Харківський національний університет радіоелектроніки

Міністерство освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

ГОЛЬДІНЕР ДЕНИС ІГОРОВИЧ

УДК 004.94 : 519.872.3

ДИСЕРТАЦІЯ

МЕТОД ТА МОДЕЛІ ОПТИМІЗАЦІЇ СИСТЕМ МАСОВОГО ОБСЛУГОВУВАННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ ПРІОРИТЕЗАЦІЇ ВХІДНОГО ПОТОКУ ЗАЯВОК

124 Системний аналіз
12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Д. І. Гольдінер

Науковий керівник:
Матвієнко Ольга Іванівна, кандидат технічних наук, доцент

Харків – 2024

АНОТАЦІЯ

Гольдінер Д.І. Метод та моделі оптимізації систем масового обслуговування з використанням технології пріоритезації вхідного потоку заявок. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 124 Системний аналіз. – Харківський національний університет радіоелектроніки, Міністерство освіти і науки України, Харків, 2024.

Метою дисертаційної роботи є розробка комп'ютерної та інформаційної моделей аналізу процесів масового обслуговування й удосконалення методу та моделей оптимізації систем масового обслуговування з використанням технології пріоритезації вхідного потоку заявок.

Об'єктом дослідження є системи масового обслуговування. Предметом дослідження є методи оптимізації систем масового обслуговування та моделі програмної симуляції поведінки складних систем.

У сучасному світі навколо є безліч різноманітних процесів, що включають в себе обробку потоків вхідних даних. Значне поширення мають прикладні задачі, що можуть бути описані та оптимізовані за допомогою теорії масового обслуговування, наприклад, це задачі:

– управління запасами;

– управління людськими ресурсами;

– оптимізації використання обладнання на виробництвах;

– керування взаємодією з клієнтами;

– обробки черг замовлень в торгівлі;

– побудови автоматизації виробництв;

– масштабування систем надання послуг;

– забезпечення збільшення пропускної здатності трафіка в цифрових мережах.

До математичних моделей можуть бути зведені задачі, що виникають в

інформаційно-керуючих системах, системах контролю та управління процесами, телекомунікаційних технологіях та мережах, а також у менеджменті. До кожної задачі можна підібрати відповідний тип системи масового обслуговування (СМО), який найкраще описуватиме специфіку фактичного процесу.

Однією з розповсюджених причин моделювання складних систем є потреба у збільшенні їх пропускної здатності. Однак ця задача суттєво ускладняється обмеженістю ресурсів на масштабування. Відповідно, гостро постає проблема раціональнішого використання наявних ресурсів та отримання максимальної користі від чинної конфігурації системи.

Водночас, для отримання швидких та інформативних результатів експерименту, виникає потреба в побудові програмного забезпечення, що буде максимально наближено до реального модельованого процесу. При цьому, при моделюванні багатоканальних СМО, значну роль грає спосіб, в який організуються паралельні обчислення.

Дисертація складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та трьох додатків.

У вступі подано загальну характеристику дисертації, обґрунтовано актуальність теми дослідження, сформульовано мету, завдання, об’єкт, предмет, розкрито наукову новизну, практичну значущість одержаних результатів, визначено особистий внесок автора, наведено дані про апробацію та впровадження результатів дослідження, надано інформацію про кількість публікацій за темою роботи.

У першому розділі проведено огляд літературних джерел, у яких вивчаються питання моделювання та оптимізації систем масового обслуговування, моделей взаємодіючого співвиконання (concurrency) та їх практичного застосування, а також розробки архітектури програмного забезпечення. У результаті аналізу літературних джерел з’ясовано, що на даний час мало дослідженими є системи масового обслуговування зі складними системами пріоритетів, а також застосування методів взаємодіючого

співвиконання до програмного моделювання СМО. Відповідно до цього сформульовано постановку задач дослідження.

Також у першому розділі приділено увагу аналізу постановок задач моделювання багатоканальних систем масового обслуговування з обмеженою чергою та відмовами. Досліджено математичні моделі аналізу поведінки досліджуваних систем. Наведено формули для розрахунку ключових характеристик системи, таких як ймовірність відмови новоприбулій заявці через відсутність вільних місць очікування. Сформульовано задачі дослідження мінімізації ймовірності відмови для систем з обмеженою чергою.

Другий розділ присвячено методам оптимізації багатоканальних систем масового обслуговування з обмеженою чергою та відмовами. Основну увагу приділено застосуванню пріоритезації до вхідного потоку заявок. Запропоновано новий метод, заснований на категоризації вимог згідно з часом необхідним на їх опрацювання із подальшим динамічним балансуванням пріоритетів. При цьому місця у загальній черзі перерозподіляються під відповідні класи заявок. Для визначення складності заявки запропоновано застосування кількох моделей. Було детально проаналізовано математичну модель для спрощеної системи: одноканальної (з необмеженою чергою) та з двома класами розмірів заявок. Виведено рівняння для знаходження граничних значень політик балансу. Розраховано умови масштабування та описано сприятливі умови для застосування даного методу. Велику увагу приділено пошуку порогового значення, за якого необхідно змінювати категорію для заявки. Також розглянуто можливості застосування методів штучного інтелекту до оцінки складності заявки.

У третьому розділі сформовано вимоги до програмного забезпечення для моделювання систем масового обслуговування, а також проведено огляд наявних на даний момент рішень. Обґрунтовано доцільність розвитку даного напрямку для досягнення кращої суміжності симуляції та досліджуваного процесу, а також для підвищення ефективності використання системних ресурсів. Було приділено особливу увагу процесу планування потоків

виконання та механізмам синхронізації. Проведено категоризацію мов програмування за ознаками паралелізму, що вони реалізують. Було детально розглянуто суть підходу взаємодіючого співвиконання та запропоновано застосування взаємодіючого співвиконання для моделювання систем масового обслуговування. Обрано мову програмування Go як таку, що найкраще з сучасних інструментів, які доступні розробникам, передає ідею взаємодіючого співвиконання, та розібрано особливості технології.

Четвертий розділ присвячено побудові архітектури та імплементації програмного забезпечення для моделювання систем масового обслуговування. Для обґрунтованого прийняття рішення щодо різновиду архітектури було проведено аналіз всіх розповсюджених підходів. Розроблено дизайн архітектури, складено структурну та компонентну діаграми, а також схеми послідовної абстракції C4. Детально розглянуто реалізацію програмного забезпечення мовою програмування Go із детальним розбором прийнятих технічних рішень. Було розібрано механізми синхронізації у мові програмування Go, надано структуру незалежних процесів та послідовність комунікації, що виникає при моделюванні систем масового обслуговування.

У висновках підсумовуються всі здобуті результати та висвітлюється доцільність застосування запропонованих методу та моделей для оптимізації систем масового обслуговування.

Додатки містять відомості про публікації основних результатів, отриманих у дисертації, акт впровадження та програмну реалізацію.

Проведені у дисертаційній роботі дослідження дозволили отримати такі нові наукові результати:

– вперше запропоновано інформаційну та комп’ютерну моделі аналізу процесів масового обслуговування із застосуванням взаємодіючого співвиконання (concurrency): особливістю інформаційної моделі є розбиття всього процесу обробки на прості асинхронні операції, а особливістю комп’ютерної моделі є максимально ефективне функціонування багатопотоковості;

– набула подальшого розвитку модель розбиття загального вхідного потоку заявок по категоріях за розміром в частині застосування поділу черги очікування за квотами;

– удосконалено метод оптимізації систем масового обслуговування в частині пріоритезації менших задач SJF (Shortest Job First), завдяки використанню комбінованої системи пріоритетів.

Одержані в дисертації результати можуть бути використані під час моделювання поведінки процесів, що можуть бути описані методами теорії масового обслуговування, а також для подальших досліджень методів та моделей підвищення ефективності використання ресурсів у системах масового обслуговування.

Розроблена реалізація автоматизації взаємодіючого співвиконання побудована на абстракціях та винесена в окремий пакет, її можна застосувати у будь-якій програмі написаній мовою Go. В такий спосіб програмне забезпечення, що розроблене в рамках роботи, набуває практичної цінності для різних галузей науки та техніки, де є потреба в застосуванні паралельних або асинхронних обчислень.

Отримані результати підтверджується їх впровадженням в освітньому процесі Харківського національного університету радіоелектроніки при викладанні дисципліни «Програмування» для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 124 Системний аналіз, у курсовому проєктуванні з цієї дисципліни та при підготовці кваліфікаційних робіт здобувачами першого (бакалаврського) та другого (магістерського) рівнів вищої освіти.

Основні результати, що становлять зміст дисертації, опубліковані у 5 наукових роботах: 3 статті, опубліковані у виданнях, зазначених в переліку наукових фахових видань України (категорія Б) за спеціальністю 124 Системний аналіз, та 2 тези доповідей, опублікованих у матеріалах міжнародних наукових конференцій.

Ключові слова: алгоритм, математичне моделювання, системи масового

обслуговування, пріоритети, оптимізація, взаємодіюче співвиконання, паралелізм, мова програмування Go, архітектура програмного забезпечення, штучний інтелект, машинне навчання, штучна нейронна мережа, задача категоризації, марковські процеси, патерни проєктування, інтенсивність відмов, комп'ютерне моделювання.

Список публікацій здобувача

Наукові публікації, в яких опубліковані
основні наукові результати дисертації

1. Гольдінер Д. І. Застосування мови програмування GO для моделювання процесів масового обслуговування. *Сучасний стан наукових досліджень та технологій в промисловості*. 2024. № 2(28). С. 65–75. DOI: 10.30837/2522-9818.2024.2.065 [Входить до міжнародної наукометричної бази Google Scholar.]

2. Гольдінер Д. І. Розробка архітектури програмного забезпечення для моделювання систем масового обслуговування під імплементацію мовою програмування GO. *Вісник Національного технічного університету «ХПІ». Серія: Системний аналіз, управління та інформаційні технології*. 2024. № 1 (11). С. 85–90. DOI: 10.20998/2079-0023.2024.01.14 [Входить до міжнародної наукометричної бази Google Scholar.]

3. Гольдінер Д. І., Матвієнко О. І. Зменшення ймовірності відмови в системах масового обслуговування з обмеженою чергою із застосуванням пріоритезації за розміром та штучного інтелекту. *Біоніка інтелекту*. 2024. № 1 (100). С. 36–42. DOI: 10.30837/bi.2024.1(100).05 [Входить до міжнародної наукометричної бази Google Scholar.]

Наукові праці, які засвічують
апробацію матеріалів дисертації

4. Goldiner D., Tevyashev A. System Analysis of the Parallel Execution Problem. *Інформаційні системи та технології – ICT-2019* : матеріали 8-ї

Міжнародної науково-технічної конференції, Коблеве-Харків, Україна, 9-14 вересня 2019 р. Харків : ХНУРЕ, 2019. С. 210–213. [Входить до міжнародної наукометричної бази Google Scholar.]

5. Goldiner D. Rejection probability reduction in queueing systems with limited queue using size-based prioritization. *Perspectives of Contemporary Science: Theory and Practic*e : Proceedings of VII International Scientific and Practical Conference, Lviv, Ukraine, 19-21 August 2024. Lviv, Ukraine. 2024. P. 256–262.

SUMMARY

Goldiner D.I. The method and models for optimizing queueing systems using input flow prioritization technology. – Qualification scientific study with manuscript copyright.

A dissertation for the degree of Doctor of Philosophy in specialty 124 System Analysis. – Kharkiv National University of Radio Electronics, Ministry of Education and Science of Ukraine, Kharkiv, 2024.

The purpose of the dissertation is to develop computer and information models for the analysis of queueing processes and to improve the method and models for optimizing queueing systems using input flow prioritization technology.

The object of the research is queueing systems. The subject of the research is the optimization methods for queueing systems and the models for software simulation of complex system behavior.

In the modern world, there are numerous diverse processes involving the processing input data streams. Most of applied tasks that can be described and optimized using the queuing theory, for example, these are the tasks:

– inventory management;

– human resource management;

– optimization of equipment use in production;

– management of interaction with customers;

– processing order queues in trade;

– building automation of production;

– scaling service delivery systems;

– ensuring increased traffic capacity in digital networks.

Mathematical models can be applied to solve problems that arise in the following areas: information management systems, process control and management systems, telecommunications technologies, networks etc. For each problem there is an appropriate type of queuing system (QMS), which will best describe the specifics of

the actual process.

One of the common reasons for modeling complex systems is the need to increase throughput. However, this task is significantly complicated by the limited resources available for scaling. Accordingly, the problem of more rational use of available resources and obtaining maximum benefit from the current system configuration is acute.

At the same time, to obtain fast and informative experimental results, there is a need to build software that will be as close as possible to the real simulated process. At the same time, when modeling multi-channel QMS, the way in which parallel calculations are organized plays a significant role.

The dissertation consists of an introduction, four chapters, conclusions, a list of references, and three appendices.

The introduction provides a general description of the dissertation, justifies the relevance of the research topic, formulates the goal, objectives, object, subject, reveals the scientific novelty, practical significance of the results obtained, identifies the author's personal contribution, provides data on the testing and implementation of the research results, and provides information on the number of publications on the topic of the work.

The first chapter reviews the literary sources on modeling and optimization of queuing systems, concurrency models and their practical applications, as well as software architecture development. As a result of the analysis of the literature, it was found that queuing systems with complex priority systems, as well as the application of concurrency methods to software modeling of QMS, are currently poorly studied. Accordingly, the research objectives are formulated. It also focuses on the analysis of the problem statements for modeling multi-channel mass service systems with a limited queue and failures. It explores mathematical models for analyzing the behavior of the systems under study. Formulas are given for calculating key system characteristics, such as the probability of rejection of a newly arrived application due to the lack of free waiting places. Research problems are formulated to minimize the probability of rejection for systems with a limited queue.

The second chapter addresses methods for optimizing multi-channel queueing systems with a limited queue and rejections. The main attention is paid to the application of prioritization to the incoming flow of requests. A new method is proposed, based on the categorization of requirements according to the time required for their processing with subsequent dynamic balancing of priorities. In this case, places in the general queue are redistributed to the corresponding classes of requests. To determine the complexity of the request, the use of several models is proposed. A mathematical model for a simplified system was analyzed in detail: single-channel (with an unlimited queue) and with two classes of request sizes. An equation for finding the limit values of balance policies was derived. Scaling conditions were calculated and favorable conditions for the application of this method were described. Much attention was paid to finding the threshold value at which it is necessary to change the category for the request. The possibilities of applying artificial intelligence methods to assessing the complexity of the request were also considered.

The third chapter forms the requirements for software for modeling queueing systems, and also reviews the currently available solutions. The feasibility of developing this direction is justified in order to achieve better contiguity of simulation and the studied process, as well as to increase the efficiency of using system resources. Special attention was paid to the process of scheduling execution flows and synchronization mechanisms. Programming languages are categorized according to the features of parallelism that they implement. The essence of the concurrency approach is considered in detail and the application of concurrency for modeling queueing systems is proposed. The Go programming language is chosen as the one that best conveys the idea of concurrency with modern tools available to developers, and the features of the technology are analyzed.

The fourth chapter is dedicated to the construction of the architecture and implementation of software for modeling queue management systems. In order to make a well-founded decision on the type of architecture, an analysis of all common approaches was conducted. The architecture design was developed, structural and component diagrams were drawn up, as well as sequential abstraction schemes C4. The

implementation of the software in the Go programming language was considered in detail with a detailed analysis of the technical solutions adopted. The synchronization mechanisms in the Go programming language were analyzed, the structure of independent processes and the sequence of communication that occurs when modeling queue management systems were provided.

The conclusions summarize all the results obtained and highlight the feasibility of applying the proposed methods and models for optimizing queue management systems.

The appendices contain information about the publications of the main results obtained in the dissertation, an implementation act, and program implementation details.

The research conducted in the dissertation allowed obtaining several new scientific results:

– for the first time, an information and computer model for analyzing queueing processes using concurrency was proposed: the main benefit of the information model is the division of the entire processing process into simple asynchronous operations, and a feature of the computer model is the maximum efficient functioning of multi-threading;

– the model for dividing the total incoming flow of applications into categories by size using the division of the waiting queue by quotas was further developed;

– the method for optimizing queueing systems in terms of prioritizing smaller tasks SJF (Shortest Job First) was improved, thanks to the use of a combined priority system.

The results obtained in the dissertation can be applied when modeling the behavior of processes that can be described by the methods of the theory of queuing, as well as for further research into methods for increasing the efficiency of resource use in queuing systems.

The developed implementation of automation of interactive co-execution is built on abstractions and is placed in a separate package, it can be used in any program written in the Go language. In this way, the software developed as part of

the work acquires practical value for various branches of science, where there is a need to use parallel or asynchronous computing.

Confirmed results of their implementation were obtained at the educational Kharkiv National University of Radio Electronics when teaching the discipline «Programming» for applicants of the first (bachelor's) level of higher education in the specialty 124 System Analysis, in course design in this discipline and in the preparation of qualification works for applicants of the first (bachelor's) and second (master's) levels of higher education.

