DOI: 10.32347/2412-9933.2023.54.48-62
УДК 004. 2

Зайцев Володимир Григорович
Доктор технічних наук, професор, професор кафедри системного програмування і спеціалізованих комп'ютерних систем, *https://orcid.org/0000-0001-9548-1959*
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ
Цибаєв Євгеній Ігорович
Кандидат технічних наук, докторант кафедри системного програмування і спеціалізованих комп'ютерних систем, *https://orcid.org/0000-0002-9115-2346*
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ

ОЦІНКА ЧАСОВИХ ХАРАКТЕРИСТИК У КОМП'ЮТЕРНИХ СИСТЕМАХ РЕАЛЬНОГО ЧАСУ З ВИКОРИСТАННЯМ СІТОК ПЕТРІ

Анотація. *Робота присвячена проблемі визначення часових характеристик задач у системах реального часу, успішність роботи яких залежить не тільки від їх логічної правильності, а й від часу, за який вони отримують результат. Визначення таких часових характеристик системи на стадії проєктування є доволі складною проблемою. Її вирішення на сьогодні засновано на використанні двох основних методів: теоретичних розрахунках, пов'язаних з обчисленням так званих критеріїв здійсненності, і моделюванням роботи системи на моделях. Серед моделей найбільш розповсюдженими є статистичні моделі систем масового обслуговування. Однак, як у першому, так і у другому варіантах неможливо отримати гарантований результат, що суттєво ускладнює процес проєктування. Останнім часом запропоновано застосовувати методи дослідження моделей, що засновані на використанні апарату сіток Петрі. У роботі запропоновано метод оцінки часових характеристик задач у системах реального часу шляхом аналізу даних, отриманих моделюванням розподілу процесорного часу між задачами згідно обраних алгоритмів планувальника шляхом використання моделі програмної системи у вигляді сітки Петрі. Метод гарантує отримання часових характеристик задач при обранні конкретних типів процесора і планувальника, що потрібно для початку технічного проєктування системи реального часу. Використання запропонованої математичної моделі та пакету прикладних програм дають змогу на ранніх стадіях розроблення системи реального часу визначити реальні терміни виконання задач та вибрати оптимальний варіант реалізації технічного і програного забезпечення. Запропоновані математичні моделі засновані на визначенні часових характеристик виконання програм за допомогою моделювання розподілу процесорного часу між задачами за умови попереднього обрання алгоритмів планування та характеристик комплексу технічних засобів.. Дослідження базуються на використанні імітаційних статистичних моделей та моделі сіток Петрі. Використання запропонованих інструментальних засобів допомагає суттєво зменшити терміни аналізу можливих варіантів реалізації системи реального часу, підвищує якість проєкту та суттєво зменшує загальні терміни і вартість усієї розробки. У роботі детально розглянуто процес моделювання багатозадачної обчислювальної системи з використанням апарату сіток Петрі. На конкретному прикладі наведено моделювання роботи комплексу програм, включаючи розроблення алгоритмів моделювання, лістінгу програми моделювання та аналіз отриманих результатів.*

Ключові слова: модель; задача реального часу; сітка Петрі

Вступ

Системою реального часу (СРЧ) [1] називається система, в якій успішність роботи будь-якої програми залежить не тільки від її логічної правильності, а й від часу, за який вона отримує результат. Якщо часові обмеження не задоволені, то фіксується збій у роботі системи.

Розрізняють сильні (hard) та слабкі (soft) обмеження до вимог реального часу. Якщо запізнення програми призводить до повного порушення роботи системи керування, то говорять про сильне порушення в часі (жорсткі СРЧ). Якщо запізнення призводить тільки до тимчасової втрати продуктивності, то говорять про слабке порушення в реальному часі.

 © В. Г. Зайцев, Є. І. Цибаєв

Часто поняття "система реального часу" плутають з поняттям "швидка система". Це не зовсім правильно. Час затримки реакції СРЧ на подію вже не так і важливий, якщо цього часу достатньо для гарантування часових обмежень щодо певних додатків. Тобто система реального часу реагує в передбачений час на непередбачувані появи зовнішніх подій.

Не існує операційних систем жорсткого або м'якого реального часу. Поняття системи реального часу і операційної системи реального часу (ОССРЧ) часто змішується.

Система реального часу – це конкретна система, пов'язана з реальним об'єктом. Вона включає в себе необхідні апаратні і програмні засоби, операційну систему і прикладне програмне забезпечення. Операційна система реального часу – це тільки інструмент, що допомагає створити конкретну систему реального часу. Тому безглуздо говорити про системи жорсткого або м'якого реального часу. Можна говорити лише про те, чи можна за допомогою конкретної операційної системи побудувати конкретну систему реального часу [2; 4].

Основні параметри завдань (задач) реального часу

Кожна задача РЧ характеризується такими часовими параметрами [1]: r – (Realize Time) – момент часу, коли виникає необхідність у передачі управління виконуваному завданню; d – (Absolute Deadline) – абсолютний крайній термін, момент часу, до якого задача повинна завершити роботу; s – (Start Time) – момент часу, коли задача фактично починає виконуватись на процесорі; c – (Completion Time) – момент часу, коли задача закінчила роботу, опрацювавши подію (розв'язавши задачу); D – (Relative Time) – відносний крайній термін: $D = d - r$; E – (Execution Time) – час виконання задачі: $e = c - s$; R – (Response Time) – час відгуку: $R = c - r$.

Діаграма на рис. 1 ілюструє ці параметри:

діаграма робота задачі має такі параметри: $r = 2,\ d = 11,\ s = 5,\ c = 9,\ D = 11 - 2 = 9,\ e = 9 - 5 = 4,\ R = 9 - 2 = 7.$

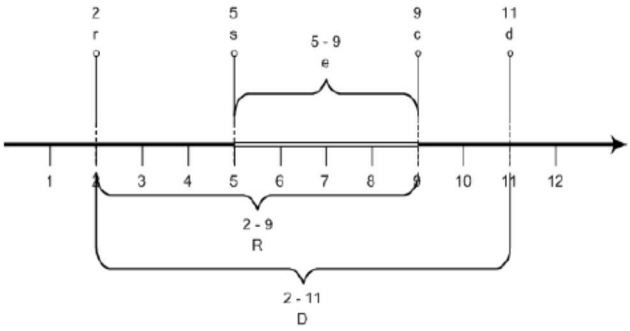

Рисунок 1 – Параметри завдань (задачі)

Запропонована модель

Загальна схема дослідження може бути представлена, як на рис. 2. Вона складається з генератора потоку задач, алгоритмів планування і диспетчеризації та моделі виконання задач у часі.

