu = \frac{1-P_0}{\nu}. \tag{8.42}$$

T_p можна визначити за формулою:

$$T_p = \nu r + \nu, \tag{8.43}$$

де r - кількість вимог, що знаходяться у черзі [10],

$$r = \omega - (1 - P_0). \tag{8.44}$$

Підставивши (8.44) у (8.43), отримаємо

$$T_p = \nu\omega + \nu P_0.$$

Враховуючи (8.40) та (8.41), отримаємо

$$T_p = T + \nu P_0, \tag{8.45}$$

$$T_{оч} = T_p - \nu . \tag{8.46}$$

Середній час вирішення задачі з номером k - $T_p^{(k)}$ можна визначити як

$$T_p^{(k)} = T_{оч} + \nu^{(k)} \tag{8.47}$$

Розглядаючи запропоновану модель СМО, зробимо припущення, що обслуговування вимог виконується у порядку їх надходження. Виникає питання: яким чином на стаціонарні характеристики моделі СМО може вплинути інший порядок обслуговування.

Слід зазначити, що проблемі обслуговування вимог не в порядку надходження, присвячено ряд робіт. Наприклад, була розглянута модель СМО, де джерела вимог пронумеровані від 1 до n. Канал (обслуговуючий пристрій) послідовно опитує стан кожного джерела. Якщо на момент опитування джерело вже встановило запит на обслуговування - воно обслуговується; якщо ні - канал опитує наступне джерело.

Після опитування джерела з номером m канал повертається до джерела з номером 1, тобто опитування виконуються циклічно.

При умові, що за один обхід обслуговується m джерел, вираз для часу циклу становитиме:

$$T_c = n\tau + m\nu, \tag{8.48}$$

де τ - час опитування джерел, ν - час обслуговування.

В результаті експериментального дослідження моделі зроблено дуже цікавий висновок: якщо врахувати додатковий час, що втрачається на перемикання і додати його до, власне, «звичайного» часу обслуговування, тобто визначити:

$$\nu^* = \frac{n}{m}\tau + \nu ; \quad \rho^* = \lambda \nu^* , \tag{8.49}$$

у режимі то стаціонарні характеристики цієї моделі практично співпадають з характеристиками моделі «про простій верстатів», що має аналогічні значення ρ, тобто коли $\rho^* = \rho$ при рівних значеннях n.

На даний час вивчені різні дисципліни обслуговування вимог, у тому числі випадковий вибір. Доведено, що середній час очікування такий же самий, як і випадку моделі СМО з обслуговування вимог у порядку надходження.

При вивченні дисципліни «прийшов останнім - обслужений першим» встановлено, що час, необхідний для першого повернення до стану, коли черга відсутня, якщо черга спочатку теж була відсутня, один і той же, як і при обслуговуванні вимог у порядку надходження.

Враховуючи те, що стаціонарні характеристики запропонованої моделі СМО, які визначаються виразами (8.39 - 8.47) залежать від величини P_0 - граничної імовірності відсутності вимог, тому порядок обслуговування вимог можна не враховувати.

Обслуговування вимог завжди виконується у порядку надходження.

Це припустимо при одній умові: при виконанні задачі процесор обслуговує тільки одну задачу. Внутрішній паралелізм на рівні задач відсутній.

Однак, з появою сучасних процесорів, ситуація дещо змінилася. На одному процесорі виконується одночасно декілька процесів (потоків), що стосуються не обов’язково одній задачі. Робиться це для того, щоб уникнути додаткових затримок, пов’язаних, наприклад, з очікуванням даних з пам’яті, виконання кодів ініціалізації, деініціалізації, файлового та консольного вводу, завантаження КЕШ тощо. Ці операції виконуються, як правило, паралельно з обчисленням. Якщо це стосується однієї задачі, то обчислення враховується при прогнозуванні і вимірі часу виконання задачі в монопольному режимі.

Якщо ж у черзі знаходиться декілька задач і допускається паралельне обслуговування, це буде впливати на фактичні результати дослідження. У принципі, час виконання програми, який ми отримали при монопольному виконанні, має скорочуватись, але на скільки? Визначити цю економію досить складно, оскільки це залежить як від архітектури процесора, особливостей операційної системи, так і від особливостей самої прикладної задачі та вхідних даних. Слід врахувати час на утворення і перемикання контекстів процесів (потоків) під час виконання програми.

Моделювання функціонування апаратних пристроїв процесора та ОС досить складна задача і її вирішення, мабуть, складніше за безпосередній вимір часу виконання програми. Тому, можливий вихід може бути у тому, що на початковому етапі слід ігнорувати проблему внутрішнього паралелізму, а у подальшому відкоригувати отримані результати реалізації конкретної програмної системи.

Припустимо, що робоча модель СМО відповідає графу випадкового процесу з дискретними станами і неперервним часом (рис.8.1). Задачі стають у чергу та виконуються у порядку їх надходження у систему.

Як було доведено вище, всі часові характеристики базової моделі обчислюються через граничну імовірність незайнятості обслуговуючого пристрою P_0, яка у випадку марківської моделі визначається простою формулою:

$$P_0 = (1 + n\rho + n(n-1)\rho^2 + ... + n(n-1)\ ...1\rho^n)^{-1}, \qquad (8.50)$$

де ρ - приведена інтенсивність, яка обчислюється

$$\rho = \frac{\lambda}{\mu} = \lambda\nu .$$

Оскільки припущення про показниковий характер закону обслуговування (часу виконання задач) досить умовне, з'ясуємо наскільки характер розподілу цього закону впливає на величину P_0 при одних і тих самих параметрах моделі СМО ρ та n.