The main results constituting the content of the dissertation have been published in 5 scientific works: 3 articles published in publications listed in the list of scientific professional publications of Ukraine (category B) in the specialty 124 System Analysis, and 2 abstracts of reports published in the materials of international scientific conferences.

Keywords: algorithm, mathematical modeling, queuing systems, priorities, optimization, concurrency, parallelism, Go programming language, software architecture, artificial intelligence, machine learning, artificial neural network, categorization problem, Markov processes, design patterns, failure rate, computer simulation.

List of applicant's published works

Scientific publications containing the main scientific results of the dissertation

1. Гольдінер Д. І. Застосування мови програмування GO для моделювання процесів масового обслуговування. *Сучасний стан наукових досліджень та технологій в промисловості*. 2024. № 2(28). С. 65–75. DOI: 10.30837/2522-9818.2024.2.065 [Входить до міжнародної наукометричної бази Google Scholar.]

2. Гольдінер Д. І. Розробка архітектури програмного забезпечення для моделювання систем масового обслуговування під імплементацію мовою програмування GO. *Вісник Національного технічного університету «ХПІ». Серія: Системний аналіз, управління та інформаційні технології*. 2024. № 1 (11).

С. 85–90. DOI: 10.20998/2079-0023.2024.01.14 [Входить до міжнародної наукометричної бази Google Scholar.]

3. Гольдінер Д. І., Матвієнко О. І. Зменшення ймовірності відмови в системах масового обслуговування з обмеженою чергою із застосуванням пріоритезації за розміром та штучного інтелекту. *Біоніка інтелекту*. 2024. № 1 (100). С. 36–42. DOI: 10.30837/bi.2024.1(100).05 [Входить до міжнародної наукометричної бази Google Scholar.]

Scientific papers validating the testing of the materials presented in the dissertation

4. Goldiner D., Tevyashev A. System Analysis of the Parallel Execution Problem. *Інформаційні системи та технології – ICT-2019* : матеріали 8-ї Міжнародної науково-технічної конференції, Коблеве-Харків, Україна, 9-14 вересня 2019 р. Харків : ХНУРЕ, 2019. С. 210–213. [Входить до міжнародної наукометричної бази Google Scholar.]

5. Goldiner D. Rejection probability reduction in queueing systems with limited queue using size-based prioritization. *Perspectives of Contemporary Science: Theory and Practic*e : Proceedings of VII International Scientific and Practical Conference, Lviv, Ukraine, 19-21 August 2024. Lviv, Ukraine. 2024. P. 256–262.

ЗМІСТ

ВСТУП

Актуальність теми. Теорія масового обслуговування залишається актуальною у сучасному світі завдяки стрімкому розвитку інформаційних технологій, інфраструктури обслуговування та збільшення вимог до швидкості й ефективності обробки інформації.

Оптимізація систем масового обслуговування відіграє важливу роль у підвищенні ефективності та продуктивності обслуговуючих систем. Вона дозволяє мінімізувати витрати на інфраструктуру, зменшити час очікування обробки заявок та рівень відмов, що особливо важливо для великих підприємств і сервісних організацій з високими вимогами до якості обслуговування. Оптимізація також сприяє раціональному використанню ресурсів та зниженню енергоспоживання, що робить цей процес ключовим для побудови стійких при масштабуванні систем.

Пріоритезація вхідного потоку заявок відіграє важливу роль у розвитку високо навантажених систем масового обслуговування, оскільки дозволяє ефективніше розподіляти наявні ресурси та стабільно обслуговувати найважливіші запити, за умови наявності обмеження на горизонтальне масштабування. У системах, де кількість заявок перевищує пропускну здатність системи, пріоритезація забезпечує гнучке управління потоками та дозволяє вирішувати, які заявки мають отримати обслуговування першочергово.

Системи пріоритезації також сприяють вдосконаленню методів управління операційними процесами, забезпечуючи максимальну продуктивність навіть за обмежених ресурсів. Це особливо важливо для складних динамічних систем, де ефективне управління пріоритетами допомагає швидко адаптуватися до зміни умов.

За умов високого навантаження, коли кількість запитів на обслуговування перевищує максимальну пропускну здатність системи, черга очікування буде заповнюватись заявками. Якщо піковий попит не спадає певний час, він

призводить до переповнення черги, внаслідок чого стає неможливим прийняття нових запитів, і відбувається відмова в обслуговуванні.

Одним із підходів до вирішення проблеми є впровадження адаптивного керування чергою, коли система динамічно змінює параметри обслуговування залежно від поточного навантаження, що дозволяє ефективніше розподіляти потік заявок, тим самим зменшуючи ризик переповнення. Такий підхід підвищує стійкість системи та дозволяє ефективніше реагувати на зміни в обсягах заявок, що є особливо важливим у сучасних умовах швидкозмінних ринкових і технологічних вимог.

Однією з головних проблем, що постають при моделюванні подібних систем масового обслуговування, є складність врахування випадкових та змінних факторів, таких як нерегулярний потік заявок, зміна потреб користувачів, обмеженість ресурсів, задіяних в обслуговуванні, та неоднорідність часу обслуговування, що ускладнює розрахунок точних показників продуктивності.

Таким чином, впровадження систем пріоритезації є не лише необхідним для підвищення оперативної ефективності, але й ключовим фактором для економічної життєздатності компаній у довгостроковій перспективі.

Великий внесок у розвиток сучасних підходів до вирішення проблеми моделювання та оптимізація систем масового обслуговування зроблено у роботах вчених L. Kleinrock [1], L. Kleinrock, R. Gail [2], R. B. Cooper [3], A. O. Allen [4], D. Gross, C. M Harris [5], J. F. C. Kingman [6], M. Sommereder [7], J. M. Matzka [10], Y. Chen, J. Dong [11], L. E. Schrage and L. W. Miler [15], D. R. Cox and W. L. Smith [29], О. Я. Хінчина [37], D. G. Kendall[39], M. I. Reiman [51], W. Whitt [51] та ін.

Незважаючи на значні успіхи в розробці методів оптимізації систем масового обслуговування було виявлено протиріччя, пов’язане з тим, що відсутні дослідження впливу поєднання розбиття вхідного потоку заявок на категорії за розміром (з подальшою пріоритезацією менших за часом обробки) та гнучкого перерозподілення місць у черзі очікування під класи вимог на

ймовірність відмови у багатоканальних системах з обмеженою чергою. Також відсутні дослідження застосування моделей взаємодіючого співвиконання до програмного моделювання систем масового обслуговування. Отже, розробка нових та вдосконалення існуючих методів та моделей, що підвищують продуктивність систем масового обслуговування з обмеженою чергою за рахунок зменшення ймовірності відмови, є актуальною науковою задачею.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційна робота виконувалася в період з 2017 р. по 2024 р. відповідно до плану науково-дослідних робіт кафедри прикладної математики Харківського національного університету радіоелектроніки.

Мета і завдання дослідження. *Метою* дисертаційної роботи є розробка комп'ютерної та інформаційної моделей аналізу процесів масового обслуговування й удосконалення методу та моделей оптимізації систем масового обслуговування з використанням технології пріоритезації вхідного потоку заявок.

Для досягнення поставленої мети необхідно виконати такі завдання:

– дослідити задачу, до якої застосовуватимуться метод та моделі оптимізації, розрахувати для неї базові характеристики;

– дослідити існуючі методи оптимізації систем масового обслуговування;

– удосконалити метод оптимізації систем масового обслуговування в частині пріоритезації менших задач SJF (Shortest Job First), завдяки використанню комбінованої системи пріоритетів;

– удосконалити модель розбиття загального вхідного потоку заявок по категоріях за розміром в частині застосування поділу черги очікування за квотами;

– дослідити паралелізм та взаємодіюче співвиконання (concurrency) у комп'ютерному програмуванні;

– дослідити можливості сучасних мов програмування щодо реалізації взаємодіючого співвиконання та обрати технологію для розробки комп'ютерної програми;

– запропонувати інформаційну та комп’ютерну моделі аналізу процесів масового обслуговування із застосуванням взаємодіючого співвиконання;

– розробити програмне забезпечення для аналізу процесів масового обслуговування.

Об’єктом дослідження є системи масового обслуговування.

Предметом дослідження є методи оптимізації систем масового обслуговування та моделі програмної симуляції поведінки складних систем.

Методи дослідження. У роботі використовуються методи моделювання систем масового обслуговування; диференціальний метод дослідження систем масового обслуговування; пріоритезація вхідного потоку заявок; взаємодіючого співвиконання; паралельні обчислення; мова програмування Go; патерни проєктування архітектури.

Наукова новизна отриманих результатів. У результаті виконання дисертаційного дослідження розроблено сучасний ефективний метод оптимізації систем масового обслуговування з використанням технології пріоритезації вхідного потоку заявок, а також покращених моделей їх програмної симуляції. При цьому отримано такі нові наукові результати:

– вперше запропоновано інформаційну та комп’ютерну моделі аналізу процесів масового обслуговування із застосуванням взаємодіючого співвиконання (concurrency): особливістю інформаційної моделі є розбиття всього процесу обробки на прості асинхронні операції, а особливістю комп’ютерної моделі є максимально ефективне функціонування багатопотоковості;

– набула подальшого розвитку модель розбиття загального вхідного потоку заявок по категоріях за розміром в частині застосування поділу черги очікування за квотами;

– удосконалено метод оптимізації систем масового обслуговування в частині пріоритезації менших задач SJF (Shortest Job First), завдяки використанню комбінованої системи пріоритетів.

Теоретичне та практичне значення одержаних результатів. Одержані

в дисертації результати можуть бути використані під час моделювання поведінки процесів, що описуються методами теорії масового обслуговування, а також для подальших досліджень методів підвищення ефективності використання ресурсів у системах масового обслуговування.

Розроблена реалізація автоматизації взаємодіючого співвиконання побудована на абстракціях та винесена в окремий пакет, її можна застосувати у будь-якій програмі написаній мовою Go. В такий спосіб програмне забезпечення, що розроблене в рамках роботи, набуває практичної цінності для різних галузей науки, де є потреба в застосуванні паралельних або асинхронних обчислень.

Практичне значення отриманих у роботі результатів підтверджується їх впровадженням в освітньому процесі Харківського національного університету радіоелектроніки при викладанні дисципліни «Програмування» для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 124 Системний аналіз, у курсовому проєктуванні з цієї дисципліни та при підготовці кваліфікаційних робіт здобувачами першого (бакалаврського) та другого (магістерського) рівнів вищої освіти.

Особистий внесок здобувача. Основні результати дисертаційної роботи отримані здобувачем і опубліковані в роботах [79 – 83]. Роботи [79, 83] опубліковані у співавторстві. У тезах [79] здобувачем проведено аналіз проблеми паралелізму у сучасному світі технологій та розбір особливостей мови програмування Go при реалізації програмного забезпечення із застосуванням паралельних обчислень, а співавтору Тевяшеву А. Д. належить постановка задач дослідження. У роботі [83] здобувачем запропоновано підхід до розбиття вхідного потоку вимог на класи за розміром, запропоновано метод оцінки та динамічного балансування пріоритетів на основі поточного завантаження системи, а співавтору Матвієнко О.І. належить аналіз поточного стану предметної області та постановка задач дослідження. Роботи [80 – 82] опубліковано без співавторів.

Апробація результатів дисертації. Основні результати дисертаційної

роботи доповідались та обговорювались на:

– 8-й Міжнародній науково-технічній конференції «Інформаційні системи та технології» (Україна, Харків, 2019 р.);

– VII Міжнародній науково-технічній конференції «Perspectives of Contemporary Science: Theory and Practice» (Україна, Львів, 2024 р.);

– наукових семінарах кафедри прикладної математики Харківського національного університету радіоелектроніки (Харків, 2020, 2024 рр.).

Публікації. Матеріали дисертації викладені у 5 наукових роботах: 3 статті, опубліковані у виданнях, зазначених в переліку наукових фахових видань України (категорія Б) за спеціальністю 124 Системний аналіз, та 2 тези доповідей, опублікованих у матеріалах міжнародних наукових конференцій.

Структура та обсяг дисертацій. Дисертаційна робота є рукописом і складається зі вступу, чотирьох розділів, висновків, списку використаних джерел з 87 найменувань на 9 сторінках та трьох додатків на 20 сторінках, а також містить 56 рисунків та 6 таблиць. Загальний обсяг роботи складає 223 сторінки , включаючи 177 сторінок основного тексту.

1 АНАЛІЗ СУЧАСНОГО СТАНУ ЗАДАЧІ МОДЕЛЮВАННЯ СИСТЕМ МАСОВОГО ОБСЛУГОВУВАННЯ

1.1 Огляд науково-технічної літератури щодо методів моделювання та оптимізації систем масового обслуговування

Моделювання систем масового обслуговування (СМО) є важливою частиною математичних методів системного аналізу, що знаходить широке застосування в різних галузях, таких як промисловість, телекомунікації та обслуговування клієнтів. Література на цю тему охоплює різні аспекти, зокрема теоретичні основи, математичні моделі, методи оптимізації та дослідження поведінки складних систем при великому навантаженні.

Класичні роботи, такі як L. Kleinrock [1], L. Kleinrock, R. Gail [2], R. B. Cooper [3], A. O. Allen [4] визначають прості моделі черг та їхні основні властивості. Ці праці є основою для розуміння базових принципів функціонування СМО, зокрема одноканальних та багатоканальних моделей обслуговування. У поєднанні з монографією D. Gross, C. M. Harris [5] ці джерела надають систематизоване і ґрунтовне пояснення основних теорій та алгоритмів, які використовуються для розрахунку часу очікування, ймовірностей відмови та оптимізації потоку заявок.

Математичні підходи до моделювання та аналізу черг детально розглядаються також у роботах J. F. C. Kingman [6], M. Sommereder [7], G. Bolch, S. Greiner, H. Meer, K. S. Trivedi [8], де досліджуються пуассонівські процеси, які є основою для багатьох моделей систем. Їх праці допомагають зрозуміти, як випадковість і ймовірнісний розподіл впливають на поведінку системи при високому навантаженні та як використовувати ці знання для прогнозування поведінки реальних процесів.

У новіших публікаціях, таких як стаття L. Rece, S. Vlase, D. Ciuiu, G. Neculoiu, S. Mocanu, A. Modrea [9], акцент зроблено на використанні моделей

систем для оптимізації виробничих процесів, зокрема в промисловому середовищі. Автори розглядають застосування математичних моделей для мінімізації витрат і підвищення ефективності виробничих ліній, зокрема використовуючи метод Монте-Карло для оцінки змінних системи. Вони досліджують поведінку виробничих систем при різних навантаженнях, що дозволяє адаптувати їх до умов реальної експлуатації. Оптимізації багатоканальних систем присвячена робота J. M. Matzka [10]. В роботі Y. Chen, J. Dong [11] проведено дослідження оптимізації СМО правилом двох класів.

Сучасні дослідження, такі як спеціальний випуск журналу «Mathematics» [12], L. Lakatos, L. Szeidl, M. Telek [13], J. H. Dshalalow [14], фокусуються на прикладних аспектах використання теорії черг. Цей випуск об’єднує результати досліджень у різних галузях, таких як комп’ютерні мережі, інженерні системи та економіка, підкреслюючи, що моделі черг стали невід’ємною частиною процесу проєктування та оптимізації систем з високим навантаженням. У дослідженнях зазначається, що розрахунки розмірів черг, кількості серверів обслуговування та ймовірностей відмов є критично важливими для забезпечення ефективного функціонування систем.

Література з планування на основі інформації про час обслуговування (або розмір завдань) має довгу історію. Одними з перших, хто дослідив політику Shortest Remaining Processing Time (SRPT) та інші політики планування для черг типу $M/GI/1$, були автори роботи [15]. Оскільки ці політики вимагають ідеальної інформації про розмір завдання, їх зазвичай застосовують у плануванні задач для машинних систем [16]. Окрім SRPT, популярними є політики поділу процесорів (Processor Sharing, PS) та Shortest Job First (SJF). Нещодавно [17] розроблено єдину структуру для аналізу таких політик планування. Однак, порівняно небагато робіт досліджують випадки з неточною інформацією [18, 19]. Варто зазначити, що політика Foreground-Background (FB) не вимагає попередньої інформації про розмір завдань і показує хороші результати, коли розподіл часу обслуговування має спадну інтенсивність відмов [20].

Загалом, політики, засновані на індексі Гіттіна (Gittins index), мінімізують середній час відгуку за різних інформаційних умов [21, 22], проте їх реалізація може бути обчислювально складною. У [23] досліджується вигода від інтелектуального планування, яке базується лише на передбачених даних про час обслуговування з використанням даних колл-центру. Дана робота робить внесок в область оптимізації систем масового обслуговування з чергами, пропонуючи модель розбиття вхідного потоку заявок за розміром з подальшою відносною пріоритезацією менших за часом виконання задач. В такий спосіб досягається зменшення відмов у системах з обмеженою чергою.