Генератор потоку задач генерує послідовності задач у межах періоду (гіперперіоду) надходження задач. Планувальник відповідно до обраного алгоритму планування визначає послідовності (номери) задач, яким слід надати процесорний час і, своєю чергою, отримує від моделі виконання задач інформацію про моменти надання процесорного часу кожній задачі, проценти та моменти завершення виконання, а також про ступінь завантаженості процесора за період виконання T_i.

У моделі виконання задач попередньо має бути задана інформація про час виконання кожної із задач e_i, а також забезпечена можливість надання вибраній на виконання задачі повного часу виконання або надання такого часу окремими порціями (режим з витисканням) за час періоду T_i.

У роботі запропоновано модель виконання задач побудувати за допомогою апарату сіток Петрі. Структура сіток Петрі являє собою сукупність позицій та переходів і складається з двох типів вузлів: позицій (кружечків) та планок (рисок) – переходів. Орієнтовані дуги поєднують позиції і переходи. Дуга, що спрямована від позиції P_i до переходу τ_j визначає позицію, що являє собою вхід у перехід. Вихідна позиція вказується дугою від переходу до позиції. Входи та виходи можуть бути кратними, що визначається на схемі або кількістю дуг, або цифрою k на дузі.

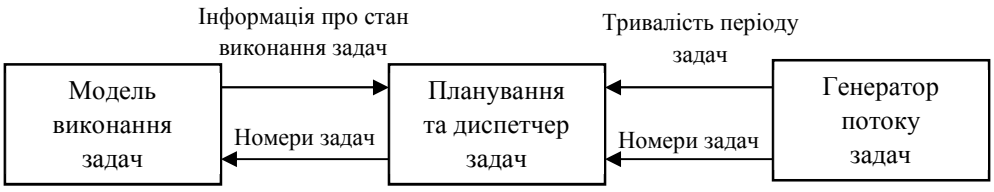

Рисунок 2 – Загальна схема моделювання

Кожна позиція може мати певну кількість маркерів, які можуть рухатись з позиції на позицію при виконанні (запуску) переходів [7; 8].

Власне сітка Петрі – це орієнтований дводольний мультиграф G. Формально граф $G=(V, Z)$, де $V=\{v_1, v_2, ...v_s\}$ – множина вершин, $Z=(z_1, z_2, ...z_n)$ – комплект орієнтованих дуг $z_i=\{v_i, v_k\}$, де v_i, v_k є V. Множина V розбита на дві непересічні множини P і θ таких, що $V=P\cup\theta$, $P\cap\theta=0$.

При аналітичному способі визначення сітки вона може бути представлена як $S=(P, \theta, F, H, \mu_0)$, де додатково визначені вхідна функція F та вихідна H. Через $F(\tau_j)$ позначається множина вхідних позицій, а через $H(\tau_j)$ – множина вихідних позицій переходу (τ_j), μ_0 – початкове маркування (розподіл маркерів по позиціях). Маркування сітки S – це присвоювання певної кількості маркерів, що перебувають в позиціях (присвоєні) P_i.

Маркування μ може бути також представлене n-мірним вектором $\mu=(\mu_0, \mu_1, ...\mu_n)$, де μ_j – розподіл фішок серед всіх позицій P_i у кожному конкретному стані сітки. Тобто, маркована сітка Петрі – сукупність структури сітки $S=(P, \theta, F, H, \mu)$ та маркування μ, або $S=(P, \theta, F, H, \mu)$.

Перехід називають дозволеним, якщо кожна з його вхідних позицій має кількість маркерів не менше ніж кількість дуг із позицій в перехід. Кратні маркери необхідні для кратних вхідних дуг. Маркери (фішки) у вхідній позиції, які дозволяють перехід, називаються дозволяючими фішками. Наприклад, якщо позиції P_1 та P_2 – входи для переходу τ_j, то перехід τ_j дозволено, якщо P_1 та P_2 мають хоча б по одній фішці.

Запуски переходів відповідають деяким подіям, що відбуваються в процесах, що моделюються за допомогою сітки Петрі. Після запуску переходів, що відповідають події, дозволені переходи спрацьовують, а недозволені – не спрацьовують. У загальному випадку сітка Петрі змінює своє маркування μ на нове μ^1. Тобто

$$\mu_0 \to \mu_1. \qquad (1)$$

Якщо переходи не запускаються, зберігається попереднє маркування сітки.

Відмітимо, що запустити можна тільки дозволені переходи, а тому кількість фішок у кожній позиції завжди залишається невід'ємною. Запуск переходу ніколи не видалить фішку, яка відсутня у вхідній позиції. Якщо деяка вхідна позиція переходу не має достатню кількість фішок, то перехід не дозволяється і не може бути запущений.

Як приклад, на рис. 3 наведена сітка Петрі, що задана графічним способом.

При аналітичному способі, окрім множини позицій P_1 та переходів θ, також мають бути задані вхідна H та вихідна F функції [3; 7].