Випадковий процес зветься марківським, якщо для передбачення його майбутньої поведінки, достатньо тільки знати про поточний стан процесу, а не всю його передісторію. Випадковий процес, якому непритаманні марківські властивості називають немарківським.

Порівняємо результати обчислень для марківської (перший крайній випадок закону розподілу) та немарківської моделі. За немарківську модель СМО візьмемо модель з постійним часом обслуговування, як другий крайній випадок закону розподілу.

Дослідження цієї моделі наведено у додатку А .

З результатів такого порівняння можна зробити висновок, що для дослідження часових характеристик багатозадачних систем можна і доцільно використовувати марківську модель СМО.

У загальному випадку ПЗ СРЧ створюється на базі багатопріоритетної програмної системи. Задачам кожного пріоритету можна поставити у відповідність свою базову модель СМО, яку будуть характеризувати свої параметри λ_i, ν_i, n_i. Пронумеруємо пріоритети і базові моделі. Рівень пріоритету буде відповідати його номеру, причому найвищий пріоритет буде відповідати номеру 1. Нехай всього буде m рівнів пріоритету. Тоді, загальній системі ПЗ відповідає m моделей СМО, які взаємодіють між собою певним чином, впливаючи на відповідні стаціонарні часові характеристики кожної окремої моделі СМО. Кожна з таких моделей є марківською зі скінченним джерелом та ідентична СМО класичній задачі про «простій верстатів» .

Позначимо такі ребра:

a) $R\left[x_i(t);\, x_j(t)\right] = 0,$ якщо залежності немає;

b) $R\left[x_i(t);\, x_j(t)\right] = 1;$ $x_j(t)$ залежить від $x_i(t)$;

c) $R\left[x_j(t);\, x_i(t)\right] = 1;$ $x_i(t)$ залежить від $x_j(t)$;

d) $\left.\begin{array}{l} R\left[x_i(t);\, x_j(t)\right] = 1 \\ R[x_j(t);\, x_i(t)] = 1 \end{array}\right\}$ взаємозалежність між $x_i(t)$ та $x_j(t)$.

На рис. 8.15 представлено можливий вигляд графу $G\{x(t)\}$.

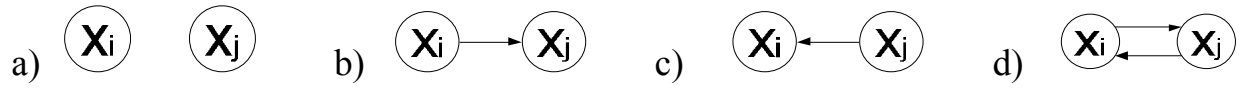


Рис. 8.15. Види залежності графу G

Розмітимо граф, який представлено на рис. 8.15 Біля ребер позначимо параметри, за якими виникає залежність між процесами. Що до складових $x_i(t)$

- моделей «загибелі та розмноження» у нашому випадку, залежність може бути по двох параметрах - загибелі та розмноження (по параметрах λ_i та μ_i).

Зупинимось на характері залежностей між моделями СМО (базовими моделями системи ПЗ).

Визначаючи характер взаємодії між моделями СМО, приймемо до уваги наступні, раніше з'ясовані, моменти.

1. Кожна з моделей є марківською і обрана система з абсолютними пріоритетами обслуговування. Для марківської моделі стаціонарні характеристики для дисципліни з обслуговуванням заново та обслуговуванням з продовженням обслуговування співпадають.

2. Порядок обслуговування вимог у базовій марківській моделі не впливає на інтервал розвантаження обслуговуючого пристрою, отже на граничну імовірність незайнятості обслуговуючого пристрою.

3. Вимоги пріоритетів від j до m не впливають на часові стаціонарні характеристики вимог $i < j$.

4. Для визначення часових характеристик вимоги j-го пріоритету достатньо розглянути дві базові моделі СМО - об'єднану модель СМО вимог від 1 до $j-1$ пріоритету та модель вимог j-го пріоритету. Вимоги пріоритетів $i > j$ не розглядаються та не враховуються.

Зважаючи на ці зауваження, базові моделі, що мають бути розглянуті для визначення стаціонарних часових характеристик задач j-го пріоритету, мають наступні особливості:

- обидві моделі є марківськими, обслуговування в обох моделях відбувається у порядку надходження вимог;
- параметри першої об'єднаної моделі визначаються згідно рівнянь (8.35), (8.36), (8.37);

- середній час обслуговування вимог j-го пріоритету не залежить від наявності вимог, пріоритет яких $i > j$, фактична середня інтенсивність обслуговування вимог j-го пріоритету залежить від наявності вимог, пріоритет яких $i < j$, тобто від вимог об'єднаної моделі вимог від 1 до $j-1$ пріоритетів;
- потоки вимог всіх рівнів пріоритету не залежать між собою.

З урахуванням цих припущень, граф $G\{x(t)\}$ буде мати вигляд, як на рис. 8.16.

Граф $G\{x(t)\}$ ілюструє той факт, що інтенсивність обслуговування вимог пріоритету j залежить від обслуговування сумісної черги вимог, пріоритет яких $i < j$. Тобто, найбільш пріоритетні вимоги 1-го рівня не залежать від вимог, пріоритет яких $i > 1$, а вимоги 2-го рівня - тільки від вимог 1-го рівня.