У [24] встановлюється дифузійна межа для черг PS. У роботах [25 – 27] розроблено дифузійні межі для процесів, пов'язаних із чергами SRPT. Подібно до цих робіт, можна встановити дифузійну межу процесів довжини черги для класів пріоритетів. Це масштабування подібне до того, що встановлено у [26, 27] для SRPT. Межа важкого трафіку для середнього часу відгуку в сталому стані за політикою SRPT досліджується у [28]. Важливим спостереженням з цієї області є те, що продуктивність політики SRPT значною мірою залежить від властивостей хвоста розподілу часу обслуговування.

Ця робота також пов'язана з дослідженнями політик планування та пріоритезації для систем з кількома класами клієнтів. Одними з перших, хто проаналізував оптимальність важливої індексної політики – правила $c\mu$, були автори [29], де c – це вартість утримання клієнта за одиницю часу, а μ – швидкість обслуговування. Згідно з правилом $c\mu$, пріоритет надається класу із більшим значенням $c\mu$. Пізніше у роботах [30, 31] розширили правило $c\mu$ до більш загальних умов, використовуючи асимптотичний аналіз важкого трафіку.

У моделі оптимізації за двома класами [11] розглядається випадок, коли вартість утримання є однаковою для всіх, що зводить правило $c\mu$ до того, де пріоритет надається класу з коротшим середнім часом обслуговування. Головне спостереження цієї області полягає в тому, що під час дифузійного масштабування черга залишається лише з завданнями з найнижчим

пріоритетом. Це призводить до схожого результату щодо колапсу станів. Основна відмінність даної роботи від існуючої літератури полягає в тому, що для кожного класу виділяється окрема черга, в результаті чого більші задачі не можуть заблокувати чергу для менших задач. Сам пріоритет за класами не є фіксованим, і може динамічно підлаштовуватись під фактичний стан системи. Клас з низьким пріоритетом скорочується з ростом інтенсивності трафіку. Дифузійне масштабування, яке застосовано у цій роботі, є меншим, ніж те, що використовувалося в літературі щодо правила $c\mu$.

Роботи [32, 33] також досліджують правило $c\mu$ з неточною інформацією. Важливо зазначити, що оптимальність політик планування зі спрощеною структурою, таких як SRPT або правило $c\mu$, була доведена лише для однолінійних черг. У асимптотичному сенсі можна розширити результати оптимальності до черг із кількома серверами за умов традиційного важкого трафіку, коли кількість серверів фіксується, а інтенсивність трафіку наближається до 1. Наприклад, у роботі [34] встановлено оптимальність SRPT для черг із кількома серверами у традиційному важкому трафіку. У багатосерверному важкому трафіку, де кількість серверів прямує до нескінченності разом зі швидкістю надходження запитів, асимптотична оптимальність політик типу $c\mu$ може не зберігатися (див., наприклад, [35, 36]).

Також в процесі аналізу та постановки задачі широко застосовувались джерела, що присвячені математичній теорії та практичному застосуванню систем масового обслуговування [37], дослідження, пов’язані з багатоканальними системами масового обслуговування [38], аналіз марковських процесів [39]. У роботах [40, 41] розкриваються деякі прикладні питання сфери масового обслуговування. У [42, 43] розглядаються диференціальні рівняння для розрахунків параметрів СМО. Питання продуктивності систем з класами заявок розглядаються в роботах [44, 45]. Балансування навантаження в системах також розглядається у [46, 47, 48]. У роботах [49, 50] досліджується час планування вимог за умови динамічних пріоритетів. У [51, 52] приділяється увага питанню лімітів та оцінці меж

продуктивності в таких системах. Роботи [53, 54] допомогли систематизувати загальну картину роботи систем з динамічними пріоритетами.

Взаємодіюче співвиконання (concurrency) є важливою концепцією в сучасному програмуванні та теорії обчислень, що вимагає як теоретичного, так і практичного підходу. У літературі розглядаються як основні принципи, так і специфічні моделі для реалізації конкурентних процесів у багатоядерних системах.

Робота M. Nielsen [55] пропонує широкий огляд моделей для взаємодіючого співвиконання, зосереджуючись на формальних підходах до взаємодії між паралельними процесами та синхронізації. R. Segala у своїй праці [56] розглядає застосування ймовірнісних моделей у системах з недетермінованою поведінкою, що важливо для розуміння впливу випадкових факторів на конкурентні процеси. У книзі J. Reppy [57] розробляє підхід до конкурентного програмування в мові ML, що є цінним для функціональних мов програмування.

Праця C. A. R. Hoare [58] зосереджена на парадигмі обміну повідомленнями, яка досі залишається однією з основних моделей для розподілених систем. У статті H. Sutter, J. Larus [59] розглядаються виклики, пов’язані з багатоядерними системами, що стимулювало розвиток моделей, здатних ефективно працювати з паралельним виконанням завдань. Дослідження M. Chabbi, M. K. Ramanathan [60] стосується реальних прикладів станів неузгодженості даних у мові програмування Go, що підкреслює актуальність проблеми конкурентного доступу до спільних ресурсів.

У дослідженні Y. Liu та ін. [61] представлений метод оптимізації для багаторівневих структур з використанням паралельних топологій, що підходить для великих розподілених систем. Ще одна важлива стаття C. A. R. Hoare [62] розглядає основні аспекти процесів обміну повідомленнями, що залишаються основою сучасних конкурентних систем. У книзі A. A. A. Donovan, B. W. Kernighan [63] подано будову мови Go, детально розібрано її інструментарій та запропоновано рекомендації щодо побудови конкурентних програм.

Стаття E. Pontelli, G. Gupta [64] присвячена аналізу різних видів паралелізму в логічному програмуванні, що сприяє глибшому розумінню взаємозв'язків між паралельними процесами. E. Komendantskaya та інші у своїй роботі [65] зосередилися на використанні коалгебраїчної логіки для реалізації конкурентних процесів, що підкреслює важливість теоретичних основ для таких систем. У статті J. Whitney, C. Gifford, M. Pantoja [66] подано дослідження конкурентного виконання процесів у Golang, що збагачує розуміння конкурентних стратегій в сучасних мовах програмування.

Книга «Mastering GoLang: A Beginner's Guide» [67] від S. Uzayr пропонує вступний курс у конкурентне програмування мовою Go, розглядаючи основні інструменти та прийоми для розробників. Дослідження D. S. Fava, M. Steffen [68] спрямоване на виявлення проблемних місць конкурентного доступу до даних у Go, що має важливе значення для зниження ймовірності виникнення конфліктів при роботі зі спільними даними. Книга M. J. Sottile, T. G. Mattson, C. E. Rasmussen [69] забезпечує комплексний огляд основних мов і моделей програмування для взаємодіючого співвиконання.

Інша робота S. Uzayr [70] також охоплює основні аспекти програмування на Go з акцентом на паралельні можливості. У книзі M. C. Contreras [71] зібрано різноманітні патерни, що можуть використовуватися для реалізації конкурентних структур. Нарешті, «Concurrency in Go» написана K. Cox-Buday [72] пропонує детальне керівництво для розробників, які бажають впровадити ефективні інструменти взаємодіючого співвиконання в проекти на Golang.

Архітектура програмного забезпечення є критично важливою складовою процесу розробки, оскільки визначає структуру, поведінку та інтеграцію програмних систем, забезпечуючи їхню продуктивність, масштабованість та стійкість до змін. З розвитком технологій і зростанням вимог до продуктивності та кібербезпеки питання архітектури набуває особливої актуальності, стимулюючи появу нових методологій та інструментів, спрямованих на оптимізацію процесів проєктування і впровадження. Сучасна технічна література висвітлює широкий спектр архітектурних підходів – від монолітних

систем до мікросервісів, а також розглядає такі фактори, як адаптивність до змінних умов і здатність витримувати високі навантаження. Такі роботи надають розробникам і дослідникам надійні теоретичні та практичні основи для створення програмних систем, здатних відповідати складним вимогам сучасних ринків і технологічного середовища.

У книзі «Microsoft Application Architecture Guide» [73] автори R. Conery, S. Hanselman, P. Haack, S. Guthrie пропонують комплексне керівництво для розробників на платформі .NET, розглядаючи ефективні архітектурні практики для побудови стійких і продуктивних програмних систем. У роботі Н. В. Кірхар [74] представлено застосування технологій архітектурного проєктування програмного забезпечення, що дає змогу враховувати сучасні вимоги до продуктивності та стабільності в системах управління інформацією.

Ю. Форкун, В. Мартинюк, О. Яшина у своїй праці [75] зосереджуються на методології розробки архітектурної складової програмних застосунків, що актуально для технологічних процесів з високими вимогами до точності й адаптивності. Дослідження А. Морозов, Т. Вакалюк, Ю. Кубрак, Д. Зосімович [76] присвячене аналізу факторів, що впливають на архітектуру програмних систем, акцентуючи увагу на адаптації сучасних архітектурних підходів до зростаючих вимог кібербезпеки та надійності.

Стаття на платформі RubyGarage «System Architecture» [77] надає огляд сучасних архітектурних підходів, таких як монолітні структури, SOA і мікросервіси, що дозволяє розробникам вибрати оптимальну архітектуру для проєктів з урахуванням гнучкості та масштабованості. У методичній розробці А. О. Лісняк, О. М. Мильцев, В. В. Мухін, О. В. [78] представлено основи архітектури та проєктування програмного забезпечення, що робить їх важливим навчальним ресурсом для вивчення базових принципів інженерії програмного забезпечення.

У роботі Безкоровайний В. В., Чоломбитько Д. В. [84] реалізовано адаптивний алгоритм багатокритеріального розподілу робіт, що дозволяє враховувати стохастичний характер потоків робіт та часу їх виконання.

Математичні моделі задачі розподілу пакетів циклічних робіт розглядаються у монографії Bezkorovainyi V., Bezuhla H., Cholombytko D. [85], також у [86] Beskorovainyi V., Kolesnyk L., Russkin V. досліджують підтримку у прийнятті рішень в умовах неповної відповідності експертних переваг. Імітаційне моделювання процесу розподілу та виконання пакетів робіт вивчається у Bezkorovainyi V., Bezuhla H. [87].

1.2 Базові поняття систем масового обслуговування

Формально під системою масового обслуговування (далі – СМО) розуміють складну систему, що складається з одного або декількох джерел запитів (заявок, вимог) на виконання певних дій (обслуговування), декількох приладів обслуговування (ліній обслуговування, каналів обслуговування), що виконують ці дії відповідно до певних правил (дисципліни обслуговування) за запитами, що надійшли в систему (рис. 1.1) [43].

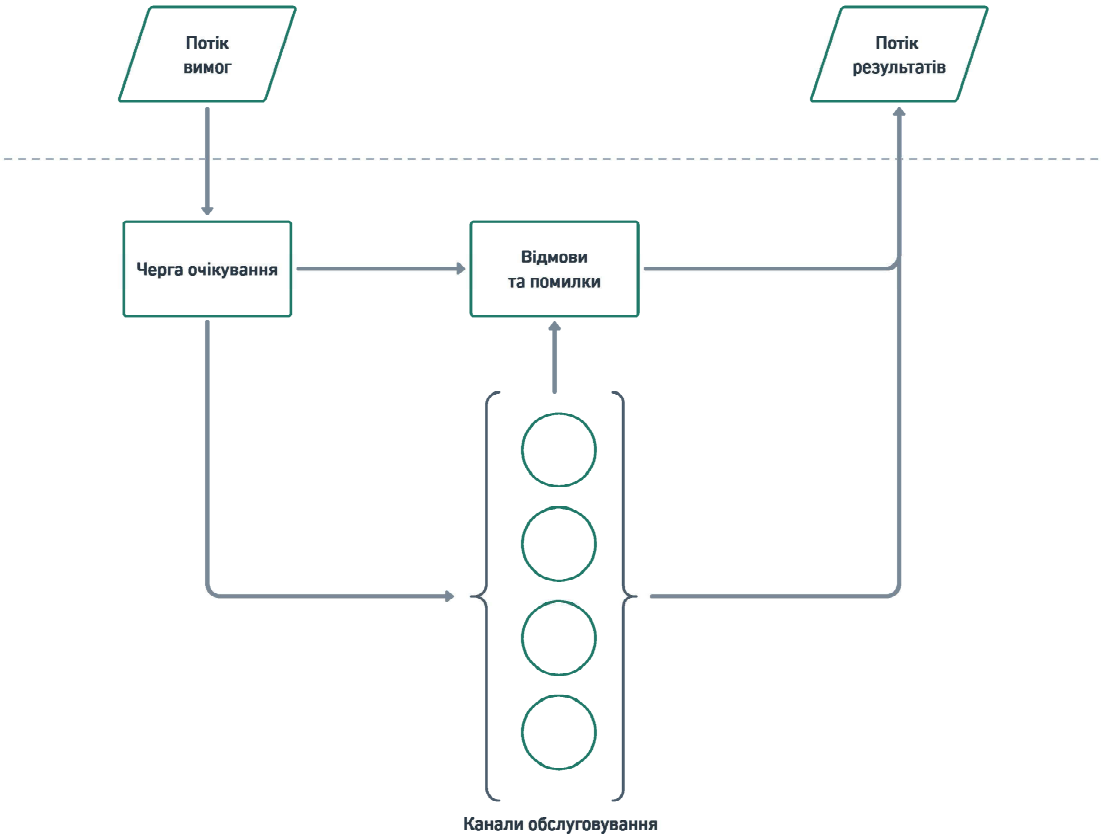

Рисунок 1.1 – Структура системи масового обслуговування

Характерною особливістю СМО є ймовірнісність процесів, що відбувається, і можливість утворення черги запитів на обслуговування. Розглянемо базові поняття СМО [43].

Джерело запитів визначається як зовнішня щодо СМО система, із якої запити надходять в неї для обслуговування. Джерело називають нескінченним або скінченним залежно від того, нескінченна чи скінченна кількість запитів міститься в ньому. Якщо джерело містить скінченну, але досить велику кількість запитів, то його зазвичай вважають нескінченним. Наприклад, хоча кількість користувачів інформаційно-пошукової системи Google є скінченним, припускають, що вони утворюють нескінченне джерело.

Вхідний потік складається з запитів, що надходять з нескінченного джерела, прибувають в канал обслуговування в моменти часу $t_0 < t_1 < ... < t_k < ....$ Інтервали часу $\xi_k = t_k - t_{k-1}$ $(k > 1)$ між послідовними моментами надходження запитів є випадковими величинами. Вважається, що ξ_k утворюють послідовність незалежних і однаково розподілених випадкових величин з функцією розподілу $A(t) = \mathbf{P}\{\xi_k < t\}$. У математичній теорії масового обслуговування використовують обмежений набір законів розподілу для опису вхідних потоків. Головними є пуассонівський (найпростіший) потік, що задається формулою Пуассона [37]:

$$p_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t},$$

де $\lambda > 0$ – параметр (інтенсивність) потоку.

Для визначення процесу обслуговування візьмемо η_k – тривалість обслуговування k-го запиту. Вважається, що η_k, $k = 1, 2, ...,$ – незалежні однаково розподілені випадкові величини з функцією розподілу $B(t) = \mathbf{P}\{\eta_k < t\}$. Функція $B(t)$ називається розподілом тривалості обслуговування. Припускаємо, що існує щільність імовірності $b(t) = B'(t)$. Як і для опису вхідних потоків запитів у теорії масового обслуговування

використовують обмежений набір законів розподілу для опису часу обслуговування. Найпоширенішим є експоненціальний закон розподілу з функцією розподілу $B(t) = 1 - \exp(-\mu t)$. Параметр μ інтерпретується як інтенсивність обслуговування. Середній час обслуговування дорівнює $\frac{1}{\mu}$. Крім експоненціального розподілу, використовують ерлангівський і гіперекспоненціальний розподіли, які отримують шляхом перетворення експоненціального закону.

Надзвичайно важливою компонентною є канали обслуговування. СМО може мати один або більше каналів (ліній, приладів, серверів) обслуговування. СМО з одним каналом називаються одноканальними або однолінійними, тоді як системи обслуговування, що містять більше каналів обслуговування, називаються багатоканальними або багатолінійними. Прилади обслуговування можуть бути однорідними та неоднорідними. У СМО з однорідними приладами всі прилади обслуговують запити однаково. У СМО з неоднорідними приладами прилади відрізняються один від одного деякими параметрами, наприклад, інтенсивністю обслуговування.

Дисципліна обслуговування складається з правил, згідно з якими запити вибираються для обслуговування. Умовно всі дисципліни обслуговування по наданих переваг в обслуговуванні діляться на дві групи: безпріоритетні та пріоритетні. Кожна з цих груп ділиться на ряд підгруп.