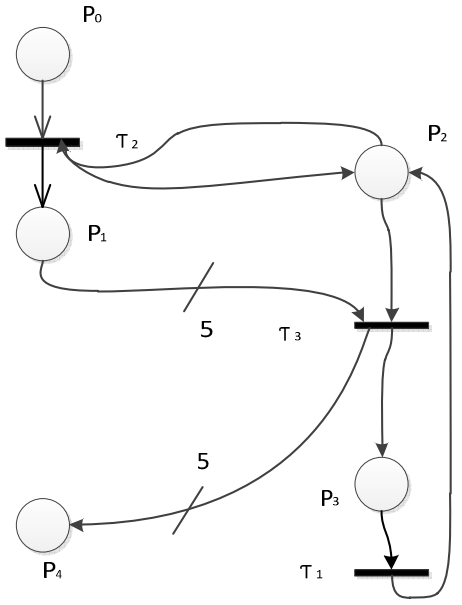

Рисунок 3 – Сітка Петрі

Через $F(\tau_j)$ позначається множина вхідних позицій, а через $H(\tau_j)$ – множина вихідних позицій переходу τ_j. Розглянемо це на прикладі. На рис. 3 наведена сітка Петрі. Згідно сітки на цьому рисунку:

$F(\tau_1)=(P_3)$; $H(\tau_1)=(P_2)$;

$F(\tau_2)=(P_0, P_2)$; $H(\tau_2)=(P_1, P_2)$;

$F(\tau_3)=(P_1, P_1, P_1, P_1, P_1, P_3)$;

$H(\tau_3)=(P_3, P_4, P_4, P_4, P_4, P_4)$.

Аналітичне представлення сітки в матричній формі може бути представлене так:

$$S=(P, \theta, M^-, M^+, \mu_0), \qquad (2)$$

де M^- та M^+ – матриці вхідних та вихідних інцеденцій розміром m x n, де m – кількість переходів; n – кількість позицій.

Кожний елемент матриці M^- дорівнює кратності дуг, що входить в i-й перехід з j-ї позиції, а елемент матриці M^+ дорівнює кратності дуг з i-го переходу в позицію j.

Матриця інцедентності сітки Петрі M визначається так:

$$M=M^+-M^-. \qquad (3)$$

Для наведеної на рис. 3 сітки початкове маркування $M=[5\,0\,0\,1\,0]$, оскільки $[P_0]$ = 5, $[P_1]$ = 0, $[P_2]$ = 0, $[P_3]$ = 1, $[P_4]$ = 0, де $[P_i]$ – кількість маркерів у позиції P_i, а матриці M^-, M^+, та M представлені, як у табл. 1 – 3.

*Таблиця 1 – **Матриця** M^-*

	P0	P1	P2	P3	P4
τ_1	0	0	0	1	0
τ_2	1	0	1	0	0
τ_3	0	5	1	0	0

*Таблиця 2 – **Матриця** M^+*

	P0	P1	P2	P3	P4
τ_1	0	0	1	0	0
τ_2	0	1	1	0	0
τ_3	0	0	0	1	5

*Таблиця 3 – **Матриця** M*

	P0	P1	P2	P3	P4
τ_1	0	0	1	-1	0
τ_2	-1	1	0	0	0
τ_3	0	-5	-1	1	5

Керування сіткою виконується шляхом запуску переходів. Перехід може бути запущено лише у тому випадку, якщо він дозволений. Як відомо [3], нове маркування після запуску переходу може бути отримане після обчислення нового маркування:

$$\mu_{i+1} = \mu_i + B_j \text{ x } M, \qquad (4)$$

де B_j – одиничний вектор – рядок переходу j, усі компоненти якого дорівнюють нулю, за винятком компоненти, що відповідає номеру j та дорівнює одиниці. Якщо перехід спрацьовує, то отримується нове маркування μ_{i+1}, в якому не може бути від'ємної кількості фішок у позиціях. Якщо це має місце при обчисленні, то це свідчить про те, що перехід не спрацьовує, а маркування залишається незмінним (попереднім), тобто μ_i.

Обчислимо значення величин $B_j \text{ x } M$ щодо запуску переходів τ_1, τ_2, τ_3.

Ці величини є константами, які додаються поелементно до кожного елемента попереднього маркування $[P_i]$ після кожного спрацювання переходу, змінюючи попереднє маркування на нове. Якщо перехід не спрацьовує, ця операція не виконується, тож маркування не змінюється.

Отже:

$$\tau_1 : B_1 \text{ x } M = [1\ 0\ 0] \text{x } M = [0\ 0\ 1\ -1\ 0];$$

$$\tau_2 : B_2 \text{ x } M = [0\ 1\ 0] \text{x } M = [-1\ 1\ 0\ 0\ 0];$$

$$\tau_3 : B_3 \text{ x } M = [0\ 0\ 1] \text{ x } M = [0\ -5\ -1\ 1\ 5].$$

Введемо поняття такту запуску сітки, що представлена на рис. 3.

Під цим будемо розуміти спробу запустити кожного разу (такту) послідовність переходів τ_1, τ_2, τ_3. На кожному такті, як це видно з табл. 4, частина з кожних трьох переходів не спрацьовує. Якщо перехід не спрацьовує, він не змінює маркування сітки, тож маркування залишається попереднім (як до запуску переходу, який не спрацював).

Аналізуючи зміну маркування сітки у процесі виконання тактів запуску, можна стверджувати, що в кожному такті один маркер з позиції P_0 переходить у позицію P_1, де накопичується N маркерів ($N = 5$ у даному випадку), і накопичені 5 маркерів після виконання переходу τ_3 на п'ятому такті переходять у позицію P_4. Окрім того, маркер з позиції P_2 переходить у позицію P_4 сигналізуючи про завершення передачі N маркерів.

Якщо пов'язати перехід маркера з позиції P_0 з деякою умовною одиницею часу Δt, то можна вважати, що перехід деякої кількості N маркерів буде відповідати T умовним одиницям часу, де

$$T = N\Delta t, \qquad (5)$$

а накопиченню k_i маркерів у позиції P_1 до переходу у позицію P_4 буде відповідати інтервал $k_i \Delta t$ умовних одиниць часу.

З аналізу наведеної на рис. 3 сітки Петрі видно, що переходи маркерів з позиції P_0 у позицію P_1 і з позиції P_1 у P_4 виконуються за тактами, що складаються з послідовного збудження сукупності переходів τ_1, τ_2, τ_3.

Якщо вважати, що величина умовного часу періоду задач РЧ дорівнює T, а час виконання задачі i в умовних одиницях часу процесора складає $k_i \Delta t$, то можна вважати, що

$$e_i = k_i \Delta t. \qquad (6)$$

І сітка Петрі, що наведена на рис. 3, відповідає моделі виконання однієї задачі за умови що $e_i = T$. При цьому перехід маркера з позиції P_3 у позицію P_2 відповідає моменту надання задачі процесора, а повернення його у позицію P_3 – завершенню виконання задачі.