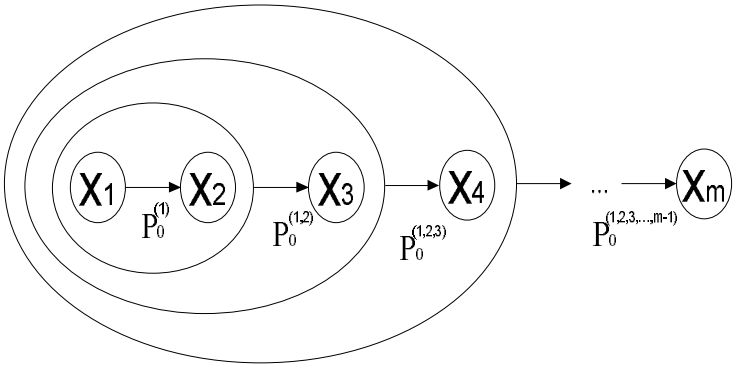


Рис. 8.16. Загальний вигляд графу $G\{x(t)\}$

Розглянемо, як можна врахувати цю залежність. Вимоги 2-го рівня пріоритету можуть обиратися на обслуговування тільки у тому випадку, коли вимоги 1-го рівня відсутні. Їх відсутність дорівнює $P_0^{(1)}$, де $P_0^{(1)}$ гранична імовірність відсутності вимог у моделі «загибелі та розмноження» з параметрами $n_1,\ \lambda_1,\ \mu_1$.

Позначимо фактичну середню інтенсивність обслуговування вимог 2-го рівня пріоритету, що знаходиться у черзі, через μ_2^*. Вона буде дорівнювати μ_2 з імовірністю $P_0^{(1)}$ і нулю, з імовірністю $(1-P_0^{(1)})$. Тоді, μ_2^* буде дорівнювати:

$$\mu_2^* = 0(1-P_0^{(1)}) + \mu_2 P_0^{(1)} = \mu_2 P_0^{(1)}. \qquad (8.51)$$

Тобто, значення $P_0^{(2)}$ можна обчислити за наведеною формулою (8.50), використовуючи параметри $n_2,\ \lambda_2,\ \mu_2^*$. Вимоги 3-го рівня пріоритету можуть обиратися на обслуговування тільки у тому випадку, коли відсутні вимоги 1-го і 2-го рівнів пріоритетів. Позначимо відповідну імовірність через $P_0^{(1,2)}$. Її можна обчислити, якщо розглянути сумісну модель СМО для вимог 1-го і 2-го рівнів пріоритету. Таку об'єднану модель можна представити, як на рис. 8.17 при відповідній заміні параметрів $(n,\ \lambda,\ \mu)$.

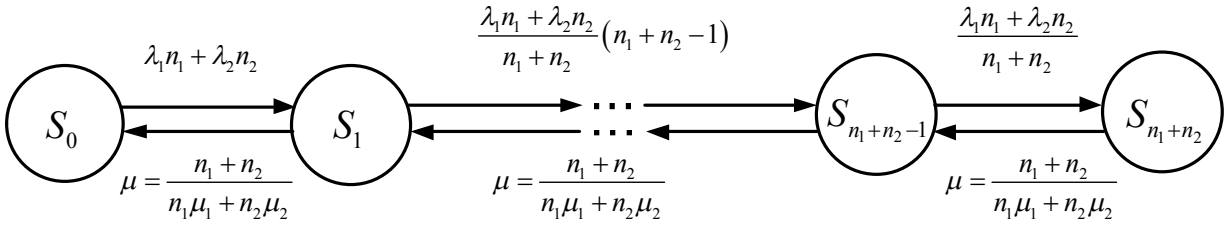


Рис. 8.17. Базова модель СМО

$$n = n_1 + n_2, \tag{8.52}$$

$$\lambda = \frac{n_1\lambda_1 + n_2\lambda_2}{n_1 + n_2}; \tag{8.53}$$

$$\mu = \frac{1}{\nu}; \quad \nu = \frac{n_1\nu_1 + n_2\nu_2}{n_1 + n_2}, \tag{8.54}$$

де $\nu_1,\ \nu_2$ - середній час виконання програм 1-го та 2-го рівнів пріоритету. Відповідно, для j-го рівня пріоритету, у об'єднаної моделі для обчислення $P_0^{(1,2,3,\ldots,j-1)}$ слід використати параметри:

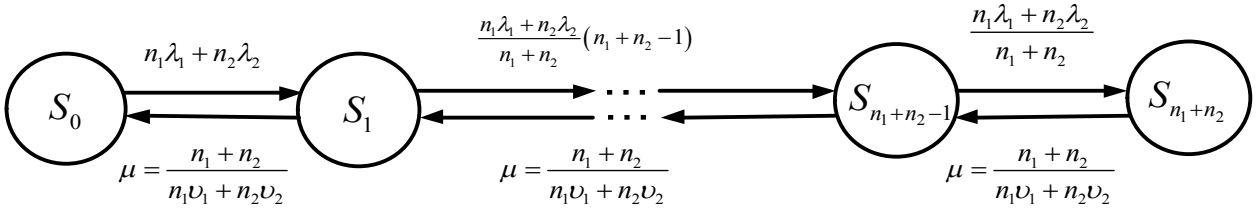
$$n = n_1 + n_2 + \ldots + n_{i-1}, \tag{8.55}$$

$$\lambda = \frac{n_1\lambda_1 + \ldots + n_{i-1}\lambda_{j-1}}{n_1 + n_2 + \ldots + n_{j-1}}, \tag{8.56}$$

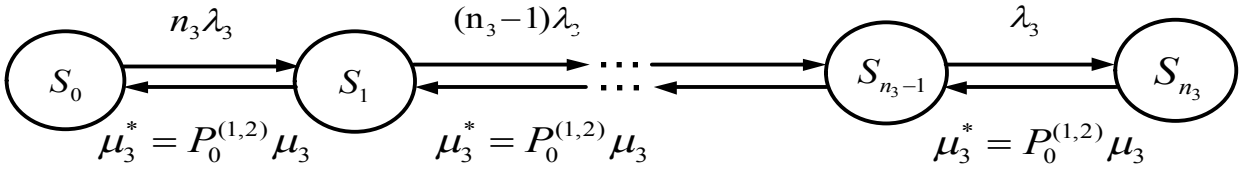
$$\mu = 1/\nu, \quad \nu = \frac{n_1\nu_1 + n_2\nu_2 + \ldots + n_{i-1}\nu_{j-1}}{n_1 + n_2 + \ldots + n_{j-1}}, \tag{8.57}$$

де $j = \overline{2,m}$.