Безпріоритетні дисципліни обслуговування поділяються на дисципліну обслуговування в порядку надходження, у зворотному порядку, із випадковим вибором з черги та циклічну дисципліну обслуговування. Дисципліна обслуговування в порядку надходження (у міжнародній нотації FIFO – First Input First Output) є найпоширенішою та більшість досліджень щодо теорії масового обслуговування виконуються для цієї дисципліни. Вона широко використовується в операційних системах. Дисципліна обслуговування в зворотному (інверсному) порядку (у міжнародній нотації LIFO – Last Input First Output) використовується в операційних системах під час обробки переривань і

організації стеків. Циклічні дисципліни обслуговування використовуються під час обробці інформації в режимі поділу часу.

При пріоритетних дисциплінах обслуговування з черги на обслуговування спочатку вибираються заявки з вищим пріоритетом. Вони поділяються на дисципліни з фіксованими й динамічними пріоритетами [38].

При дисципліні обслуговування з відносним пріоритетом не дозволяється переривання обслуговування запиту у каналі. Якщо в систему з дисципліною обслуговування з абсолютним пріоритетом надійде запит з пріоритетом вищим, ніж той, що обслуговується, то він припинить обслуговування цього запиту та надійде на обслуговування. Системи з абсолютним пріоритетом розрізняють за кількістю рівнів пріоритету, а також за алгоритмами для обслуговування перерваних запитів.

При дисциплінах обслуговування з динамічним пріоритетом пріоритет конкретних запитів змінюється залежно від змінювання деяких величин, наприклад, часу очікування в черзі.

За наявністю певної ознаки системи масового обслуговування можна класифікувати наступним чином.

За кількістю вимог, що надходять за одиницю часу, СМО поділяються на системи з ординарним і неординарним потоками вимог. Якщо імовірність надходження двох і більше вимог одночасно дорівнює нулю або має настільки мале значення, що ним можна знехтувати, то отримаємо систему з ординарним потоком вимог. Наприклад, потік вимог – літаки, що надходить на злітно-посадкову смугу аеродрому, можна вважати ординарним.

За зв’язком між вимогами СМО поділяються на системи без післядії та з післядією. Якщо ймовірність надходження вимог у систему в деякий момент часу не залежить від того, скільки вимог уже надійшло, тобто не пов’язана з передісторією досліджуваного процесу, то отримаємо систему без післядії, в іншому разі – з післядією. Прикладом системи з післядією може слугувати потік студентів, що складають екзамен викладачу.

За реакцією вимоги на зайнятість каналів СМО поділяються на системи з

відмовами й очікуваннями. Якщо вимога, яка надійшла на обслуговування, застала всі канали зайнятими і змушена залишити систему, то отримаємо систему з відмовами.

Системи з очікуванням розподіляються на системи з обмеженим і необмеженим очікуванням. Якщо вимога залишає систему, коли черга набула певного розміру, то отримаємо систему з обмеженим очікуванням. Прикладом є самоскид з розчином. Якщо час очікування настільки великий, що розчин може затвердіти, то самоскид доречно розвантажити в іншому місці. Якщо вимога, яка надійшла, застала всі канали зайнятими і змушена очікувати своєї черги доти, доки вона не буде обслужена, то отримаємо систему з очікуванням без обмежень. Прикладом є літак, що перебуває на аеродромі й очікує звільнення злітної смуги.

За способом вибору вимог на обслуговування СМО розподіляються так:

– із пріоритетом вимог;

– у процесі надходження вимог;

– із випадковим вибором вимог;

– остання вимога обслуговується першою.

Якщо система масового обслуговування охоплює декілька категорій вимог і за якимись ознаками визначається порядок їх вибору на обслуговування, то отримаємо систему з пріоритетом вимог. Наприклад, під час надходження виробів на будівельний майданчик насамперед монтують ті, які вимагає послідовність виконання будівельної технологією.

Якщо канал, що звільнився, обслуговує вимогу, яка надійшла в систему раніше за інші, то отримаємо систему обслуговування вимог у процесі їх надходження. Наприклад, покупець, що підійшов першим до продавця, обслуговується першим.

Якщо вимоги з черги в канал обслуговування надходить випадково, то отримаємо систему з випадковим вибором вимог на обслуговування. Прикладом є вибір слюсарем-сантехніком однієї з декількох заявок від мешканців, із часом надходження яких він не ознайомлений.

Якщо для обслуговування обирається остання вимога, що надійшла, то отримаємо систему з вибором «останній обслуговується першим». Наприклад, під час укладання будівельних виробів штабелями зручніше обирати зі штабеля (черги) виріб, покладений останнім.

За часом обслуговування вимоги СМО поділяють на системи з детермінованим і випадковим часом обслуговування. Якщо інтервал часу між моментом надходження вимоги в канал обслуговування та моментом виходу вимоги з каналу є сталим, то отримаємо систему з детермінованим часом обслуговування, в іншому разі – з випадковим. Наприклад, миття автомобілів є системою обслуговування з детермінованим часом обслуговування.

За кількістю каналів обслуговування розрізняють одно- та багатоканальні системи. Наприклад, під час монтажу будинку може використовуватися один підіймальний кран (один канал обслуговування) або декілька (багато каналів обслуговування).

За кількістю етапів обслуговування СМО поділяють на одно- й багатофазні системи. Якщо канали обслуговування розташовуються послідовно й є неоднорідними, то отримаємо багатофазну систему обслуговування. Прикладом такої системи є обслуговування автомобілів на СТО (миття, діагностика, заміна фільтрів тощо).

За однорідністю вимог розрізняють системи з однорідними та неоднорідними потоками вимог. Наприклад, якщо під навантаження прибувають фургони однієї вантажопідйомності, то отримаємо систему з однорідним потоком вимог, якщо різної – із неоднорідним.

За завантаженістю каналів СМО поділяють на впорядковані та невпорядковані системи. У впорядкованих системах обслуговуючі канали завантажені нерівномірно. Вимога, що надійшла, обслуговується чітко визначеним каналом із наявних вільних, а саме – каналом з найменшим номером (вважається, що всі канали пронумеровані). У невпорядкованих системах усі канали однакові і вимога, що надійшла, обслуговується одним із вільних каналів без будь-яких переваг.

Для скороченого позначення розімкнутих безпріоритетних невпорядкованих систем масового обслуговування D. G. Kendall (див. [39]) запропонував таке позначення: $A/B/n/m$, де A і B описують, відповідно, закони розподілу інтервалів часу між подіями вхідного потоку запитів і часом обслуговування, n задає кількість каналів обслуговування, m – кількість місць очікування.

Певні закони розподілу позначають так: M – експоненціальний розподіл, E або E_k – ерлангівський розподіл; H або H_r – гіперекспоненціальний розподіл; D – регулярний потік подій; G – розподіл загального вигляду. Наприклад, запис $D/H_2/3/10$ позначає систему з трьома каналами обслуговування, сталим (детермінованим) часом між запитами вхідного потоку, час обслуговування розподілено за гіперекспоненціальним законом другого порядку й кількістю місць для очікування дорівнює 10. Якщо параметр m відсутній, то це означає, кількість місць для очікування є нескінченною.

1.3 Показники ефективності систем масового обслуговування

Показники ефективності систем масового обслуговування поділяються на технічні, що характеризують якість і умови роботи обслуговуючої системи, і економічні, що відображають економічні особливості системи.

Показники першої групи зазвичай формують на підставі отриманих із розрахунків значень ймовірностей станів системи. Показники другої групи – на підставі показників першої групи.

1.3.1 Технічні показники ефективності систем масового обслуговування

Серед технічних показників можна відокремити такі:

1. Імовірність відмови обслуговування. Імовірність того, що вимога, яка надходить у систему, відмовиться приєднатися до черги та втратиться

системою, позначимо $P_{\text{відм.}}$. Цей показник для системи масового обслуговування з відмовами дорівнює імовірності того, що в системі міститься стільки вимог, скільки є каналів обслуговування:

$$P_{\text{відм.}} = p_n,$$

де n – кількість каналів обслуговування.

Для системи з обмеженою довжиною черги $P_{\text{відм.}}$ дорівнює імовірності того, що в системі міститься $n+m$ вимог:

$$P_{\text{відм.}} = p_{n+m},$$

де m – допустима довжина черги.

Протилежним показником є імовірність обслуговування вимоги:

$$P_{\text{обсл.}} = 1 - p_{n+m}.$$

2. Середня кількість вимог, що очікують на обслуговування:

$$L_q = \sum_{i=n+1}^{n+m} (i-n)\, p_i,$$

де p_i – імовірність того, що в системі міститься i вимог.

3. Відносна й абсолютна пропускні здатності системи, які визначають за такими формулами:

– відносна пропускна здатність:

$$Q = 1 - P_{\text{відм.}};$$

– абсолютна пропускна здатність:

$$A = \lambda \cdot Q;$$

4. Середня кількість зайнятих обслуговуванням каналів для систем $G/G/n/m$ (тобто n – обслуговуючі пристрої, m – місця для очікування):

$$\overline{n} = \sum_{i=1}^{n-1} ip_i + n\sum_{i=1}^{n+m} p_i .$$

Для систем масового обслуговування з відмовами середню кількість зайнятих обслуговуванням каналів можна знайти за формулою:

$$\overline{n} = \sum_{i=1}^{n} ip_i .$$

5. Загальна кількість вимог, що містяться в системі L_s. Цей показник визначають так:

– для систем масового обслуговування з відмовами:

$$L_s = \overline{n};$$

– для систем масового обслуговування з обмеженою довжиною черги:

$$L_s = \overline{n} + L_q .$$

6. Середній час очікування вимогами початку обслуговування. Якщо відома функція розподілу ймовірності часу очікування вимогою початку обслуговування $W(t) = \mathbf{P}(T_q < t)$, то середній час очікування вимогами початку обслуговування T_q визначається як математичне сподівання випадкової величини T_q:

$$\tau_q = \mathbf{M}[T_q] = \int_0^{+\infty} tdW(t) .$$

Для показникового (експоненціального) закону розподілу вимог у

вхідному потоці τ_q можна визначити за формулою

$$\tau_q = \frac{L_q}{\lambda}.$$

1.3.2 Економічні показники ефективності систем масового обслуговування

Показники, що характеризують економічні особливості систем масового обслуговування, зазвичай формують відповідно до певного виду системи та її призначення. Одним із загальних показників є економічна ефективність системи:

$$G = \lambda P_{\text{обсл.}} C_s T - E,$$

де C_s – середній економічний ефект, отриманий під час обслуговуванні однієї вимоги;

T – розглянутий інтервал часу;

E – величина втрат у системі.

Останню величину (втрати) можна визначити так:

– для систем з відмовами:

$$E = (C_{\text{е}} \overline{n} + \lambda C_{\text{зб.}} P_{\text{відм.}} + C_{\text{прост.}} \overline{n}_{\text{вільн.}}) T,$$

де $C_{\text{е}}$ – вартість експлуатації одного каналу за одиницю часу;

$C_{\text{зб.}}$ – вартість збитків унаслідок виходу вимог із системи за одиницю часу;

$C_{\text{прост.}}$ – вартість одиниці часу простоювання каналу обслуговування;

$\overline{n}_{\text{вільн.}}$ – середня кількість вільних каналів (що простоюють), $\overline{n}_{\text{вільн.}} = n - \overline{n}$;

– для систем з очікуванням:

$$E = (C_{\text{є}}\overline{n} + \lambda C_q L_q + C_{\text{зб.}}\overline{n}_{\text{вільн.}})T,$$

де C_q – вартість втрат, пов'язаних із простоюванням вимоги в черзі за одиницю часу.

1.4 Диференціальний метод дослідження систем масового обслуговування

Головним завданням дослідження СМО є встановлення залежностей базових характеристик, а саме: середній час очікування, ймовірність відмови обслуговування від параметрів системи – інтенсивностей обслуговування на приладах, кількості приладів обслуговування, дисципліни обслуговування.

Диференціальний метод можна застосовувати для дослідження СМО з пуассонівськими потоками подій, тобто час обслуговування й інтервали часу між надходженням запитів підпорядковані експоненціальному розподілу. У цьому разі процес змінювання кількості запитів у системі є марковським і його можна дослідити так:

– із аналізу системи обслуговування відокремлюють стани системи, можливі переходи між станами та їх інтенсивності і складають граф станів і переходів (рис 1.2);

– із кожним станом зв'язується відповідна ймовірність і складається система диференціальних рівнянь для ймовірностей станів системи.

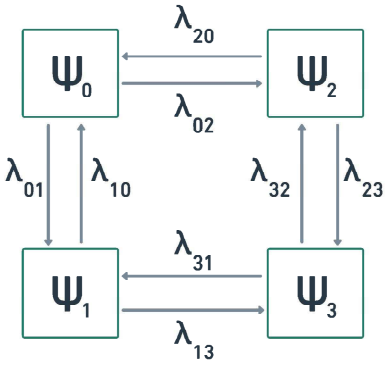

Рисунок 1.2 – Граф станів і переходів для системи з відмовами

Складену систему доцільно розв'язувати та досліджувати за допомогою однієї з систем комп'ютерної математики, наприклад, Maple, Mathematica чи Mathcad.

Якщо метою є дослідження системи у стаціонарному режимі, то складається система лінійних алгебраїчних рівнянь для стаціонарних ймовірностей станів. Зазвичай розв'язок вдається звести до простих розрахункових формул.

Використання диференціального методу для аналізу СМО розглянемо на системі обслуговування з груповим вибором запитів для обслуговування [40]. Ця СМО складається з накопичувача, у який надходить пуассонівський потік запитів інтенсивністю λ. Через деякі інтервали часу запити, що накопичилися, вибираються для обслуговування. Ємкість накопичувача є k запитів. Інтенсивність вибору запитів з накопичувача дорівнює μ_i $(i = 0, 1, ..., k)$, тобто вона обумовлюється кількістю запитів у ньому. Цю СМО можна використовувати, наприклад, як модель роботи узгоджувального буфера для інформаційної системи.

Нехай S_i – стан, коли в накопичувачі міститься i запитів, ймовірність чого є p_i, $i = 1, 2, ..., k$. Ця СМО відповідно до надходження нового запиту переходить зі стану S_i у стан S_{i+1}, інтенсивність цього переходу є λ. Під час вибору запитів із накопичувача відбувається перехід зі стану S_i, $i = 1, 2, ..., k$, у стан S_0. Інтенсивність цього переходу є μ_i. Граф станів і переходів для цієї СМО при $k = 4$ зображений на рис. 1.3.

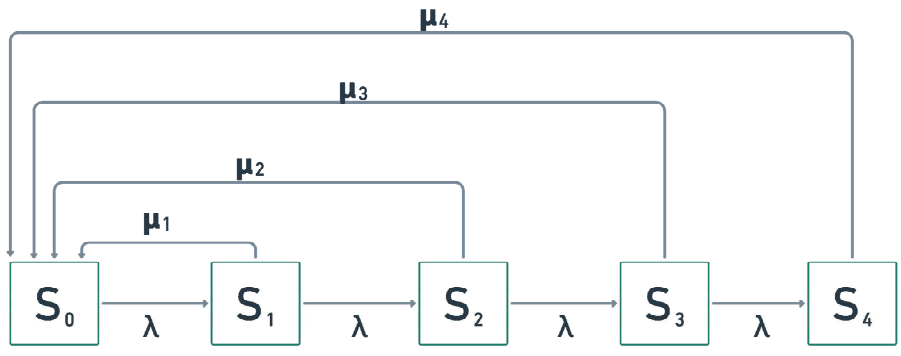

Рисунок 1.3 – Граф станів і переходів для системи з груповим вибором

Склавши для досліджуваної СМО систему диференціальних рівнянь, отримаємо:

$$p_0'(t) = -\lambda p_0(t) + \sum_{i=1}^{k} \mu_i p_i(t),$$

$$p_i'(t) = -(\lambda + \mu_i) p_i(t) + \lambda p_{i-1}(t), \quad 1 \le i \le k,$$

$$p_k'(t) = -\mu_k p_k(t) + \lambda p_{k-1}(t). \qquad (1.1)$$

Початкові умови є $p_0(0) = 1,\ p_i(0) = 1,\ i \neq 0$.