При розробці загальної моделі роботи n задач слід передбачити можливість переривання і поновлення видачі задачі процесорного часу. Оскільки функціонування моделі полягає у послідовному збудженні трьох переходів за кожним тактом, то ця задача може бути розв'язана шляхом припинення їх видачі на черговому такті, і повернення до цього при поверненні задачі процесора на чергових тактах до її повного виконання.

Враховуючи сказане, загальну модель видачі процесорного часу декільком задачам можна

представити, як на рис. 4 [3]. З цього рисунку видно, що деяка сукупність з n задач, які представлені моделями сітки Петрі, еквівалентними моделі сітки на рис. 3, об'єднані в одну модель шляхом включення позиції P_0 у кожну з моделей – складових. При цьому початкова кількість маркерів у позиції P0 має відповідати T умовним одиницям часу. Надання процесора конкретній задачі буде відповідати збудженню її власних переходів на відповідному такті. Припиненню виконання задачі і постановці на виконання іншої задачі буде відповідати припинення збудження переходів першої і збудження переходів другої (при виконанні чергового такту роботи моделі).

Для того щоб розрізняти позиції (за винятком позиції P_0 , яка є загальною для всіх моделей – складових) та переходи моделей окремих задач, запропоновано ввести подвійну індексацію, де перший індекс додатково вказує на номер задачі, тобто P_{i1} , P_{i2} і т. д. Отже, запропоновано простий спосіб постановки та зняття з виконання конкретної задачі – шляхом запуску відповідних до задачі переходів при виконанні чергового такту.

На завершення опису моделі слід вирішити ще одне питання – як виміряти невикористаний час процесора за час періоду. Цей час можна врахувати, якщо додати в загальну модель ще одну складову (псевдозадачу – A_0), час виконання якої дорівнює одиниці ($k_0 = 1$) . Тоді кількість маркерів, що будуть накопичені у позиції P_{04} за весь проміжок часу T , і буде цією величиною в умовних одиницях.

Розглянемо роботу цієї моделі на прикладі. Шляхом програмного моделювання роботи системи реального часу довести можливість (або неможливість) задовольнити вимоги до виконання комплексу програм *Ai* реального часу, що виконуються циклічно з періодом *T*, за таких умов:

T=42. *A1: r=0, e=12,d= 30* або скорочуючи *A1: 0, 12; A2*: 2, 14, 34; *A3*:5, 6, 42; *A4*: 4, 10, 16.

Виконання

Допускається використання дисципліни обслуговування з динамічним пріоритетом EDF з витисканням та без пріоритетної дисципліни [5]. Порівняйте ефективність роботи двох планувальників з точки зору необхідної швидкодії процесора для обов'язкового виконання вимог. Для моделювання роботи комплексу програм була використана модель мережі Петрі [3]. Логіка роботи мережі Петрі реалізована за допомогою класу *PetriModel*. Конструктор класу приймає список задач та значення періоду й ініціалізує матрицю інцидентності мережі Петрі і вектор маркерів. Допускається виконання на кожному такті певної програми (метод *doTask*), номер якої в кожний момент часу задається планувальником. Реалізовано два планувальники у вигляді двох класів, які успадковуються від спільного базового класу *Planner*. Планувальник має метод *getNextTask*, що за номером такту видає номер програми для виконання. Планувальник без пріоритетів (клас *NonPriorityPlanner*) на кожному кроці послідовно переглядає список всіх задач. Якщо чергова задача вже може бути розпочата ($r_i \geq t$, де – номер поточного такту) та ще не була розв'язана, то саме вона і подається на виконання.

Якщо жодної такої задачі немає, то подається фонова задача (що також використовується для підрахунку невикористаного часу). Планувальником EDF із витисканням (клас *EDFSqueezingPlanner*) на кожному такті серед усіх задач, які ще не завершилися та готові для виконання ($r_i \geq t$), вибирається та, в якої найменшим є значення абсолютного крайнього терміну (d_i). Програма дає змогу користувачу задавати значення періоду, кількість задач, а також параметри задач(*r*, *e* та *d*). Також користувач може обрати тип планувальника: без пріоритетів або EDF із витисканням. Результати моделювання лобіюються та в подальшому виводяться у вигляді діаграми. Планувальником EDF із витисканням (клас *EDFSqueezingPlanner*) на кожному такті серед усіх задач, які ще не завершилися та готові для виконання ($r_i \geq t$), вибирається та, в якої найменшим є значення абсолютного крайнього терміну (d_i). Весь код програми розташовано у файлі Lab.cpp. Програма дає змогу користувачу задавати значення періоду, кількість задач, а також параметри задач(*r*, *e* та *d*). Також користувач може обрати тип планувальника: без пріоритетів або EDF із витисканням. Результати моделювання лобіюються та в подальшому виводяться у вигляді діаграм.

Висновки з моделювання

Розроблено та реалізовано програму моделювання роботи комплексу програм на основі моделі мережі Петрі. Виконано моделювання для заданого набору задач із заданими параметрами для випадків використання планувальника без приорітетів та планувальника EDF із витисканням. В обох випадках всі задачі встигають бути виконані за період, а невикористаний час дорівнює нулю.Проте в обох випадках обов'язкові умови виконуються не для всіх задач. В обох випадках одна задача лишається протермінованою. Проте у разі використання EDF із витисканням задачу протерміновано на значно менший час, ніж при використанні планувальника без приорітетів.