Як приклад, на рис. 8.18 наведено графи базових моделей СМО для випадку $m = 3,\ j = 3$: а) об'єднаної базової моделі СМО для визначення характеристик 3-го рівня пріоритету; б) базової моделі СМО для визначення характеристик 3-го рівня пріоритету.



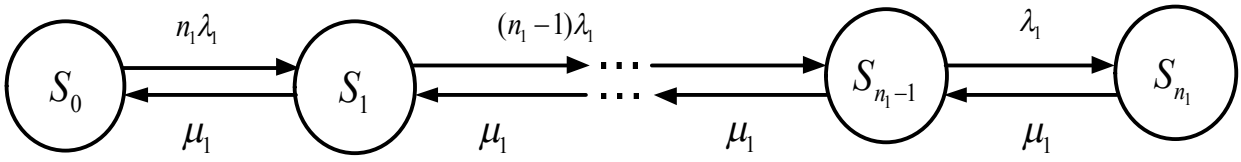
а)



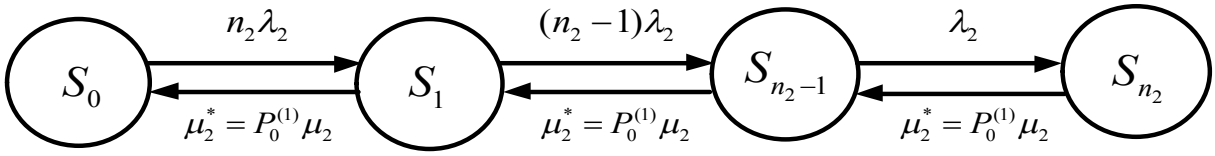
б)

Рис. 8.18. Графи базових моделей СМО для випадку $m = 3,\ j = 3$

На рис. 8.19 наведено графи базових моделей СМО для випадку $m = 3,\ j = 2$: а) об'єднаної базової моделі СМО для визначення характеристик 2-го рівня пріоритету; б) базової моделі СМО для визначення характеристик 2-го рівня пріоритету.



а)



б)

Рис. 8.19. Графи базових моделей СМО для випадку $m = 3,\ j = 2$

а) об’єднаної базової моделі СМО для визначення характеристик 2-го рівня пріоритету; б) базової моделі СМО для визначення характеристик 2-го рівня пріоритету.

Для визначення характеристик базової моделі СМО для найвищого 1-го рівня пріоритетів залишається тільки одна базова модель, оскільки об’єднана базова модель у цьому випадку не існує. Граф цієї моделі наведено на рис. 8.20.

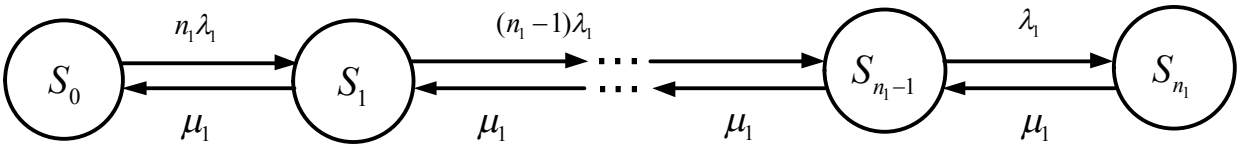


Рис. 8.20. Граф базової моделі СМО для першого рівня пріоритету

$$m = 3,\ j=1$$

8.4. Розрахунок критерію здійсненності для циклічних задач СРЧ

Для обчислення середніх часових характеристик багатопріоритетних моделей програм реалізовано метод, який викладений у розділі 8.3 та передбачає розрахунок для задач від 1 до m-го пріоритетів. Всі розрахунки для задач j-го пріоритету, $j = \overline{2,\ m}$ виконуються шляхом побудови та аналізу двох базових моделей СМО: базової моделі СМО задач j-го пріоритету та об’єднаної базової моделі СМО задач від 1-го до $j-1$ пріоритетів.

Об’єднана базова модель СМО використовується для обчислення граничної імовірності незайнятості обслуговуючого пристрою $P_0^{(1,\ldots,j-1)}$ цієї моделі СМО, а базова модель j-го пріоритету для обчислення граничної імовірності $P_0^{(j)}$ незайнятості обслуговуючого пристрою j-го пріоритету.

Обидві моделі пов’язані між собою по каналу «загибель» таким чином, що значення інтенсивності обслуговування базової моделі j-го пріоритету μ_j зменшується на величину $P_0^{(1,\ldots,j-1)}$ та становить:

$$\mu_j^* = P_0^{(1,\dots,j-1)}\mu_j \tag{8.58}$$

або $$\nu_j^* = \frac{\nu_j}{P_0^{(1,\dots,j-1)}}\,. \tag{8.59}$$

Розрахунки значення величини $P_0^{(1,\dots,j-1)}$ та $P_0^{(j)}$ виконуються за однією і тією ж формулою:

$$P_0 = (1 + n\rho + n(n-1)\rho^2 + \dots n(n-1)\dots 1\rho^n)^{-1}, \tag{8.60}$$