Для стаціонарного стану система (1.1) перетворюється на таку:

$$-\lambda p_0 + \sum_{i=1}^{k} \mu_i p_i = 0,$$

$$-(\lambda + \mu_i) p_i + \lambda p_{i-1} = 0,\ 1 \le i \le k,$$

$$-\mu_k p_k + \lambda p_{k-1} = 0. \qquad (1.2)$$

Умова нормування має вигляд

$$p_0 + p_1 + p_2 + ... + p_k = 1. \qquad (1.3)$$

З рівнянь системи (1.2), починаючи з другого, послідовно отримаємо:

$$p_1 = \frac{\lambda}{(\lambda + \mu_1)} p_0,\ \ p_2 = \frac{\lambda}{(\lambda + \mu_2)} p_1 = \frac{\lambda^2}{(\lambda + \mu_1)(\lambda + \mu_2)} p_0, \ \ldots,$$

$$p_i = \frac{\lambda^i}{\prod_{l=1}^{i} (\lambda + \mu_l)} p_0, \ \ldots,$$

а отже,

$$p_k = \frac{\lambda^k}{\mu_k \prod_{l=1}^{k-1} (\lambda + \mu_l)} p_0.$$

Підставивши значення p_i, виражені через p_0, в умову нормування (1.3), отримаємо:

$$p_0 = \left[1 + \sum_{i=1}^{k-1} \frac{\lambda^i}{\prod_{l=1}^{i} (\lambda + \mu_l)} + \frac{\lambda^k}{\mu_k \prod_{l=1}^{k-1} (\lambda + \mu_l)} \right]^{-1}.$$

Базові характеристики досліджуваної системи:

– ймовірність відмови щодо обслуговування:

$$P_{\text{відм.}} = p_k;$$

– середня кількість запитів у накопичувачі:

$$L_q = \sum_{i=1}^{k} i p_i;$$

– середній час перебування запитів у накопичувачі:

$$\tau_q = \sum_{i=1}^{k} \frac{p_i}{\mu_i}.$$

За цими виразами можна обирати обсяг буфера, щоб імовірність його переповнення містилася в заданих межах.

1.5 Система масового обслуговування з відмовами

СМО з відмовами складається з n каналів обслуговування, на які надходить найпростіший потік запитів з інтенсивністю λ, час обслуговування розподілений за експоненціальним законом з параметром μ, тобто потік

обслуговування також найпростіший з параметром $\frac{1}{\mu}$. Черги не спостерігається, тобто запит, який надійшов коли всі канали зайняті, отримує відмову щодо обслуговування. Ця система є однією з найпростіших і на ній будується теорія систем масового обслуговування. Вперше її розглянув видатний дослідник A. K. Erlang (див. [37]). Типовим прикладом застосування СМО з відмовами є автоматичні телефонні станції. Тут запитом, який надійшов у систему, є звернення клієнта до телефонної станції. Якщо потрібна лінія зв'язку вже зайнята розмовою, абонент отримує відмову. Автоматична телефонна станція дає часті гудки й запит губиться [41].

Дослідимо її. Позначимо через S_i стан системи, коли в ній перебуває i, $i = 0, 1, ..., n$, запитів. Граф станів і переходів для цієї системи при $n = 5$ зображено на рис. 1.4.

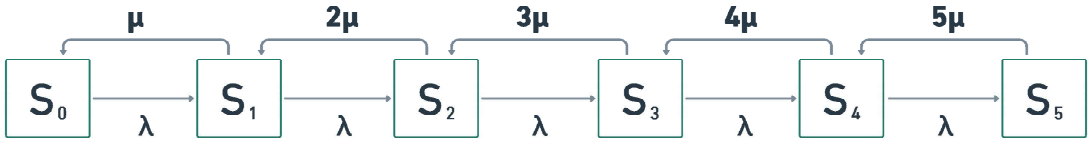

Рисунок 1.4 – Граф станів і переходів СМО з відмовами

Позначимо через $p_i(t)$ ймовірність стану S_i, складемо систему диференціальних рівнянь для станів S_i, $i = 0, 1, 2, ..., n$:

$$p_0'(t) = -\lambda p_0(t) + \mu p_1(t),$$
$$p_1'(t) = -(\lambda + \mu) p_1(t) + \lambda p_0(t) + 2\mu p_2(t),$$
$$p_i'(t) = -(\lambda + i\mu) p_i(t) + \lambda p_{i-1}(t) + (i+1)\mu p_{i+1}(t),\ 1 \le i \le n,$$
$$p_n'(t) = -n\mu p_n(t) + \lambda p_{n-1}(t). \qquad (1.4)$$

Початкові умови мають вигляд:

$$p_0(0) = 1,\ p_i(0) = 0,\ 1 \le i \le n.$$

Якщо n невелике, то, використовуючи техніку розв'язування систем лінійних диференціальних рівнянь [42] при конкретних значеннях параметрів можна знайти аналітичний розв'язок цієї системи. Наприклад, при $\lambda = 1$, $\mu = 0,6$, $n = 2$ розв'язок системи (1.4) виглядає так:

$$p_0(t) = 0,1169e^{1,0693t} + 0,6364e^{2,7307t} + 0,2466e^{3,8t},$$
$$p_1(t) = -0,3375e^{1,0693t} - 0,0736e^{2,7307t} + 0,411e^{3,8t},$$
$$p_2(t) = 0,2204e^{1,0693t} - 0,5628e^{2,7307t} + 0,3425e^{3,8t}.$$

Графіки залежностей функцій $p_0(t)$, $p_1(t)$, $p_2(t)$ від часу наведено на рис. 1.5.

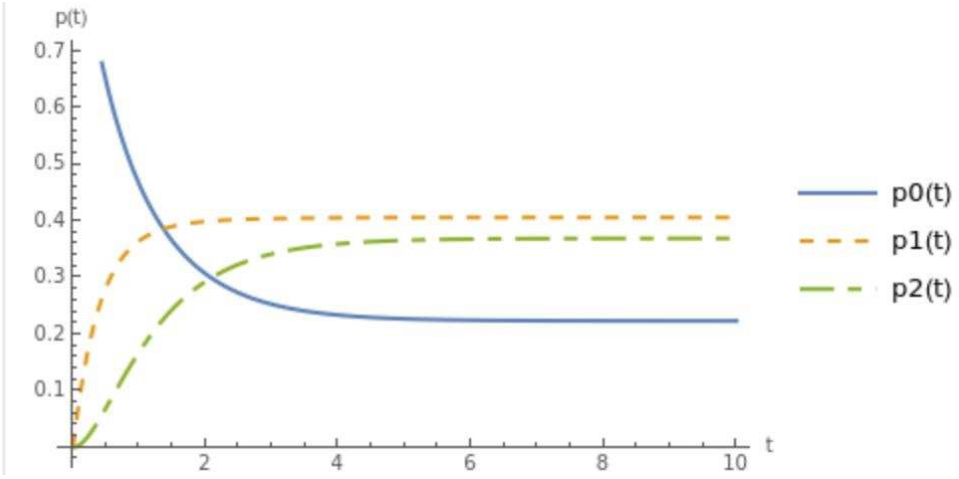

Рисунок 1.5 – Залежності ймовірностей станів системи з відмовами від часу

Зрозуміло, що в разі необмеженого зростання t поведінка функцій $p_i(t)$ перестає залежати від часу й система переходить у стаціонарний режим, тобто $\lim\limits_{t\to+\infty} p_i(t) = p_i$, $i = 0, 1, 2$, де p_i – константи. З аналізу виразів для функцій $p_i(t)$ випливає, що, коли $t \to +\infty$, вони прямують до відповідних констант у правій частині, тобто $p_0 = 0,224$, $p_1 = 0,406$, $p_2 = 0,370$.

Ймовірності станів системи в стаціонарному режимі можна визначити простішим методом. З огляду на те що $\lim\limits_{t\to+\infty} p_i'(t) = \lim\limits_{t\to+\infty} p_i' = 0$ система рівнянь (1.4) перетворюється так:

$$
\begin{gathered}
-\lambda p_0 + \mu p_1 = 0, \\
-(\lambda + \mu) p_1 + \lambda p_0 + 2\mu p_2 = 0, \\
-(\lambda + i\mu) p_i + \lambda p_{i-1} + (i+1)\mu p_{i+1} = 0,\ 1 \le i \le n, \\
-n\mu p_n + \lambda p_{n-1} = 0.
\end{gathered} \tag{1.5}
$$

Умова нормування є

$$
\sum_{i=0}^{n} p_i = 1. \tag{1.6}
$$

Поділимо кожне рівняння системи (1.5) на μ і введемо параметр $\rho = \frac{\lambda}{\mu}$, що має сенс навантаження системи по одному каналу. Система рівнянь (1.6) стане такою:

$$
\begin{gathered}
-\rho p_0 + p_1 = 0, \\
-(\rho + 1) p_1 + \rho p_0 + 2 p_2 = 0, \\
-(\rho + 1) p_i + \rho p_{i-1} + (i+1) p_{i+1} = 0,\ 1 \le i \le n, \\
-n p_n + \rho p_0 = 0.
\end{gathered} \tag{1.7}
$$

Послідовно розв’язуючи систему (1.7) відносно p_0, отримаємо:

$$
p_i = \frac{\rho^i}{i!} p_0.
$$

Підставивши цей вираз у (1.6), визначимо:

$$
p_0 = \left(1 + \sum_{i=1}^{n} \frac{\rho^i}{i!} \right)^{-1}.
$$

Остаточно ймовірність того, що обслуговується i запитів, тобто система

перебуває в стані S_i, визначається виразом

$$p_0 = \left(\frac{\rho^i}{i!} \sum_{k=0}^{n} \frac{\rho^k}{k!} \right)^{-1}, \; 0 \le i \le n.$$

Ці вирази називають формулами Ерланга. Доведено, що ця формула має місце для будь-якого неперервного закону розподілу часу обслуговування у каналі [41].

Базові характеристики досліджуваної СМО з відмовами такі:

– імовірність відмови щодо обслуговування

$$P_{\text{відм.}} = p_n = \frac{\rho^n}{n!} \left(\sum_{k=0}^{n} \frac{\rho^k}{k!} \right)^{-1},$$

що означає, що цьому запиту буде відмовлено щодо обслуговування, якщо всі лінії будуть зайняті;

– абсолютна пропускна здатність дорівнює середній кількості запитів, які може обслужити СМО за одиницю часу, тобто це частка вхідного потоку заявок, яка надходить у систему й обслуговується нею:

$$A = \lambda Q = \lambda \left[1 - \frac{\rho^n}{n!} \left(\sum_{k=0}^{n} \frac{\rho^k}{k!} \right)^{-1} \right];$$

– середня кількість запитів у системі, а відповідно й середня кількість каналів, що зайняти обслуговуванням запитів, обчислюється як математичне сподівання:

$$L_s = \sum_{i=1}^{n} i p_i = \sum_{i=1}^{n} i \frac{\rho^i}{i!} \left(\sum_{k=0}^{n} \frac{\rho^k}{k!} \right)^{-1};$$

– коефіцієнт завантаженості системи

$$K_z = \frac{L_s}{n} = \sum_{i=1}^{n} i \frac{\rho^i}{ni!} \left(\sum_{k=0}^{n} \frac{\rho^k}{k!} \right)^{-1}.$$

Доцільно знати залежності базових характеристик системи від n і коефіцієнта завантаженості за всіма каналами $\alpha = \frac{\lambda}{n\mu} = \frac{\rho}{n}$. Звідси $\rho = n\alpha$. Тоді базові характеристики будуть такими:

– імовірність відмови щодо обслуговування

$$P_{\text{відм.}} = p_n = \frac{(n\alpha)^n}{n!} \left(\sum_{k=0}^{n} \frac{(n\alpha)^k}{k!} \right)^{-1};$$

– коефіцієнт завантаженості системи

$$K_z = \frac{L_s}{n} = \sum_{i=1}^{n} i \frac{(n\alpha)^i}{ni!} \left(\sum_{k=0}^{n} \frac{(n\alpha)^k}{k!} \right)^{-1}.$$

На рис. 1.6 зображено графік залежності ймовірності відмови щодо обслуговування від кількості каналів обслуговування при різних ρ.

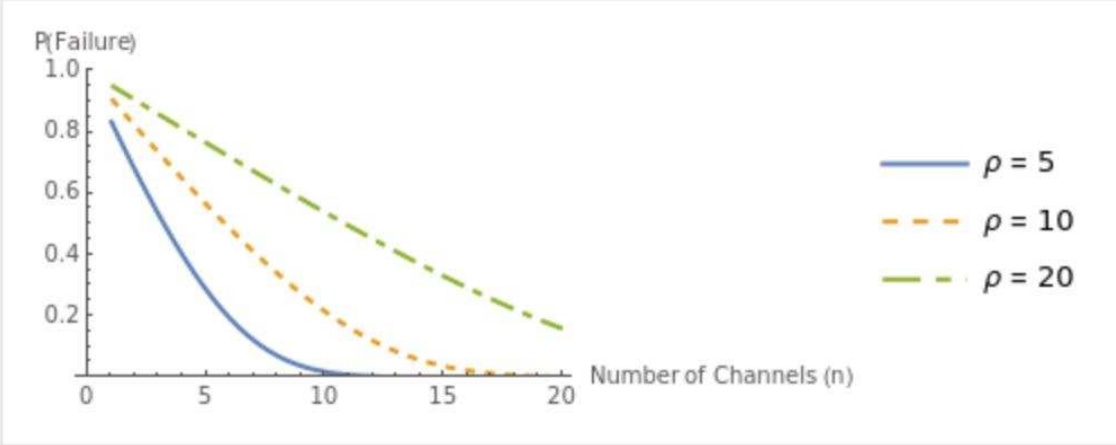

Рисунок 1.6 – Залежність імовірності відмови щодо обслуговування для СМО з відмовами

На рис. 1.7 зображено графік залежності коефіцієнта завантаженості

системи від кількості каналів обслуговування при різних ρ. За цими графіками можна обирати кількість каналів обслуговування при різних коефіцієнтах завантаженості одного каналу ρ залежно від технічних завдань на проектування системи. Наприклад, якщо потрібно обрати кількість каналів обслуговування в разі імовірності відмов 0,1 при $\rho = 10$, то з рис. 1.6 випливає, що їх має бути не менше 13 одиниць.

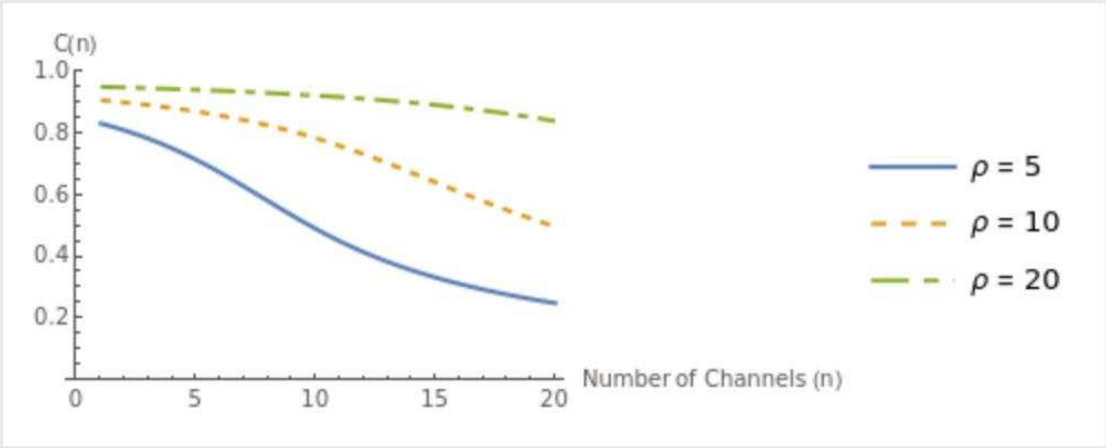

Рисунок 1.7 – Залежність коефіцієнта завантаженості системи для СМО з відмовами

Якщо кількість каналів обирається залежно від коефіцієнта завантаження системи, то потрібно використовувати графіки, подані на рис. 1.7. Наприклад, якщо потрібно обрати кількість каналів обслуговування в разі коефіцієнта завантаженості системи $K_z = 0,7$ при $\rho = 10$, то з рис. 1.7 випливає, що їх має бути не більше 14 одиниць.

1.6 Багатоканальна система масового обслуговування з обмеженою чергою

Розглянемо пуассонівську систему масового обслуговування з n каналами обслуговування і кількістю міст для очікування m ($M/M/n/m$ в позначеннях Кендалла). Позначимо через $\{i\}$, $i = 1, 2, ..., n+m-1, n+m$, стан

системи, коли в ній перебуває i запитів. Якщо $i \geq n$, то n запитів будуть обслуговуватися, а решта перебуватиме у черзі. Імовірність стану $\{i\}$ позначимо p_i. Граф станів і переходів для цієї системи зображено на рис. 1.8.