Таблиця 4 – ***Виконання сітки Петрі***

Номер такту	Перехід	Початкове маркування	Результат запуску	Спрацювання переходу	Нове маркування
1	τ_1 τ_2 τ_3	5 0 0 1 0 5 0 1 0 0 4 1 1 0 0	5 0 1 0 0 4 1 1 0 0 4 -4 0 1 5	+ + -	5 0 1 0 0 4 1 1 0 0 4 1 1 0 0
2	τ_1 τ_2 τ_3	4 1 1 0 0 4 1 1 0 0 3 2 1 0 0	4 1 2 -1 0 3 2 1 0 0 3 -3 0 1 5	- + -	4 1 1 0 0 3 2 1 0 0 3 2 1 0 0
3	τ_1 τ_2 τ_3	3 2 1 0 0 3 2 1 0 0 2 3 1 0 0	3 2 2 -1 0 2 3 1 0 0 2 -2 0 1 5	- + -	3 2 1 0 0 2 3 1 0 0 2 3 1 0 0
4	τ_1 τ_2 τ_3	2 3 1 0 0 2 3 1 0 0 1 4 1 0 0	2 3 2 -1 0 1 4 1 0 0 1 -1 0 1 5	- + -	2 3 1 0 0 1 4 1 0 0 1 4 1 0 0
5	τ_1 τ_2 τ_3	1 4 1 0 0 1 4 1 0 0 0 5 1 0 0	1 4 2 -1 0 0 5 1 0 0 0 0 0 1 5	- + +	1 4 1 0 0 0 5 1 0 0 0 0 0 1 5

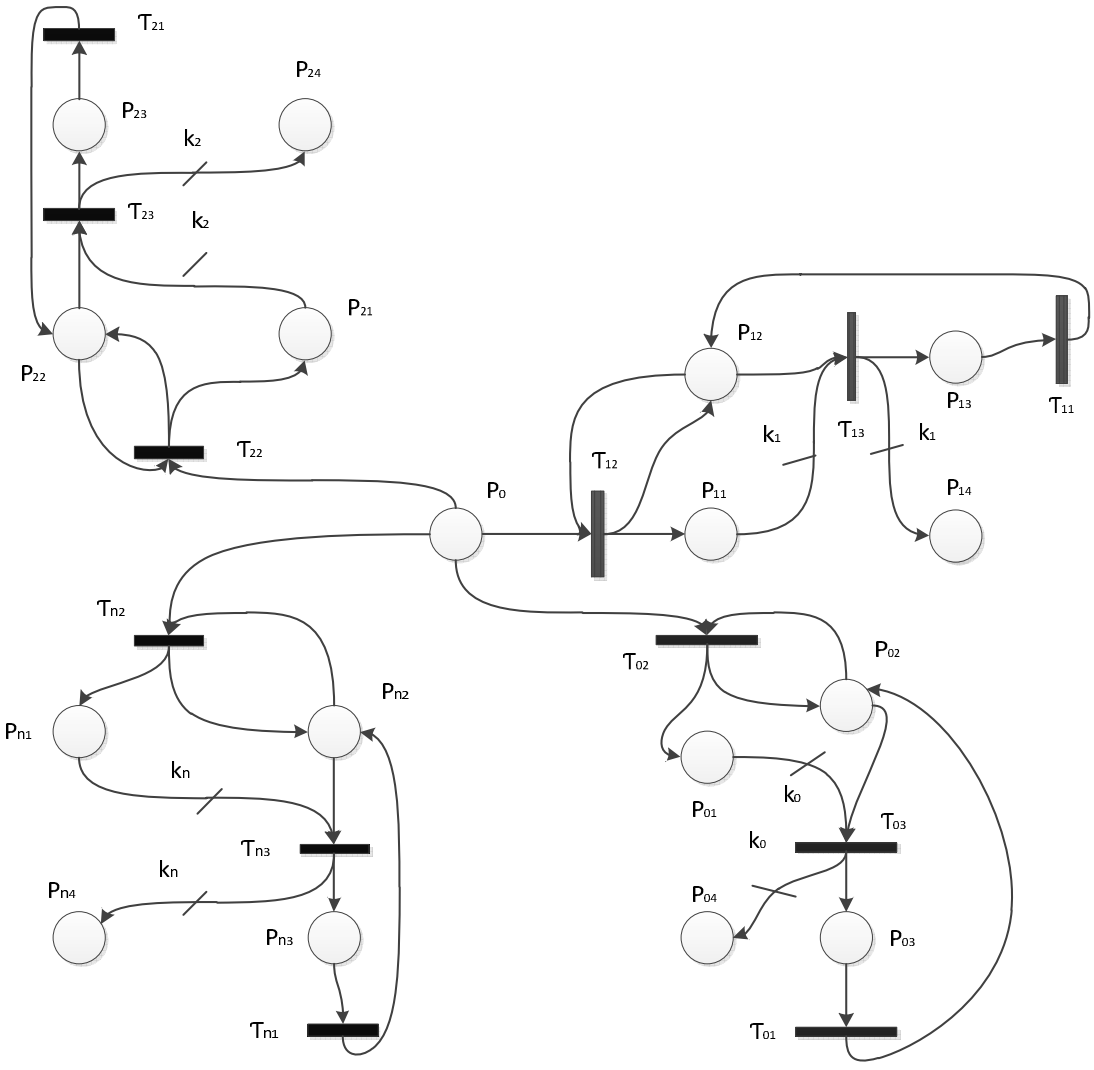

Рисунок 4 – Модель виконання задач

Код:

time_measure.h

```
#include
<iomanip>   #include
<iostream>  #include
<memory>    #include
<numeric>
#include
<string>    #include
<utility>   #include
<vector>
struct Task
{
    Task() = default;
Task(int _r, int _e, int _d) :r(_r), (_e),
    d(_d)
    {
    }
int r = 0;
int e = 0;
int d = 0;
int start = -1;int
end = -1;
};
using Tasks = std::vector<Task>;
class PetriModel
{
public:
PetriModel(const Tasks& tasks, int period):
    m_tasks_n(tasks.size())
    {
    initMarkers(period);
    initJumps(tasks);
    }
    bool doTask(size_t task_id)
    {
                        const size_t task_offset = task_id * 3;
                        const bool t1_done = tryJump(task_offset + 0);
                        const bool t2_done = tryJump(task_offset + 1);
                        const bool t3_done = tryJump(task_offset + 2);
        return t3_done;
    }
    int getTaskTime(size_t task_id)
    {
        return m_markers[task_id * 4 + 4];
    }
private:
    void initMarkers(int period)
    {
    m_markers.assign(1 + m_tasks_n * 4, 0);
    m_markers[0] = period;
        for (size_t i = 0; i < m_tasks_n; ++i)
        {
            m_markers[i * 4 + 3] = 1; // P3
        }
```

```
    }
    void initJumps(const Tasks& tasks)
    {
    m_jumps.assign(3 * m_tasks_n, std::vector<int>(1 + m_tasks_n * 4, 0));for (size_t i = 0;
    i < m_tasks_n; ++i)
    {
        const auto& task = tasks[i]; auto& t1 =
        m_jumps[i * 3 + 0];
        auto& t2 = m_jumps[i * 3 + 1];
        auto& t3 = m_jumps[i * 3 + 2]; const size_t
offset = 1 + i * 4;const size_t p0 = 0;const size_t p1 =
offset + 0;const size_t p2 = offset + 1;const size_t p3 =
offset + 2;const size_t p4 = offset + 3;
                    t1[p2] = 1;
                    t1[p3] = -1;
                    t2[p0] = -1;
                    t2[p1] = 1;
                    t3[p1] = -task.e;
                    t3[p2] = -1;
                    t3[p3] = 1;
                    t3[p4] = task.e;
            }
     }
    bool tryJump(size_t row_id)
    {
        auto markers_new = m_markers; const
        auto& row = m_jumps[row_id];
        for (size_t i = 0; i < row.size(); ++i)
        {
                markers_new[i] += row[i]; if
                (markers_new[i] < 0)
                {
                    return false;
                    }
            }
            m_markers = std::move(markers_new);
            return true;
        }
        using Row = std::vector<int>;
        size_t m_tasks_n;
        Row m_markers;
        std::vector<Row> m_jumps;
     };
     class Planner
     {
     public:
            virtual ~Planner() = default;
            virtual size_t getNextTask(int tick) = 0;
     };
     class NonPriorityPlanner: public Planner
     {
     public:
            NonPriorityPlanner(const Tasks& tasks):
            m_tasks(tasks)
            {
            }
```

```
        virtual size_t getNextTask(int tick) override
        {
            for (size_t i = 0; i < m_tasks.size(); ++i)
            {
                const auto& task = m_tasks[i];
                if (task.r <= tick) // already ready
                {
                    if (task.end == -1) // not finished
        using Row = std::vector<int>;
  size_t    m_tasks_n;
  Row m_markers;
        std::vector<Row> m_jumps;
  };
  class Planner
  {
  public:
        virtual ~Planner() = default;
        virtual size_t getNextTask(int tick) = 0;
  };
  class NonPriorityPlanner: public Planner
  {
  public:
       NonPriorityPlanner(const   Tasks&   tasks):
     m_tasks(tasks)
{
}
virtual size_t getNextTask(int tick) override
{
        for (size_t i = 0; i < m_tasks.size(); ++i)
        {
            const auto& task = m_tasks[i];
            if (task.r <= tick) // already ready
            {
                if (task.end == -1) // not finished
                {
                    return i;
                }
            }
        }
        return m_tasks.size() – 1; // dummy as default
    }
  private:
        const Tasks& m_tasks;
  };
  class EDFSqueezingPlanner : public Planner
  {
  public:
        EDFSqueezingPlanner(const  Tasks&  tasks)  :
        m_tasks(tasks)
        {
        }
        virtual size_t getNextTask(int tick) override
        {
            size_t task_id = m_tasks.size() – 1; // dummy as defaultfor
            (size_t i = 0; i < m_tasks.size(); ++i)
            {
```

```
                const auto& task = m_tasks[i];
                if (task.r <= tick) // already ready
                {
                    if (task.end == -1) // not finished
                    {
                        if (task.d < m_tasks[task_id].d) // find lowest deadline
                        {
                            task_id = i;
                        }
                    }
                }
            }
            return task_id;
        }
    private:
        const Tasks& m_tasks;
    };
    std::unique_ptr<Planner> create_planner(const Tasks& tasks, const std::string& type)
    {
        if (type == "EDFs")
        {
      return std::unique_ptr<Planner>(new EDFSqueezingPlanner(tasks));
    }
    else
    {
      return std::unique_ptr<Planner>(new NonPriorityPlanner(tasks));
        }
    }
    enum class LogStatus
    {
        None,
    Started,
    Processing,
    Finished
    };
    using LogLine = std::vector<LogStatus>;
using Logging = std::vector<LogLine>;
    char get_visualization(LogStatus s)
    {
        switch (s)
        {
    case   LogStatus::None:
        return '.';
    case      LogStatus::Started:
        return '+';
    case      LogStatus::Processing:
        return '=';
    case      LogStatus::Finished:
        return '-';
        default:
            return '?';
        }
    }
    char get_visualization(const Task& task, size_t tick)
    {
        if (task.r == tick)
```

```
            {
                return 'O';
            }
        else if (task.d == tick)
        {
            }
       else          return 'X';
          {
          }
        }
                return ' ';
       void draw_diagram(const Logging& logging, const Tasks& tasks, size_t tasks_to_visualize, size_t
time_to_visualize)
        {
            for (size_t i = 0; i < tasks_to_visualize; ++i)
            {
                std::cout << "Task #" << std::left << std::setw(4) << i + 1;
                for (size_t j = 0; j < time_to_visualize; ++j)
                {
                    if (i < logging.size() && j < logging[i].size())
                {
                std::cout << get_visualization(logging[i][j]);{
                    }
                    else
                    {
                        std::cout << get_visualization(LogStatus::None);
                    }
                }
                std::cout << std::endl;
                std::cout << std::string(10, ' ');
                for (size_t j = 0; j < time_to_visualize; ++j)
                {
                {
                    if (i < logging.size() && j < logging[i].size())
                    std::cout << get_visualization(logging[i][j]);
                    }
                    else
                    {
                        std::cout << get_visualization(LogStatus::None);
                    }
                }
                std::cout << std::endl;
                std::cout << std::string(10, ' ');
                for (size_t j = 0; j < time_to_visualize; ++j)
                {
                    std::cout << get_visualization(tasks[i], j);
                }
                std::cout << std::endl;
          }
          std::cout << std::string(10, ' ') << std::string(time_to_visualize, '-') << "> " <<std::endl;
        }
        void print_analytics(const Tasks& tasks)
        {
            bool is_ok = true;
            for (size_t i = 0; i < tasks.size() – 1; ++i)
            {
```

```
                const auto& task = tasks[i];
            std::cout << "Task #" << i + 1 << ": r=" << task.r << " e=" << task.e << " d=" <<task.d << " Actual:   "
<< task.start << ".." << task.end << " ";
                if (task.end <= task.d)
            {
            }
                else
                {
                }
    std::cout << "OK";
    std::cout << "OVERTIME=" << task.end – task.d;is_ok = false;
                std::cout << std::endl;
            }
            std::cout << (is_ok ? "SUCCESS" : "FAILED") << std::endl;
      }
      int main()
      {
            int period;
            std::cout  <<  "Input  period  T:  ";
    std::cin >> period;
            size_t tasks_n;
            std::cout  <<  "Input  tasks  num:  ";
    std::cin >> tasks_n;
            Tasks tasks;
            for (size_t i = 0; i < tasks_n; ++i)
            {
                int r, e, d;
                std::cout << "Input r, e, d for task #" << i + 1 << ": ";std::cin >> r
            >> e >> d;
                tasks.emplace_back(r, e, d);
            }
            std::string planner_type;
            std::cout << "Input planner type (EDFs | None): ";std::cin
    >> planner_type;
            //dummy task:
            tasks.emplace_back(0, 1, std::numeric_limits<int>::max());
            PetriModel model(tasks, period);
            auto planner = create_planner(tasks, planner_type);
            Logging logging(tasks.size(), LogLine(period + 1, LogStatus::None));for (int tick = 0;
    tick < period; ++tick)
            {
                const size_t task_id = planner->getNextTask(tick);
                if (tasks[task_id].start == -1)
                {
                    tasks[task_id].start  =  tick;  logging[task_id][tick]  =
                    LogStatus::Started;
                }
                else
                {
                     logging[task_id][tick] = LogStatus::Processing;
                 }
                 bool done = model.doTask(task_id);
                 if (done && task_id != tasks.size() – 1)
                {
                     tasks[task_id].end = tick + 1; logging[task_id][tick + 1] =
                LogStatus::Finished;
```

```
            }
        }
        draw_diagram(logging, tasks, tasks.size() – 1, period + 1);
        print_analytics(tasks);
        std::cout << "Unused time: " << model.getTaskTime(tasks.size() – 1) << std::endl;
    }
```