де $\rho = \lambda / \mu$;

n - кількість задач,

з тією різницею, що для об'єднаної базової моделі за вхідні параметри

λ , ν, n приймаються параметри:

$$n = \sum_{i=1}^{j-1} n_i\,, \tag{8.61}$$

$$\lambda = \frac{\sum_{i=1}^{j-1} n_i\lambda_i}{n}\,, \tag{8.62}$$

$$\nu = \frac{\sum_{i=1}^{j-1} n_i\nu_i}{n}\,, \tag{8.63}$$

$$\rho = \lambda\nu\,, \tag{8.64}$$

де n_i - кількість задач i-го пріоритету від 1 до $j-1$;

ν_i - середній час виконання пулу задач i -го пріоритету;

де n_i - кількість задач i-го пріоритету від 1 до $j-1$;

ν_i - середній час виконання пулу задач i -го пріоритету;

λ_i - середня інтенсивність пулу задач i -го пріоритету,

а для базової моделі j-го пріоритету:

$$n = n_j; \tag{8.65}$$

$$\lambda = \lambda_j; \tag{8.66}$$

$$\nu = \frac{\nu_j}{P^{(1,\ldots,j-1)}} = \nu_i^*; \tag{8.67}$$

$$\rho = \lambda\nu. \tag{8.68}$$

Тобто, обчислення значення величини $P_0^{(1,\ldots,j-1)}$ має допоміжний характер до обчислення значення величини $P_0 = P_0^{(j)}$, що є основною для подальшого обчислення значення величин A_j, $\mathrm{T}_{оч(j)}$, ν_j^*, T_j, $T_{p(j)}$, $\mathrm{T}_{p(j)}^{(K)}$ пріоритету j як:

$$T_j = \nu(n - \frac{1-P_0}{\rho}); \tag{8.69}$$

$$A_j = \frac{1-P_0}{\nu}; \tag{8.70}$$

$$T_{p(j)} = T_j + \nu P_0; \tag{8.71}$$

$$T_{оч(j)} = T_{p(j)} - \nu; \tag{8.72}$$

$$\nu^*(j) = \nu, \tag{8.73}$$

$$\mathrm{T}_{p(j)}^{(K)} = T_{оч(\mathrm{J})} + \nu_j^{(K)} / P_0^{(1,\ \ldots\ ,\mathrm{j}\text{-}1)} \tag{8.74}$$

де значення величин n, λ, ν та ρ є значення параметрів базової моделі j-го пріоритету згідно рівнянь (8.65 - 8.68).

Вхідні дані та редагування. Існує два способи введення даних:

- перший спосіб - вводять характеристики кожної окремої K-ї задачі $\lambda_i^{(K)}$, $\nu_i^{(K)}$, вказуючи до якого i-го пріоритету вони належать;

- другий спосіб - вводяться усереднені характеристики пулів задач - λ_i, ν_i.

У першому випадку підраховується загальна кількість n_i задач i-го пріоритету та обчислюються середні значення для пулу задач цього пріоритету:

$$\lambda_i = \frac{\sum_{K=1}^{n_i} \lambda_i^{(K)}}{n_i}, \tag{8.75}$$

$$\nu_i = \sum_{K=1}^{n_i} \frac{\nu_i^{(K)}}{n_i}. \tag{8.76}$$

Наведемо приклади розрахунку можливості виконання задач за умови використання моделі алгоритму *MQS* та використання алгоритму *FIFO* для планування кожної окремої черги задач одного пріоритету .

Слід зауважити, що кожного разу при виборі (розробці) алгоритму обчислення критерію здійсненності слід враховувати особливості обраного планувальника від якого залежить конкретний спосіб використання і обліку процесорного часу.

Нехай пули задач, які виконуються циклічно, характеризуються параметрами: ν_i, λ_i, n_i. , а період задач співпадає з відносним часом виконання задачі - t_i. Тоді умову - критерій виконання всіх задач запишемо як:

$$\sum_i \frac{n_i T_{p(i)}}{t_i} \le 1; \quad \mathrm{t}_i = 1/\lambda_i \tag{8.77}$$

Тобто, всі додатки будуть виконані, якщо завантаження процесору не буде перевищувати 1. Розподілимо задачі по чотирьох рівнях пріоритету. Оберемо: $\lambda_i = const = 1$ (деякої одиниці часу), $\nu_1 = 0.01$; $n_1 = 3$; $\nu_2 = 0.05$; $n_2 = 2$; $\nu_3 = 0.08$;

$n_3 = 2;\ \ \nu_4 = 0{,}1;\ \ n_4 = 2.$ Результати розрахунків вище наведених алгоритмів представлено у таблиці 8.2.

Спробуємо додати задачі з 4 рівнем пріоритету. Додамо ще 2 задачі. Щоб спростити умови порівняння, припустимо, що їх додавання не змінять величину ν_4. Тоді отримаємо результати, які наведені у таблиці 8.3.

Таблиця 8.2. Результати розрахунків

Пріоритет	$T_{p(i)}$	n_i	$n_i \times T_{p(i)}$
1	0,010005	3	0,030015
2	0,051777	2	0,103554
3	0,116852	2	0,233704
4	0,149592	2	0,299184

$$\sum_{\text{i}} n_i T_{p(i)} \lambda = 0{,}666457 < 1$$

Критерій - умова виконана.

Таблиця 8.3 Результати розрахунків

Пріоритет	$T_{p(i)}$	n_i	$n_i \times T_{p(i)}$
1	0,010005	3	0,030015
2	0,051777	2	0,103554
3	0,116852	2	0,233704
4	0,177594	4	0,710376

$$\sum_{\text{i}} n_i T_{p(i)} \lambda = 1{,}077649 > 1$$

Критерій не виконується (більше 1) - додавання ще 2 задач неможливе. Звернемо увагу, що величини $T_{p(i)}$ для трьох вищих пріоритетів не змінилися, як це і відповідає алгоритму MQS.