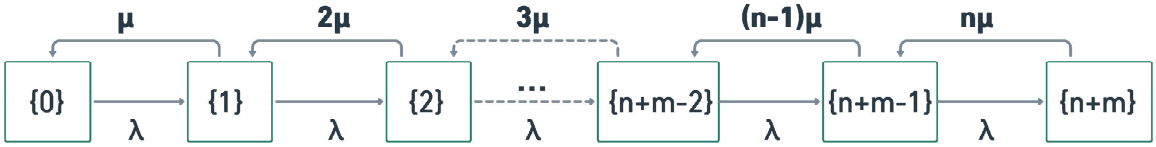

Рисунок 1.8 – Граф станів і переходів для системи $M/M/n/m$

Складемо систему лінійних рівнянь для ймовірностей p_i, $i = 1, 2, ..., n+m$:

$$
\begin{gathered}
-\lambda p_0 + \mu p_1 = 0, \\
-(\lambda + i\mu) p_i + \lambda p_{i-1} + (i+1)\mu p_i = 0,\ 1 \leq i < n, \\
-(\lambda + n\mu) p_i + \lambda p_{i-1} + n\mu p_i = 0,\ n \leq i < n+m, \\
-n\mu p_{n+m} + \lambda p_{n+m-1} = 0.
\end{gathered} \tag{1.8}
$$

Система (1.8) доповнюється умовою нормування

$$
p_0 + p_1 + p_2 + ... + p_{n+m-1} + p_{n+m} = 1. \tag{1.9}
$$

Поділимо кожне рівняння на $n\mu$ і введемо позначення $\rho = \dfrac{\lambda}{n\mu}$ – завантаженість всієї системи. Система (1.8) перепишеться так:

$$
\begin{gathered}
-\rho p_0 + \frac{p_1}{n} = 0, \\
-\left(\rho + \frac{i}{n}\right) p_i + \rho p_{i-1} + (i+1) n p_i = 0,\ 1 \leq i < n, \\
-(\rho + 1) p_i + \rho p_{i-1} + p_i = 0,\ n \leq i < n+m, \\
-p_{n+m} + \rho p_{n+m-1} = 0.
\end{gathered} \tag{1.10}
$$

З першого рівняння системи (1.10) випливає, що $p_1 = n\rho p_0$. Підставимо цей вираз у друге рівняння системи (1.10). Отримаємо, що $p_2 = \frac{n^2}{2!}\rho^2 p_0$. Продовживши цю операцію, отримаємо:

$$p_i = \frac{n^i}{i!}\rho^i p_0,\ 1 \le i \le n,$$

$$p_i = \frac{n^n}{n!}\rho^i p_0,\ n < i \le n+m.$$

Підставивши p_i в умову нормування (1.9), знайдемо вираз для імовірності простоювання системи:

$$p_0 = \left[1 + \sum_{i=1}^{n-1}\frac{n^i\rho^i}{i!} + \sum_{i=n}^{n+m}\frac{n^n\rho^i}{n!}\right]^{-1} = \left[\sum_{i=0}^{n-1}\frac{n^i\rho^i}{i!} + \frac{n^n}{n!}\frac{\rho^n(1-\rho^{m+1})}{1-\rho}\right]^{-1}.$$

Отже, усі імовірності станів визначено. Визначимо базові показники цієї системи.

Імовірність відмови щодо обслуговування дорівнює

$$P_{\text{відм.}} = p_{m+n} = \frac{n^n}{n!}\rho^{n+m} p_0.$$

Цей вираз означає, що запит отримає відмову щодо обслуговування, якщо всі лінії й місця очікування будуть зайняті.

Кількість запитів $N_{\text{відм.}}$, що отримають відмову щодо обслуговування за час T, можна обчислити за формулою

$$N_{\text{відм.}} = \lambda p_{m+n} T = \lambda\frac{n^n}{n!}\rho^{n+m} p_0 T.$$

Імовірність того, що запит буде обслуговано системою, дорівнює

$$P_{\text{обсл.}} = 1 - P_{\text{відм.}} = 1 - \frac{n^n}{n!}\rho^{n+m} p_0 .$$

Окрім того, це відносна пропускна здатність системи, яка дорівнює середній частці запитів, що надійшли в систему й обслуговується нею.

Абсолютна пропускна здатність A дорівнює середній кількості запитів, які може обслужити СМО за одиницю часу, тобто це частка вхідного потоку заявок, яка надходить у систему й обслуговується нею:

$$A = \lambda P_{\text{обсл.}} = \lambda\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right).$$

Визначимо L_q – середню кількість запитів, що очікують на обслуговування. Якщо СМО перебуває в стані $\{n+1\}$, то в черзі буде один запит, імовірність цього є p_{n+1}. Якщо СМО буде перебувати в стані $\{n+m\}$, то в черзі буде m запитів, імовірність цього є p_{n+m}. Середня кількість запитів у черзі на обслуговування дорівнює математичному сподіванню кількості запитів у черзі, її можна обчислити за формулою:

$$L_q = \sum_{i=1}^{m} i \cdot p_{n+i} = \sum_{i=1}^{m} i \frac{n^n}{n!}\rho^{n+i} p_0 = \frac{n^n}{n!}\rho^{n+i}(1 + 2\rho + ... + m\rho^{m-1}) p_0 . \qquad (1.11)$$

Якщо $\rho = 1$, то вираз у дужках перетворюється на арифметичну прогресію $1 + 2 + 3 + ... + m$. Її сума дорівнює $\frac{m(m+1)}{2}$. Тоді

$$L_q = \frac{n^n}{n!} \cdot \frac{m(m+1)}{2} p_0 .$$

Нехай $\rho \neq 1$. Спростимо вираз у дужках за виразом (1.11):

$$1+2\rho+3\rho^2+...+m\rho^{m-1}=\frac{d}{d\rho}(\rho+\rho^2+\rho^3+...+\rho^m).$$

Вираз у дужках – це скінченна геометрична прогресія зі знаменником ρ. Сума її членів:

$$\rho+\rho^2+\rho^3+...+\rho^m=\frac{\rho(1-\rho^m)}{1-\rho}. \qquad (1.12)$$

Обчислимо похідну від виразу (1.12):

$$\frac{d}{d\rho}\left(\frac{\rho(1-\rho^m)}{1-\rho}\right)=\frac{1-\rho^m(1+m-m\rho)}{(1-\rho)^2}. \qquad (1.13)$$

Підставивши (1.13) у (1.11), можна знайти L_q при $\rho\neq 1$. Остаточно

$$L_q=\begin{cases}\dfrac{n^n}{n!}\dfrac{1-\rho^m(1+m-m\rho)}{(1-\rho)^2}\rho^{n+1}p_0, & \rho\neq 1,\\ \dfrac{n^n}{n!}\cdot\dfrac{m(m+1)}{2}p_0, & \rho=1.\end{cases} \qquad (1.14)$$

На рис. 1.9 зображено графіки залежностей середньої кількості запитів, що очікують на обслуговування, від навантаження всієї СМО ρ при різних значеннях кількості каналів n і однаковій кількості місць очікування $m=7$.

Зрозуміло, що при $\rho\geq 0,5$ довжина черги різко збільшується; при зростанні n і однаковому значенні ρ величина L_q теж різко збільшується, а потім збільшення сповільнюється.

На рис. 1.10 зображено графік залежностей середньої кількості запитів, що очікують на обслуговування, від навантаження всієї СМО ρ при різних значеннях кількості місць очікування каналів n і однаковій кількості каналів $n=5$.

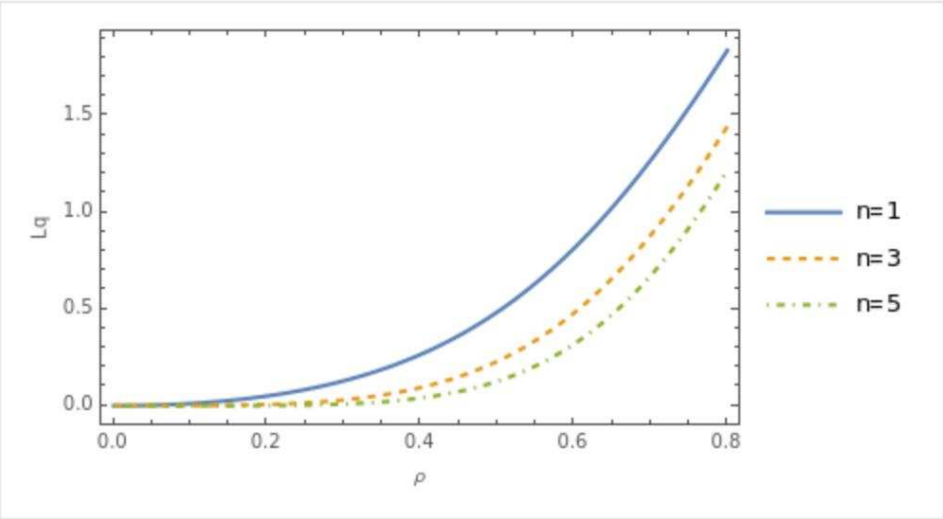

Рисунок 1.9 – Залежність L_q від завантаженості системи ρ при різних n

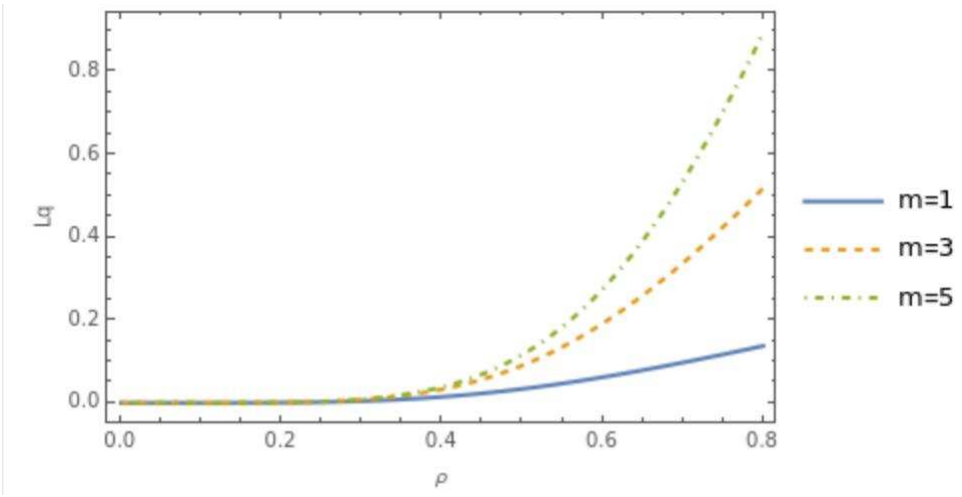

Рисунок 1.10 – Залежність L_q від завантаження системи ρ при різних m

Зрозуміло, що кількість місць очікування істотно впливає на довжину черги в системі. Зі зростанням m величина L_q також зростає. Це пояснюється тим, що менша кількість запитів отримує відмову щодо обслуговування. Окрім того, як і в попередньому випадку, при $\rho \geq 0,5$ довжина черги різко збільшується.

Визначимо $\overline{n}$ – середню кількість зайнятих обслуговуванням каналів, а відповідно, і середню кількість запитів, що обслуговуються. Кожен зайнятий канал обслуговує в середньому μ запитів за одиницю часу, а СМО загалом обслуговує A запитів. Поділивши A на μ, отримаємо $\overline{n}$:

$$\overline{n} = \frac{A}{\mu} = \lambda\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right) : \mu = n\rho\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right).$$

Звідси показник z_c – коефіцієнт завантаженості каналів системи:

$$z_c = \frac{\overline{n}}{n} = \rho\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right).$$

Середня кількість запитів щодо обслуговування та в черзі (у системі):

$$L_s = \overline{n}_z + L_q = n\rho\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right) + \frac{n^n}{n!}\frac{1 - \rho^m(1 + m - m\rho)}{(1-\rho)^2}\rho^{n+1} p_0,\ \rho \neq 1. \quad (1.15)$$

Середній час τ_q очікування запиту в черзі знайдемо за формулою Літтла $L_q = \lambda\tau_q$:

$$\tau_q = \frac{L_q}{\lambda} = \frac{L_q}{n\mu\rho}.$$

Підставивши (1.14) у (1.15), отримаємо:

$$\tau_q = \begin{cases} \dfrac{n^n}{\lambda n!}\dfrac{1 - \rho^m(1 + m - m\rho)}{(1-\rho)^2}\rho^{n+1} p_0, & \rho \neq 1, \\ \dfrac{n^n}{\lambda n!} \cdot \dfrac{m(m+1)}{2} p_0, & \rho = 1, \end{cases}$$

або

$$\tau_q = \begin{cases} \dfrac{n^{n-1}}{\lambda n!}\dfrac{1 - \rho^m(1 + m - m\rho)}{(1-\rho)^2}\rho^{n} p_0, & \rho \neq 1, \\ \dfrac{n^{n-1}}{\mu n!} \cdot \dfrac{m(m+1)}{2} p_0, & \rho = 1. \end{cases}$$

Середній час перебування запиту в системі τ_s для $\rho \neq 1$ також знайдемо за формулою Літтла $L_s = \lambda \tau_s$:

$$\tau_s = \frac{L_s}{\lambda} = \frac{L_s}{n\mu\rho}. \qquad (1.16)$$

Підставивши (1.15) у (1.16), отримаємо:

$$\tau_s = \frac{1}{\mu}\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right) + \frac{n^n}{n!}\frac{1-\rho^m(1+m-m\rho)}{\lambda(1-\rho)^2}\rho^{n+1} p_0,$$

або

$$\tau_s = \frac{1}{\mu}\left(1 - \frac{n^n}{n!}\rho^{n+m} p_0\right) + \frac{n^{n-1}}{n!}\frac{1-\rho^m(1+m-m\rho)}{\mu(1-\rho)^2}\rho^n p_0.$$

Якщо СМО $M/M/n/m$ моделює економічну систему, у якій обслуговуються запити, то для її оптимізації можна використати як критерій функцію E вартості збитків у системі за час T:

$$E = (C_{\text{прост.}}(1 - z_c)n + C_q L_q + \lambda C_{\text{зб.}} P_{\text{відм.}} + C_{\text{е}} z_c n)T,$$

де $C_{\text{прост.}}$ – вартість одиниці простоювання каналу;

C_q – вартість збитків від простоювання запиту в черзі за одиницю часу;

$C_{\text{зб.}}$ – вартість збитків від втрати запиту, який не був обслугований;

$C_{\text{е}}$ – вартість експлуатації кожного каналу системи за одиницю часу.

Як критерій можна також використовувати прибуток G від функціонування СМО:

$$G = \lambda C_s P_{\text{обсл.}} T - (C_{\text{прост.}}(1 - z_c)n + C_q L_q + \lambda C_{\text{зб.}} P_{\text{відм.}} + C_{\text{е}} z_c n)T,$$

де C_s – вартість обслуговування кожного запиту, тобто це валовий прибуток, отриманий під час обслуговування кожного запиту.

Оптимізацію можна проводити або за кількістю каналів обслуговування n, або за інтенсивністю обслуговування μ за умови, що цей параметр можна змінювати. Також можна призначати вартість обслуговування кожного запиту C_s, щоб прибуток G був не менше запланованого. Оптимізацію доцільно здійснювати чисельними методами [40].

1.7 Постановка задачі про оптимізацію систем масового обслуговування

1.7.1 Загальне формулювання задачі

Зазвичай, прикладні задачі, що описуються за допомогою систем масового обслуговування, накладають обмеження, пов’язані з фактичними можливостями масштабування. Більшість задач, що мають черги, не можуть дозволити собі зняти обмеження на кількість заявок, що перебувають в очікуванні. Також, зазвичай, не є можливим збільшувати чергу кожен раз, як вона переповнюється. Отже, набувають актуальності моделі з обмеженими чергами та відмовами у випадку переповнення. Такі СМО надають математичне обґрунтування та рекомендації до оптимізації процесів задля отримання кращої пропускної здатності. Тобто головна задача поставлена так: необхідно зменшити ймовірність відмови черговій заявці, що надходить до СМО, не змінюючи при цьому її структуру та основні характеристики.

Перед нами постає задача про збільшення пропускної здатності описаної системи задля зменшення загальної ймовірності відмови. При цьому, з точки зору результатів, для нас всі заявки мають однакове значення, незалежно від їхнього розміру та часу перебування в системі.

Розглядатимемо задачу про багатоканальну систему масового обслуговування із обмеженою чергою вимог та відмовою у випадку переповнення черги. У цій СМО до n однакових каналів обслуговування

надходить сім'я з r пуассонівських потоків заявок інтенсивності λ_j, $j = 1, 2, ..., r$. Отже, сукупний потік є пуассонівським з інтенсивністю:

$$\lambda_{\bar{r}} = \sum_{j=1}^{r} \lambda_j .$$

Якщо на момент приходу нової заявки є хоча б один вільний канал, він негайно починає обробку. У випадку, коли усі канали зайняті, вимога стає останньою до загальної черги ємності k. Заявки покидають чергу для подальшого обслуговування у тій самій послідовності, у якій вони надходили на очікування. Канал, що звільнюється від виконання, одразу починає обробку першої в черзі вимоги. При цьому кожна вимога обслуговується тільки одним каналом, і кожен канал може обслуговувати не більше однієї вимоги одночасно. У випадку, якщо вільними є декілька каналів обслуговування, для обробки буде обрано канал випадковим чином [43]. Час, необхідний на обробку однієї вимоги, є випадковою величиною з експоненціальним законом розподілу ймовірностей, та, загалом кажучи, відрізняється для різних потоків вимог:

$$F_j(x) = 1 - e^{-\nu_j x},\ \nu_j > 0,\ j = 1, 2, ..., r .$$

В даному випадку ν_j є коефіцієнтом складності задачі. Вимоги, що мають менше значення, будуть обслуговуватись швидше, а більші – довше. Існує безліч поширених практичних застосувань, що вписуються у дану модель, та яким притаманно мати задачі різної тривалості. Отже, задана система підпадає під умовне визначення $M/M/n/m$, де:

– перша M вказує на марковський вхідний потік вимог;

– друга M вказує на те, що процес обробки вимог також є марковським;

– n визначає, що система є багатоканальною, і задає кількість каналів;

– m описує обмежену ємність системи та визначає кількість місць для очікування, $m > 0$.