Результати:

```
    Input period T: 42
Input tasks num: 4
    Input r, e, d for task #1: 0 12 30
    Input r, e, d for task #2: 2 14 34
    Input r, e, d for task #3: 5 6 42
    Input r, e, d for task #4: 4 10 16
    Input planner type (EDFs | None): None
                    0          10          20          30          40
                    |          |           |           |           |
      Tasc 1   o**********|**, , , , , , , |, ,  , , , , , , , x , , , , , , , , , |, , 
      Tasc 2   |,o , , , , , , , |, *********| ******, , , , |, , , , , , , , , |, , 
       Tasc 3  |, , , , o , , , ,|, , , , , , , , |, , , , , ******* , , , , , , , , |.x
      Tasc 4   |, , , o , , ,  , ,|, , , , , , , , |, , , , , , , , , , |, , . **********
     Task #1: r=0 e=12 d=30 Actual: 0..12 OK
   Task #2: r=2 e=14 d=34 Actual: 12..26 OK
   Task #3: r=5 e=6 d=42 Actual: 26..32 OK
   Task #4: r=4 e=10 d=16 Actual: 32..42 OVERTIME=26FAILED
    Unused time: 0

    Input planner type (EDFs | None): EDFs
                    0          10          20          30          40
                    |          |           |           |           |
      Tasc 1   o****, , , , , |, , , ******|** , , , , , , , . |. , , , , , , , , |, , .
      Tasc 2   |,o , , , , , , |, , , , , , , , , , | , ********|******* , , , , |, , 
      Tasc 3   |, , , , o , , , ,|, , , , , , , , , ,|, , , , , , , , , , |, , , , , *******
      Tasc 4   |, , , o*****|****, x , , , ,|, ,  , , , , , , , |, , , , , , , , , , |, , 
           Task #1: r=0 e=12 d=30 Actual: 0..22 OK
                Task #2: r=2 e=14 d=34 Actual: 22..36 OVERTIME=2
                Task #3: r=5 e=6 d=42 Actual: 36..42 OK
                Task #4: r=4 e=10 d=16 Actual: 4..14 OK
                FAILED
          Unused time: 0
```

Висновки

Запропонована програмна модель виконання задач реального часу гарантовано визначає можливість виконання вхідного набору задач за вибраний період T. Доцільність використання програмної моделі виконання задач доведена за допомогою тестування на різних наборах даних. При визначенні періоду виконання відбувається перехід від звичного часу до квантового часу. Розміри квантування визначають точність моделювання. Послідовно модель запускалася з різними типами планувальників. Результатом кожного запуску була генерація трьох таблиць:

– таблиці розкладу планувальника;
– таблиці виконання задач;
– таблиці завершених задач.

Остання таблиця наводить квантові моменти завершення задач та їх крайні терміни. Порівняння цих двох моментів уможливлює отримати точну відповідь, чи може задача бути виконана у відведені їй терміни.

Отже, можна стверджувати, що модель виконання задач за допомогою сіток Петрі є ефективним інструментом при розробці систем реального часу. Гарантованість отримання результату моделювання була доведена за допомогою тестування та аналізу різних наборів задач систем.