Зменшимо кількість задач четвертого рівня пріоритету до трьох. Тоді отримаємо результати, які наведені у таблиці 8.4.

Таблиця 8.4. Результати розрахунків

Пріоритет	$T_{p(i)}$	n_i	$n_i \times T_{p(i)}$
1	0,010005	3	0,030015
2	0,051777	2	0,103554
3	0,116852	2	0,233704
4	0,159957	3	0,479871

$$\sum_{i} n_i T_{p(i)} \lambda = 0{,}847144 < 1$$

Результат позитивний, критерій - менше 1.

Збільшимо кількість задач третього рівня пріоритету до чотирьох. Тоді отримаємо результати, як у таблиці 8.5.

Таблиця 8.5. Результати розрахунків.

Пріоритет	$T_{p(i)}$	n_i	$n_i \times T_{p(i)}$
1	0,010005	3	0,030015
2	0,051777	2	0,103554
3	0,131168	4	0,524672
4	0,203152	2	0,406304

$$\sum_{i} n_i T_{p(i)} \lambda = 1{,}064545 > 1$$

Критерій не виконано. Звернемо увагу на те, що зміна кількості задач на третьому рівні пріоритету не змінила величини $T_{p(i)}$ на першому та другому рівнях пріоритету, але вплинула на величини $T_{p(i)}$ на третьому та четвертому рівнях.

Висновок: використання запропонованого інструментарію дозволяє успішно моделювати планувальники задач ОС СРЧ, що засновані на алгоритмах *MQS* та *FIFO* .

8.5. Контрольні запитання до розділу 8

1. Які існують методи для визначення часу виконання програм у СРЧ.

2 Охарактеризуйте підходи до визначення часу виконання окремих програм.

3. Які фактори впливають на точний вимір часу виконання програми.

4. Який метод виміру часу виконання програми існує в ОС Windows.

5. Охарактеризуйте методику виміру часу виконання лінійних блоків програм на мові С в ОС Windows.

6. Охарактеризуйте метод прогнозування часу виконання прикладної програми за допомогою марківської моделі програми.

7. Яка марківська модель програмної системи може бути поставлена у відповідність комплексу прогам реального часу, якщо не використовуються пріоритети задач.

8. Яка марківська модель програмної системи може бути поставлена у відповідність комплексу програм реального часу, яка використовує пріоритети задач.

9. Наведіть алгоритм обчислення критерію здійсненності для програмної системи реального часу, що використовує пріоритети задач.

10. Наведіть алгоритм обчислення критерію здійсненності для марківської моделі програмної системи реального часу, що використовує пріоритети задач.

11. Які способи існують для виміру і зменшення часу відгуку певної прикладної програми у системі реального часу.

Додатково рекомендована навчальна література

1. Лекции по дисциплине “Системы реального времени”. Электронный ресурс - https/www.twirpx.com/file/124745

2. К.Е. Климентьев Системы реального времени. Электронный ресурс - https/www.twirpx.com

3. Вентцель Е.С. Исследование операций / Вентцель Е.С. - М. : Советское радио, 1972. - 552 с.

4. Вагнер Г. Основы исследования операций. Ч.3 / Г. Вангер. - М.: Мир, 1973. - 504 с.

5. Н. Джейсуол. Очереди с приоритетами / Н. Джейсуол - М. : Советское радио, 1973. - 280 с.

6. Бурдонов И.Б., Косачев А.С., Пономаренко В.Н. Операционные системы времени. Электронный ресурс - https/www.twirpx.com

7. Саати Т.Л. Элементы теории массового обслуживания и ее приложения / Т.Л. Саати. - М. : Советское радио, 1971. - 520 с.

Додаток А. **Дослідження немарківської моделі функціонування СРЧ**

Дослідження немарківської моделі виконаємо методом вкладених ланцюгів Маркова .

Побудуємо вкладений марківський ланцюг, для чого розглянемо випадковий процес у моменти регенерації, коли вимоги покидають СМО після обслуговування, $t,\ t+\delta,\ t+2\delta,\ \ldots,\ t+i\delta$, де t може бути будь-яким, зокрема $t=0,\ a\ \ \delta=\nu=1/\mu=const$. Якщо у деякий момент часу T у системі перебуває j вимог на обслуговування, то до моменту $T+\nu$ одна вимога буде обслужена, і кількість вимог у системі стане $j-1+m_{n-j}$, де m_{n-j} - число надходжень від $n-j$ джерел за інтервал $\{T,\ T+\nu\}$. У результаті, отримаємо наступну матрицю для марківських переходів, що описує стан системи в моменти часу, розділені інтервалами тривалістю ν (рис. А.1). Позначимо k_j^i - імовірність надходження j вимог від i джерел за час ν, що для $\nu=const$, визначається як:

$$k_j^i=\mathrm{C}_i^j(1-e^{-\rho})^j\,\mathrm{e}^{-(i-j)\rho},\ \ де\ \ \rho=\lambda/\mu \qquad (\text{А.1})$$

	0	1	2	...	$n-1$	n
0	k_0^n	k_1^n	k_2^n	...	k_{n-1}^n	k_n^n
1	k_0^{n-1}	k_1^{n-1}	k_2^{n-1}	...	k_{n-1}^{n-1}	0
2	0	k_0^{n-2}	k_1^{n-2}	...	k_{n-2}^{n-2}	0
3	0	0	k_0^{n-3}	...	k_{n-3}^{n-3}	0
...	0	0	0	...	...	...
$n-1$	0	0	0	...	k_1^1	0
n	0	0	0	...	k_0^0	0