Кожен вхідний потік вимог має задовольняти наступні вимоги [43]:

– стаціонарність потоку;

– відсутність післядії;

– ординарність.

1.7.2 Ймовірнісний розподіл вхідного потоку

Функція розподілу загального часу обслуговування вимог потоків всіх розмірів має вигляд [44]:

$$F(x) = \frac{1}{\lambda} \sum_{j=1}^{r} \lambda_j F_j(x),$$

де $F_j(x)$ – функція розподілу часу обслуговування вимоги j-го розміру, що має середній час обслуговування $\frac{1}{\mu_j}$. Переривання обслуговування в досліджуваній системі не допускається. На даному етапі потоки заявок усіх розмірів мають рівний пріоритет і поєднуються в загальний потік вимог.

Ймовірність надходження нової вимоги довільного розміру до системи за проміжок часу t може бути визначена так:

$$p_i(t) = \frac{(t\lambda_{\bar{r}})^i}{i!} e^{-t\lambda_{\bar{r}}},$$

де $\lambda_{\bar{r}}$ – сукупна інтенсивність прибуття вимог усіх розмірів, що надходять до системи на одиницю часу;

i – кількість вимог, що присутні у системі, включно з наступною, в момент часу t.

Враховуючи обмеження на кількість вимог, що можуть очікувати на обслуговування у черзі, розрахуємо ймовірність відмови у виконанні вимоги через переповнення черги. Майбутнє протікання обслуговування, у контексті

теорії ймовірностей, не залежить від того, що відбувалось до моменту часу t_0 через особливості ймовірнісного розподілу [43, 44].

Інтенсивність навантаження j-ї категорії знайдемо з рівняння $\rho_j = \frac{\lambda_j}{n\nu_j}$, $j = 1, 2, ..., r$. Тоді загальне навантаження системи можемо розрахувати так:

$$R_{\text{заг.}} = \sum_{j=1}^{r} \rho_j, \; R_0 = 0, \; j = 1, 2, ..., r. \tag{1.17}$$

1.7.3 Ймовірність відмови

Для описаної системи ймовірність відмови дорівнює ймовірності знаходження у системі точно i вимог на момент надходження чергової заявки довільного розміру. Маємо

$$p_i = \begin{cases} \frac{n^i R_{\text{заг.}}^i}{i!} p_0, & 1 \le i \le n, \\ \frac{n^n R_{\text{заг.}}^i}{n!} p_0 & n < i \le n + m, \end{cases}$$

де

$$p_0 = \left[\sum_{i=0}^{n-1} \frac{n^i R_{\text{заг.}}^i}{i!} + \frac{n^n}{n!} \cdot \frac{R_{\text{заг.}}^n (1 - R_{\text{заг.}}^{m+1})}{1 - R_{\text{заг.}}} \right]^{-1}. \tag{1.18}$$

Розглядатимемо випадки, коли $R_{\text{заг.}} > n$, оскільки за такої умови забезпечується наповнення черги і відмова через брак місця для очікування. Отже, ймовірність відмови через переповнення черги може бути розрахована виразом

$$P_{\text{відм.}} = p_{n+m} = \frac{n^n R_{\text{заг.}}^{n+m}}{n!} p_0 . \quad (1.19)$$

Підставивши (1.18) у (1.19), отримаємо:

$$P_{\text{відм.}} = \frac{n^n R_{\text{заг.}}^{n+m}}{n!} \left[\sum_{i=0}^{n-1} \frac{n^i R_{\text{заг.}}^i}{i!} + \frac{n^n}{n!} \cdot \frac{R_{\text{заг.}}^n (1 - R_{\text{заг.}}^{m+1})}{1 - R_{\text{заг.}}} \right]^{-1} .$$

Даний вираз означає, що запит отримає відмову щодо обслуговування, якщо всі канали й місця очікування будуть зайняті. При цьому розбиття загального потоку заявок на категорії за розміром, саме по собі, не змінює загальну пропускну здатність системи.

1.8 Постановка задач дослідження

Виходячи з проведеного аналізу сучасного стану задачі моделювання систем масового обслуговування виникає необхідність у розробці сучасного ефективного методу та моделей оптимізації систем масового обслуговування з обмеженою чергою та відмовами.

Виходячи з цього, завданнями дисертаційної роботи є:

– дослідити задачу, до якої застосовуватимуться метод та моделі оптимізації, розрахувати для неї базові характеристики;

– дослідити існуючі методи оптимізації систем масового обслуговування;

– удосконалити метод оптимізації систем масового обслуговування в частині пріоритезації менших задач SJF (Shortest Job First), завдяки використанню комбінованої системи пріоритетів;

– удосконалити модель розбиття загального вхідного потоку заявок по категоріях за розміром в частині застосування поділу черги очікування за квотами;

– дослідити паралелізм та взаємодіюче співвиконання (concurrency) у комп’ютерному програмуванні;

– дослідити можливості сучасних мов програмування щодо реалізації взаємодіючого співвиконання та обрати технологію для розробки комп’ютерної програми;

– запропонувати інформаційну та комп’ютерну моделі аналізу процесів масового обслуговування із застосуванням взаємодіючого співвиконання;

– розробити програмне забезпечення для аналізу процесів масового обслуговування.

Висновки за розділом 1

1. Як свідчить аналіз літературних джерел, проблема моделювання систем масового обслуговування має глибоку історію дослідження. Тим не менш в площині оптимізації СМО з високою завантаженістю та технічними обмеженнями все ще існує потреба в інноваціях. Особливу увагу привертають задачі моделювання багатоканальних систем з обмеженою чергою та відмовами у випадку переповнення. Процеси, що підпадають під даний опис є поширеними та потребують в нових методах і моделях, спрямованих на зменшення ймовірності відмови. Існуючі дослідження пропонують різноманітні підходи до пріоритезації вхідного потоку заявок, однак в кожного з них є недоліки. Таким чином, аналізована задача є актуальною. Дане дослідження покликане запропонувати моделі, застосування яких буде збільшувати продуктивність СМО з обмеженнями, нівелюючи при цьому недоліки існуючих підходів.

2. Для моделювання систем масового обслуговування існує методологія класифікації. Розроблений математичний апарат добре описує та характеризує довільну систему, а також дозволяє прогнозувати поведінку за різних рівнів навантаження. Ймовірність відмови для систем з обмеженою чергою залежить від інтенсивності надходження заявок, функції ймовірнісного розподілу вхідного потоку, а також продуктивності обслуговування. Задля зменшення ймовірності відмови, за обмежених ресурсів, нам необхідно вдосконалювати

систему планування роботи. Для цього в подальшому, будемо використовувати систему пріоритетів.

3. Описано систему масового обслуговування з обмеженою чергою та можливістю відмов. Сформульовано математичну модель для досліджуваного типу систем.

4. Проаналізовано характеристики вхідного потоку, процесу обслуговування. Визначено ймовірнісний розподіл вхідного потоку, що відповідає пуассонівському процесу, забезпечуючи математичну основу для моделювання потоку заявок.

5. Розраховано ймовірність відмови системи при перевищенні обсягу черги, що дозволяє оцінити надійність системи за певних параметрів навантаження. В подальшому саме ймовірність відмови новоприбулій вимозі буде відігравати ключову роль в оцінці ефективності оптимізації. Побудовано математичну модель роботи системи на основі марковських процесів, що дозволяє визначити ймовірності станів системи та її ефективність.

Список джерел, які використано у даному розділі, наведено у повному списку використаних джерел [1 – 78, 84 – 87].

2 МЕТОД ПРІОРИТЕЗАЦІЇ МЕНШИХ ЗАДАЧ

2.1 Моделювання роботи досліджуваної системи

Розглянемо роботу чотирьох канальної СМО з обмеженою чергою та відмовами. Розглядатимемо її поведінку в моменти часу t_n, $n = 0, 1, 2, 3, \ldots$. В момент часу t_1 у системі відсутні заявки, усі 4 канали обслуговування знаходяться у стані очікування. Черга не має жодних вимог. Процес обробки розпочинається з надходження до системи заявки (рис. 2.1). Розмір цієї заявки відповідає часу, необхідному на її обробку, та у прикладі дорівнює XL (дуже велика).

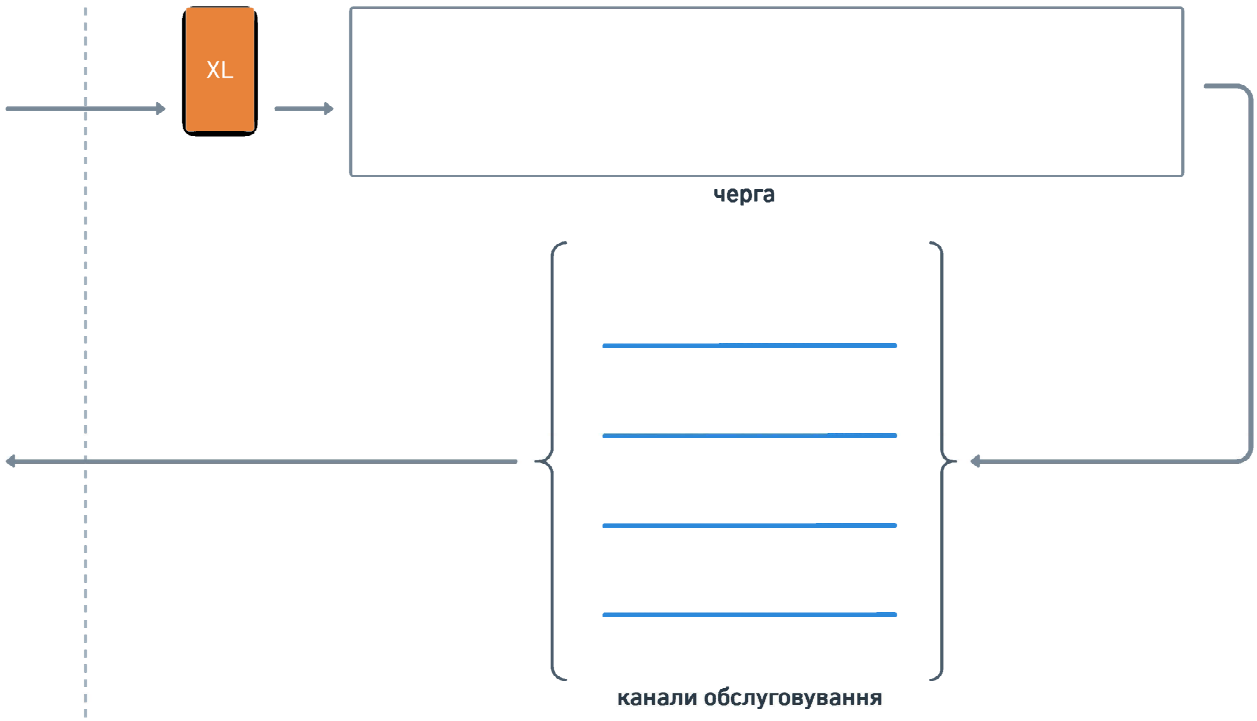

Рисунок 2.1 – Початковий стан системи

Оскільки в черзі є вільні місця, заявка може зайняти вільну позицію (рис. 2.2). Після чого одразу буде взята в роботу вільним каналом обслуговування (рис. 2.3). Аналогічно система поводитиметься наступні ітерації, допоки не закінчаться вільні канали обслуговування. На наступній

схемі зображено процес накопичення задач у черзі (рис. 2.4). Оскільки час виконання завдань різний, а також надходили вони на обробку в різний час, то і звільнюватись канали обслуговування будуть не одночасно. Наступним кроком, коли задача малого розміру завершила своє виконання, її місце займе наступна за чергою вимога (рис. 2.5). Для кращої аналогії задачі, що знаходяться у черзі, розташовані вертикально, щоб відобразити, що кожна з них, незалежно від складності, займає одне місце. Однак при переході до виконання вони розташовуються горизонтально, щоб підкреслити різницю у тривалості опрацювання.

Даний процес повторюється і система працює справно для випадків, коли інтенсивність надходження заявок менша за одиницю. В такому разі, черга грає роль буфера, який амортизує ситуації, коли заявок надходить більше звичайного. Окрема логіка передбачена для випадку, коли черга переповнена, та немає більше можливості розмістити в ній чергову заявку. Тоді вимозі буде відмовлено в обслуговуванні та вона покине систему (рис. 2.6).