Список літератури

1. Зайцев В. Г., Цибаєв Є. І. Комп'ютерні системи реального часу: навчальний посібник. Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського". Київ, 2019. URL: https:// ela.kpi.ua/handle//123456789/29604
2. Зайцев В. Г., Дробязко І. П. Операційні системи: навчальний посібник / Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського": Київ, 2019. URL: https:// ela.kpi.ua/handle//123456789/2
3. Зайцев В. Г., Цибаєв Є. І. Модель оцінки часових характеристик у комп'ютерних системах реального часу з використанням сіток Петрі. *Управління розвитком складних систем*. 2019. № 40. С. 76 – 86; dx.doi.org\10.6084/m9.figshare.11969013.
4. Golubev A. S. (2010). Real-time systems: lecture notes. Vladim. state un-t; comp. Vladimir: Publishing house Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute. Electronic resource of KPI named after Igor Sikorsky: https: // ela.kpi.ua/handle//123456789/29600.
5. Baker T. Multiprocessors EDF and Deadline Monotonic Shedulability Analysis. *Proceeding of 24 IEEE Real – Time Systems Symposium*, 2003, p. 120–129.
6. Andersen B., Daruah S., Jonson J. Static – Priority Shedulings on Microprocessors. *Proccedings of 22 IEEE Real – Time System Symposium*. 2001, p. 193 – 202.
7. Ferrari A. D. Real – Time Shtduling Algorithms // Dr. Dobb's Jornal. 1994, N12, p. 60 – 66.
8. Стеценко І. В., Бойко О. В. Система імітаційного моделювання засобами сіток Петрі. *Математичні машини і системи*. 2009. № 1. С. 117–124.

Стаття надійшла до редколегії 20.05.2023

Zaitsev Vladimir
DSc (Eng.), Professor, Professor of the Department of System Programming and Specialized Computer Systems, Kyiv Polytechnic Institute, *https://orcid.org/0000-0001-9548-1959*
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv
Tsybaev Evgeniy
PhD (Eng.), Department of System Programming and Specialized Computer Systems, Kyiv Polytechnic Institute, *https://orcid.org/0000-0002-9115-2346*
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv

ESTIMATION OF TIMING CHARACTERISTICS IN REAL-TIME COMPUTER SYSTEMS USING PETRI NETS

Abstract. *The work is devoted to the problem of determining the time characteristics of tasks in real-time systems, the success of which depends not only on their logical correctness, but also on the time for which they receive the result. Determining such time characteristics of the system at the design stage is a rather difficult problem. Its solution is currently based on the use of two main methods: theoretical calculations related to the calculation of the so-called feasibility criteria and modeling of the system's operation on models. Among the models, statistical models of mass service systems are the most widespread. However, in both the first and second options, it is impossible to obtain a guaranteed result, which significantly complicates the design process. Recently, it has been proposed to use the methods of researching models based on the use of the Petri net apparatus. The paper proposes a method for estimating the time characteristics of tasks in real-time systems by means of data analysis. Obtained by modeling the distribution of processor time between tasks according to the selected algorithms of the scheduler by using a model of the software system in the form of a Petri net. The paper proposes a method for estimating the time characteristics of tasks in real-time systems by means of data analysis. Obtained by modeling the distribution of processor time between tasks according to the selected algorithms of the scheduler by using a model of the software system in the form of a Petri net. The method guarantees obtaining the time characteristics of the tasks when choosing specific types of processor and scheduler, which is required to start the technical design of the real-time system. The use of the proposed mathematical model and a package of application programs allow, at the early stages of the development of a real-time system, to determine the real terms of tasks and choose the optimal option for the implementation of technical and lost support. The proposed mathematical models are based on determining the time characteristics of program execution by modeling the distribution of processor time between tasks, subject to the prior selection of planning algorithms and the characteristics of a complex of technical means. The research is based on the use of simulated statistical models and Petri net models, The use of the proposed tools allows you to significantly reduce the time of analysis of possible options for*

real-time system implementation, increases the quality of the project and significantly reduces the overall time and cost of the entire development. The paper examines in detail the process of modeling a multitasking computer system using a Petri net device. A concrete example shows the simulation of the operation of a complex of programs, including the development of simulation algorithms, the listing of the simulation program, and the analysis of the obtained results.

Keywords: model; real-time task; Petri net

References

1. Zaitsev, V. G., Tsybaev, E. I. (2019). Real-time computer systems: a textbook. National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute": Kyiv. Igor Sikorsky KPI Electronic Resource: https:// ela.kpi.ua/handle//123456789/29604.

2. Zaitsev, V. G., Drobyazko, I. P. (2019). Operating Systems. Real-time computer systems: a textbook. National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute": Kyiv. Igor Sikorsky KPI Electronic Resource: https://ela.kpi.ua/handle//123456789/29604.

3. Zaitsev, V. G. Tsybaev, E. I. (2019). A model for estimating time characteristics in real-time computer systems using Petri nets. *Management of development of complex systems,* 40, 76–86; dx.doi.org\10.6084/m9.figshare.11969013.

4. Golubev, A. S. (2010). Real-time systems: lecture notes. Vladim. state un-t; comp. Vladimir: Publishing house Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute. Electronic resource of KPI named after Igor Sikorsky: https: // ela.kpi.ua/handle//123456789/29600.

5. Baker, T. (2003). Multiprocessors EDF and Deadline Monotonic Shedulability Analysis. *Proceeding of 24 IEEE Real – Time Systems Symposium,* Pp. 120–129.

6. Andersen, B., Daruah, S., Jonson, J. (2001). Static– Priority Shedulings on Microprocessors. *Proccedings of 22 IEEE Real – Time System Symposium,* Pp. 193–202.

7. Ferrari, A. D. (1994). Real – Time Sheduling Algorithms. *Dr. Dobb's Journal,* 12, 60–66.

8. Stetsenko, I. V., Boyko, O. V. (2009). Imitation modeling system using Petri nets. *Mathematical machines and systems,* 1, 117–124.

Посилання на публікацію

APA Zaitsev, Vladimi, & Tsybaev, Evgeniy. (2023). Estimation of timing characteristics in real-time computer systems using Petri nets. *Management of Development of Complex Systems*, 54, 48–62, dx.doi.org\10.32347/2412-9933.2023.54.48-62.

ДСТУ Зайцев В. Г., Цибаєв Є. І. Оцінка часових характеристик у комп'ютерних системах реального часу з використанням сіток Петрі. *Управління розвитком складних систем.* Київ, 2023. № 54. С. 48 – 62, dx.doi.org\10.32347/2412-9933.2023.54.48-62.