Рис. А.1. Матриця марківських переходів

Напишемо рівняння для визначення граничних ймовірностей станів відповідно до наведеної матриці:

$$P_j = \mathrm{k}_j^n P_0 + \sum_{i=1}^{j+1} k_{j-i+1}^{n-i} P_i; \quad j = \overline{0,\ n-1}; \qquad \sum_{i=0}^{n} P_i = 1; \tag{A.2}$$

Рівняння (А.2) вирішимо безпосередньо або за допомогою рекурентної формули:

$$P_{j+1} = (\mathrm{k}_0^{n-j-1})^{-1} \left\{ P_j - k_j^n P_0 - \sum_{i=1}^{j} k_{j-i+1}^{n-1} P_i \right\}. \tag{A.3}$$

При великих значеннях n, використаємо рекурентну формулу. Імовірність P_0 у формулі (А.2) може бути знайдена з наступних міркувань. Нехай, задана деяка початкова величина:

$$Z_0 = qP_0; \qquad q \neq 0. \tag{A.4}$$

Тоді, позначивши $Z_i = qP_i$ та використовуючи співвідношення (А.3), (А.4), обчислимо :

$$Z_{j+1} = qP_{j+1}; \quad j = \overline{0,\ \text{n-1}}. \tag{A.5}$$

Скористаємось для визначення q умовою нормування (А.2):

$$\sum_{i=0}^{n} Z_i = \sum_{i=0}^{n} qP_i = q \sum_{i=0}^{n} P_i = q, \tag{A.6}$$

звідки:

$$P_i = Z_i \left(\sum_{i=0}^{n} Z_i\right)^{-1}. \tag{A.7}$$

Тобто,

$$P_0 = Z_0 \left(\sum_{i=0}^{n} Z_i\right)^{-1} \tag{A.8}$$

Виконаємо порівняння результатів за допомогою спеціальної програми. Її головне вікно представлено на рис. А.2.

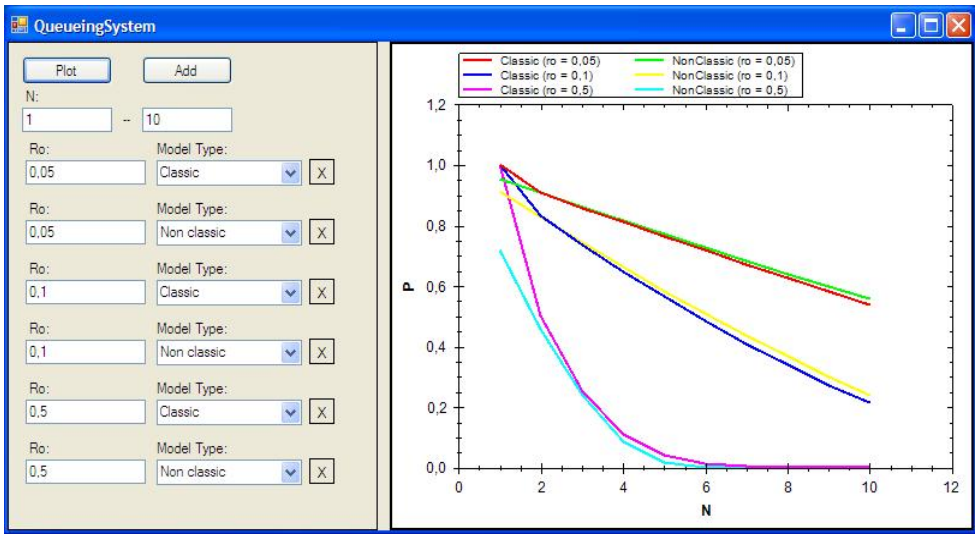


Рис. А.2. Головне вікно програми

Для того, щоб обрахувати та вивести графіки залежності P_0, як функції ρ та n, необхідно задати діапазон зміни n, параметр ρ та $ModelType$($Classik$ або $NonClassik$). Для порівняння на рис. А.3, А.4 та А.5 наведені приклади розрахунків у вигляді графіків.

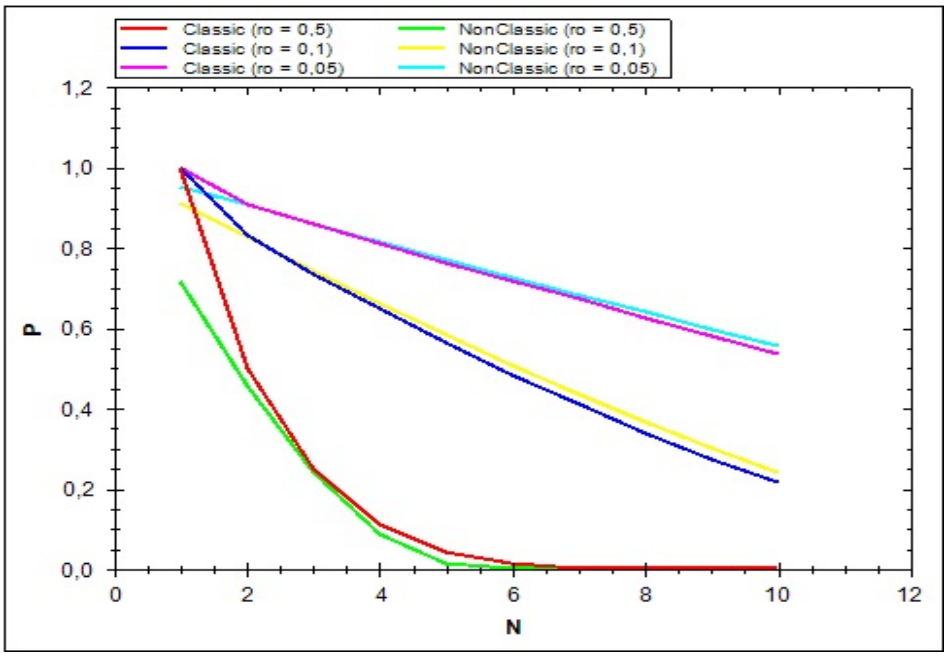


Рис. А.3. Результати розрахунку для $N = 1 \div 10$

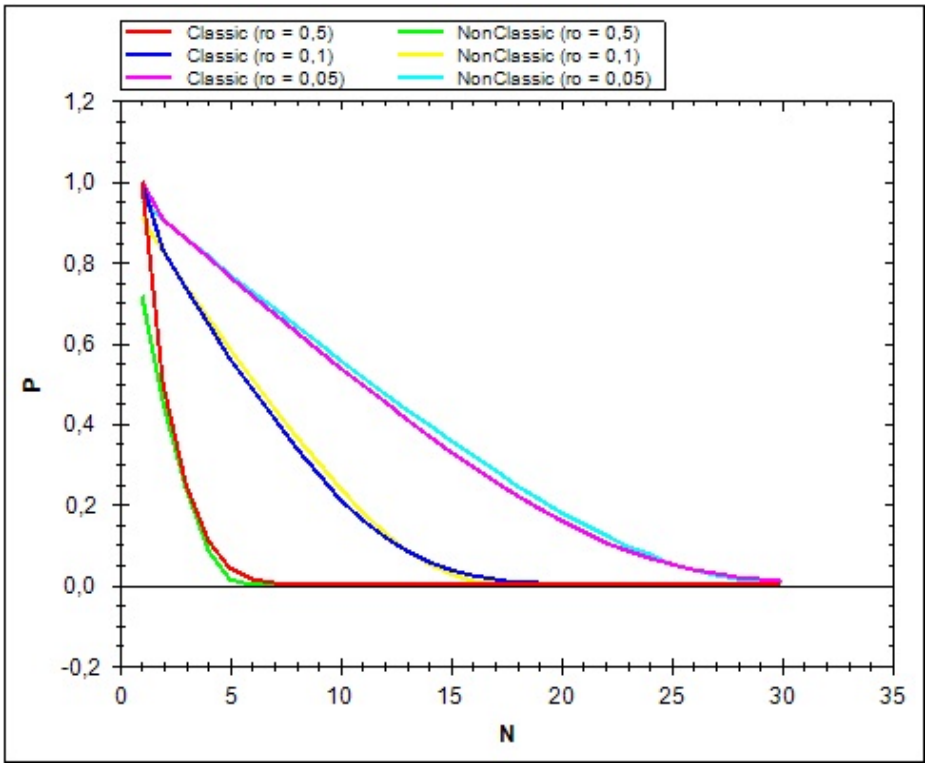


Рис. А.4. Результати розрахунку для $N = 1 \div 30$

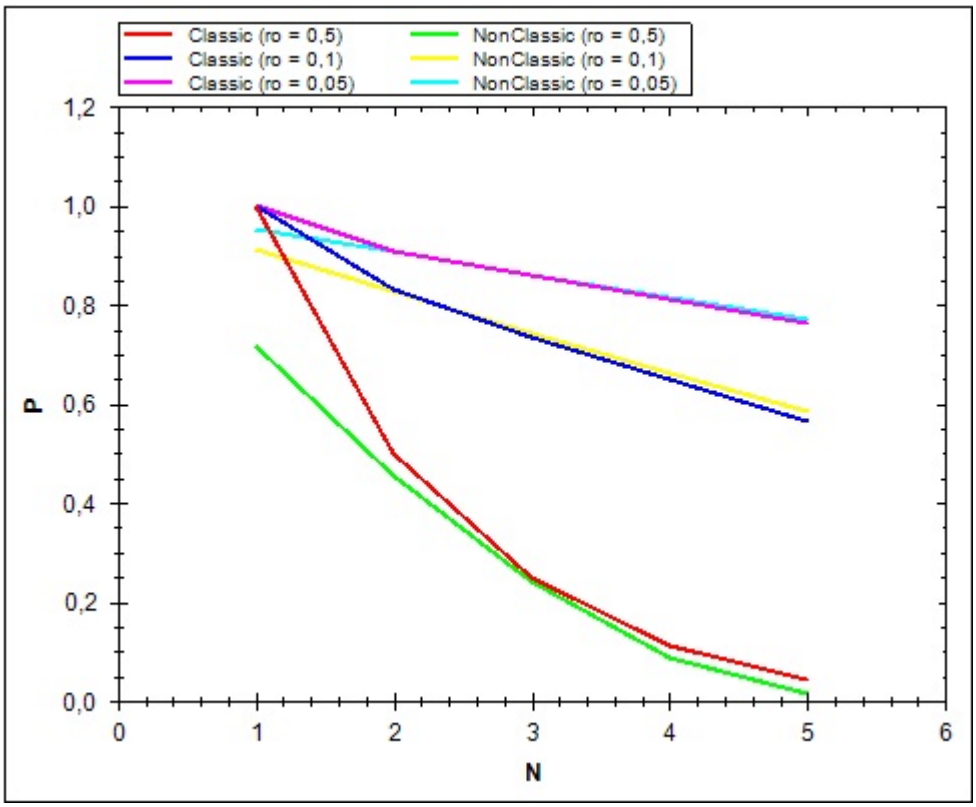


Рис. А.5. Результати розрахунку для $N = 1 \div 5$

Вони дають змогу зробити висновок, що для величин ρ у діапазоні від 0,01 до 0,2 та n - від 2 до 100 значення величини P_0 для обох класів моделей практично співпадають.

З наведених результатів (рис. А.2 - А.5) можна зробити висновок, що для дослідження часових характеристик багатозадачних систем можна і доцільно використовувати марківську модель СМО.