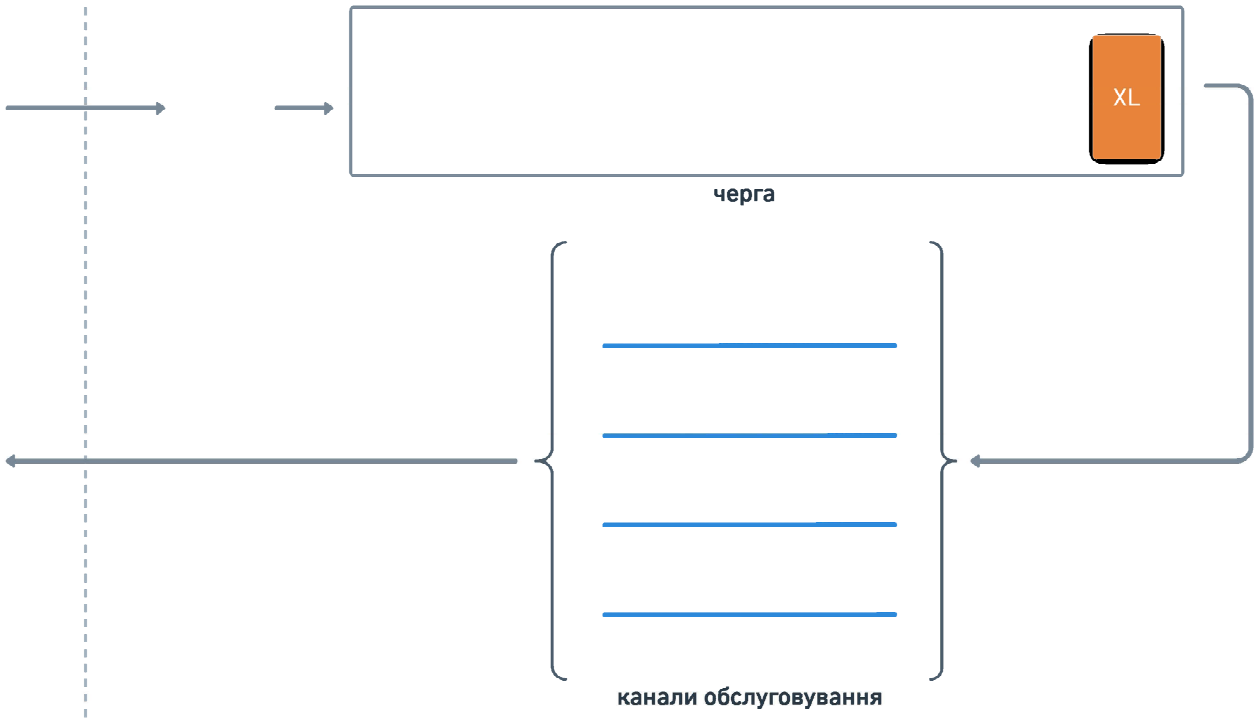

Рисунок 2.2 – Постанова вимоги до черги

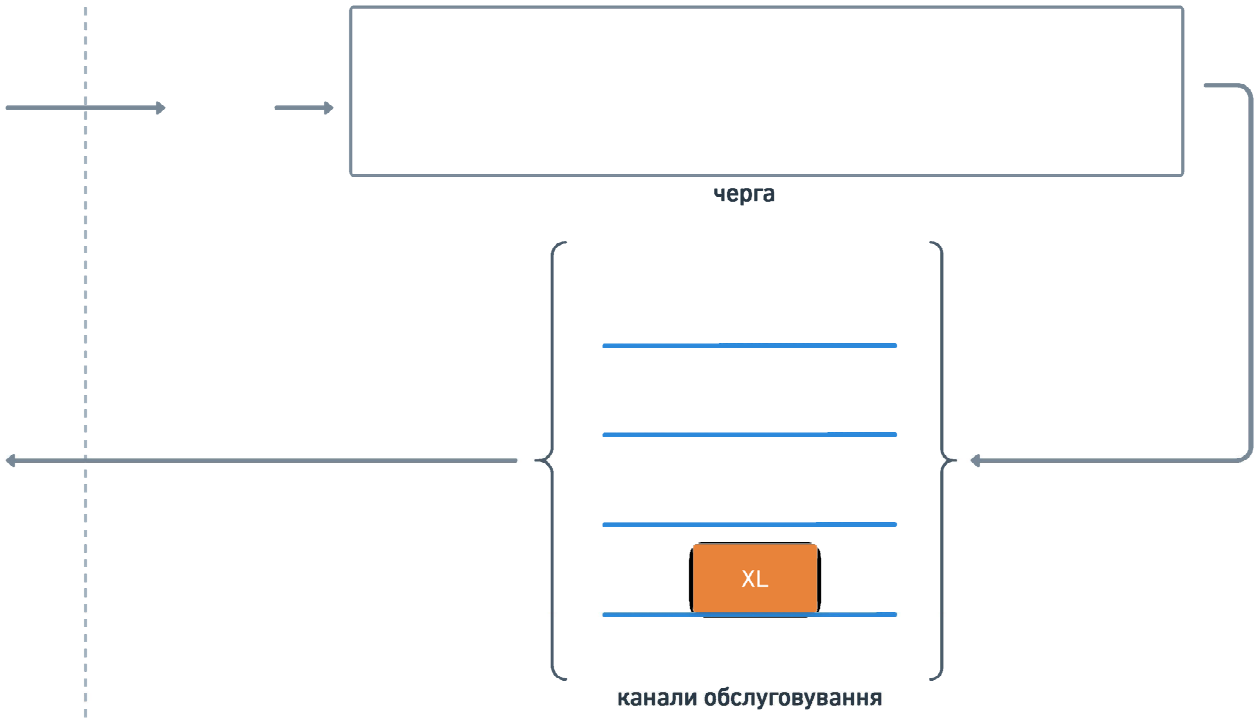

Рисунок 2.3 – Прийняття вимоги на обробку

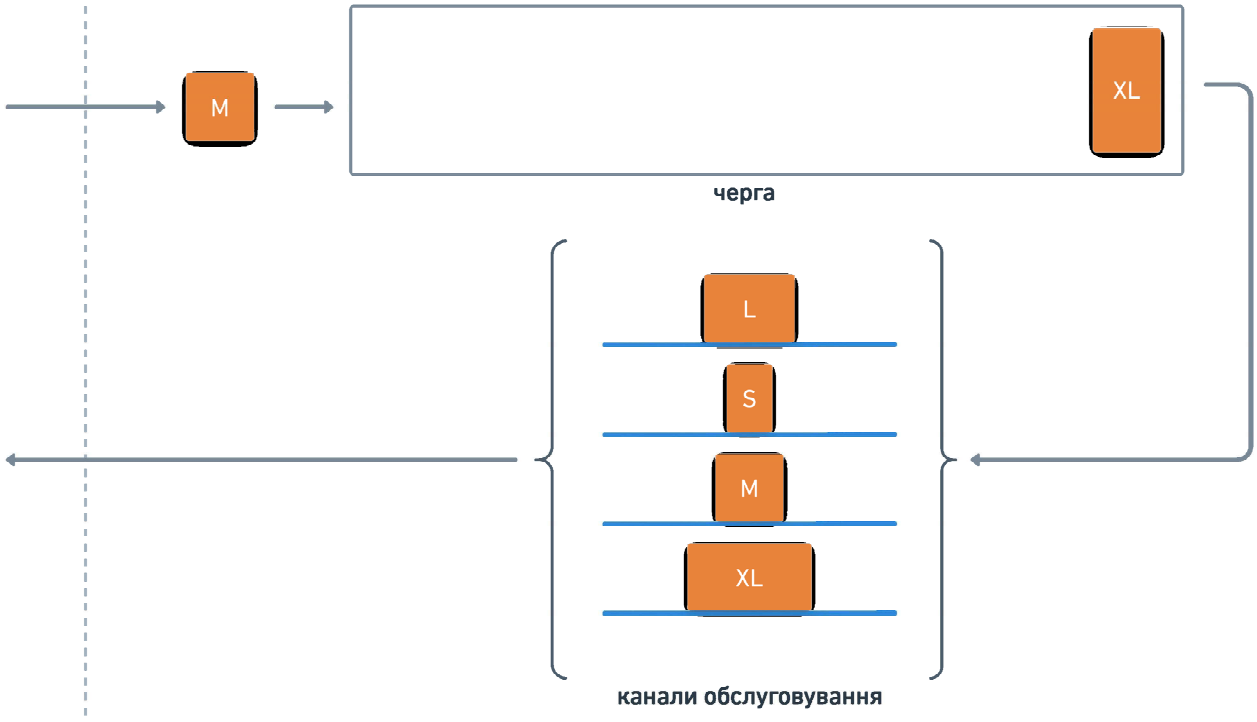

Рисунок 2.4 – Процес накопичення заявок у черзі

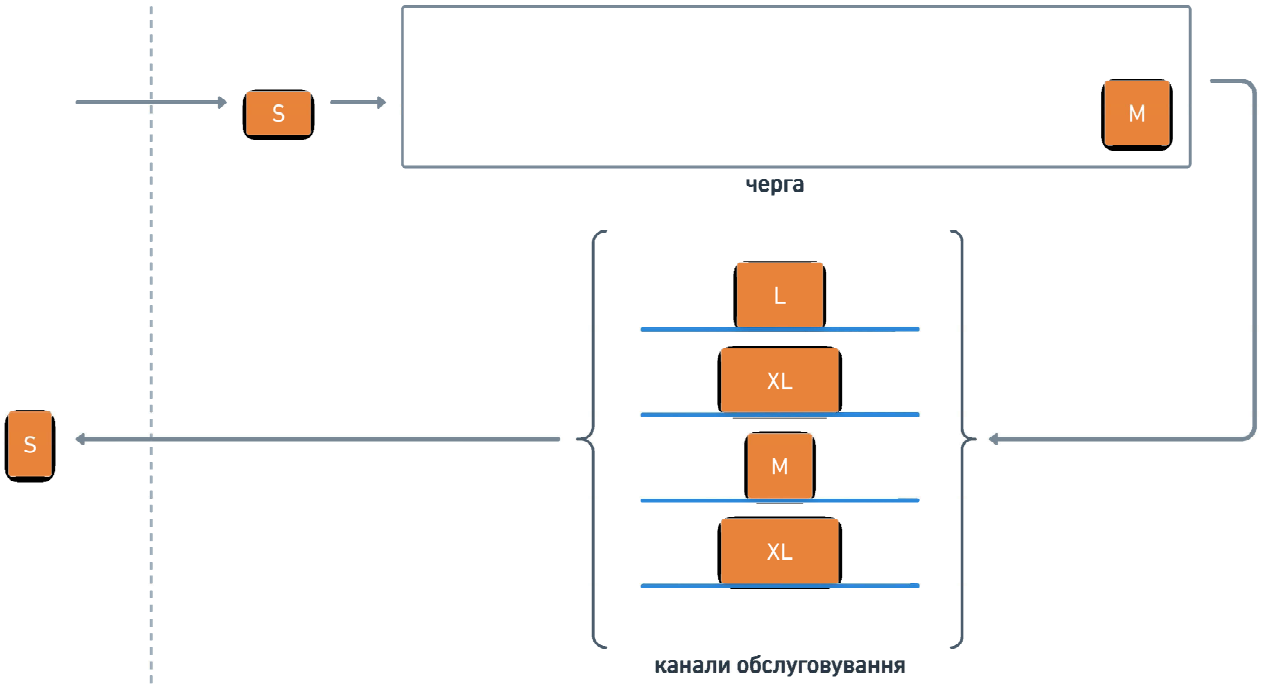

Рисунок 2.5 – Заміна успішно обробленої заявки

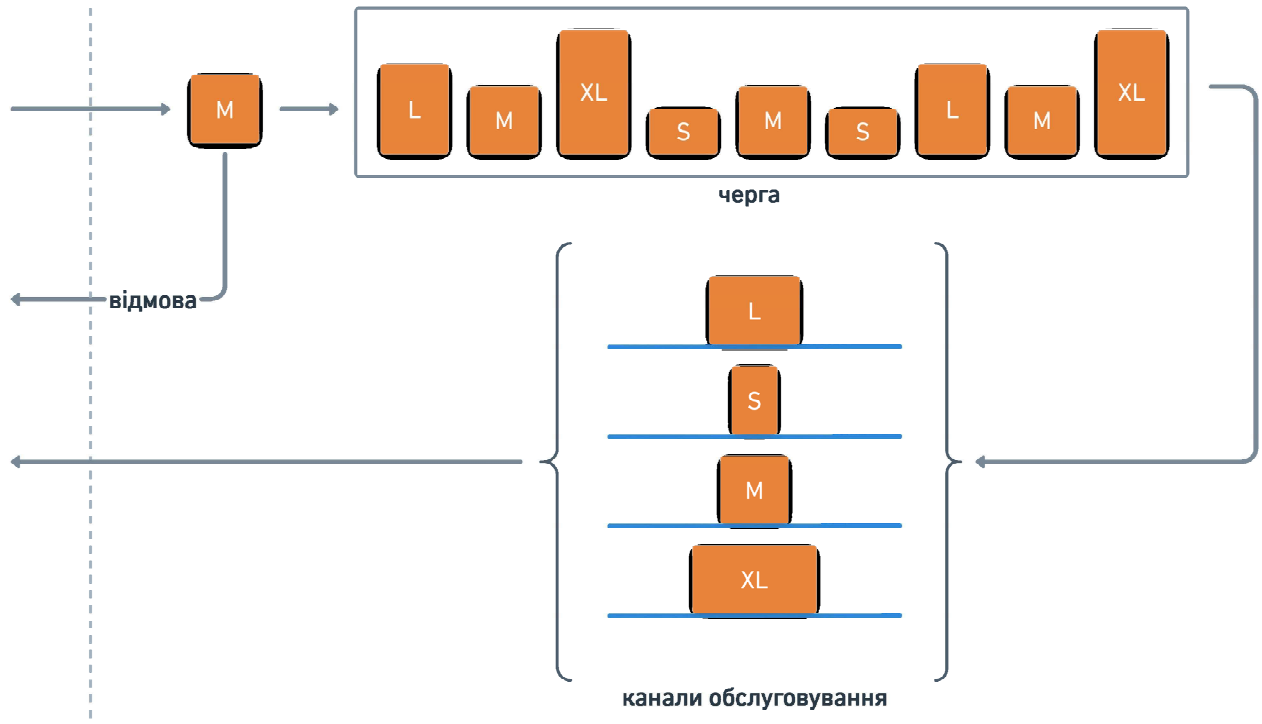

Рисунок 2.6 – Відмова в обслуговування через переповнення черги

Наша задача в рамках даного дослідження полягає у розробці методу та моделей призначених для оптимізації описаних систем.

2.2 Огляд підходів до оптимізації

Згідно з теорією масового обслуговування, для збільшення пропускної здатності системи зі сталим потоком вхідних заявок використовуються декілька підходів.

Перший підхід – збільшення кількості серверів (каналів обслуговування): додавання додаткових обслуговуючих станцій є найбільш прямолінійним рішенням для збільшення пропускної здатності [10]. Даний підхід є елементом горизонтального масштабування. Однак він має свою межу ефективності. Якщо вхідний потік заявок стабільний або малий, надмірна кількість серверів може призвести до простоїв і марного використання ресурсів. Також варто враховувати витрати на додаткове обладнання або персонал необхідний для розширення. Такі моделі систем масового обслуговування належать до типу $M/M/n$, де n – це кількість каналів. Задля якісного налаштування необхідно відстежувати, як збільшення кількості серверів впливає на час очікування і продуктивність досліджуваної системи. Наприклад, у телекомунікаційних системах додаткові оператори зменшують час очікування для клієнтів, але знижують ефективність, якщо завантаження нерівномірне. Окрім того, в деяких випадках, може існувати технічне обмеження на максимальну кількість каналів обслуговування.

Другий підхід – оптимізація розкладу та розподілу завдань: оптимальне розподілення заявок між серверами можна досягти через використання стратегій балансування навантаження або динамічного планування. Системи балансування навантаження динамічно спрямовують нові заявки на найменш завантажені сервери, що зменшує затори і підвищує ефективність. Наприклад, метод Round Robin рівномірно розподіляє навантаження між серверами по колу, тоді як Least Connection направляє заявки на сервери з найменшою кількістю поточних з’єднань. Ці підходи можуть значно покращити продуктивність у системах, де навантаження нерівномірне або нестабільне, наприклад, у хмарних сервісах або веб-серверах. Але для нашого випадку

моделювання СМО ми одразу припускаємо, що всі сервери беруть наступну задачу з загальної черги як тільки звільнюються від виконання попередньої заявки.

Третій підхід – зменшення часу обслуговування: оптимізація часу обслуговування дозволяє прискорити обробку кожної окремої заявки, що збільшує загальну пропускну здатність системи [9]. Цього можна досягти за рахунок оптимізації обладнання, оновлення або тонкого налаштування, що може значно покращити швидкість обробки. У комп'ютерних системах це може включати покращення мережевої інфраструктури або оновлення обладнання серверів для швидшого оброблення запитів. Також можна працювати над оптимізацією самої логіки обслуговування, наприклад, замінивши алгоритм на продуктивніший. Такий підхід завжди буде доречним, але він стосується безпосередньо заявки, що моделюється, і його неможливо екстраполювати на довільну задачу.

Четвертий підхід – розподіл завдань: подрібнення задач на складові може дозволити їх швидше виконати за допомогою паралельної обробки. Це також знижує затримки, пов'язані з довготривалими операціями. У випадку виникнення помилки обслуговування, ми втрачатимемо менше часу на повторну спробу, оскільки загальна тривалість обслуговування однієї операції знизиться.

П'ятий підхід – удосконалення управління чергою: вдосконалення систем управління чергою дозволяє ефективніше використовувати наявні ресурси та зменшити час очікування для клієнтів. Дані підходи працюють для систем з чергою і особливо корисні у випадку відмов через переповнення черги або перевищення часу перебування заявки у системі. Ця сім'я підходів включає в себе різні політики пріоритетів заявок. Системи з пріоритетами забезпечують покращення загальної ефективності системи незалежно від фактичної логіки обслуговування.

Шостий підхід – паралелізація процесів: паралелізація дозволяє обробляти кілька частин одного процесу одночасно. Це особливо корисно у

системах, де операції не залежать одна від одної і можуть виконуватися окремо. Для реалізації цього підходу часто використовуються асинхронні обчислення або багатопотоковість. Наприклад, у програмному забезпеченні для обробки зображень можна паралельно обробляти різні частини зображення, що значно знижує час очікування для користувачів. Такий підхід має бути застосовано за замовченням, і в подальшому ми будемо працювати над збільшенням продуктивності системи, враховуючи паралельну обробку заявок.

Кожен з цих підходів може бути використаний у поєднанні з іншими для досягнення максимальної продуктивності та ефективного управління ресурсами системи.

Як вже було зазначено у попередньому розділі, ми вирішуємо задачу оптимізації багатоканальної системи масового обслуговування з обмеженою чергою очікування та відмовами у випадку її переповнення. При цьому канали обслуговування працюють паралельно, а їх кількість обмежена. Рішення не має залежати від конкретної логіки обслуговування та має збільшувати пропускну здатність будь-якої системи, що задовольняє критеріям функціонування. Подальшим напрямком дослідженням буде впровадження системи пріоритетів для заданої СМО.

2.3 Системи з пріоритетами

2.3.1 Різновиди пріоритетів

Пріоритети у системах масового обслуговування є важливим інструментом для забезпечення оптимального функціонування цих процесів, особливо коли мова йде про обробку типів запитів з різними рівнями важливості. Основна суть систем пріоритетів полягає в тому, щоб надати певним завданням або клієнтам вищий рівень доступу до каналів обслуговування, дозволяючи обробляти важливіші заявки раніше. Це особливо актуально в умовах обмежених ресурсів, коли система не може одночасно

задовольнити всі запити через високе навантаження або затримки в обробці.

Необхідність впровадження пріоритетів виникає у ситуаціях, де час обслуговування або затримка мають критичне значення для деяких заявок. Наприклад, у медичних установах системи масового обслуговування з пріоритетами забезпечують швидку реакцію на екстрені випадки, тоді як менш критичні пацієнти можуть бути обслуговані пізніше. Аналогічно у сфері телекомунікацій або IT-підтримки запити від важливих клієнтів або критичних систем можуть мати вищий пріоритет і оброблятися швидше, ніж стандартні запити. Такий підхід допомагає збалансувати навантаження і зменшити ризики негативних наслідків для критичних процесів.

У навантажених системах неминучим є явище загибелі (starvation) завдань через надмірне очікування або відсутність місць в черзі. В такому випадку кажуть, що заявці було відмовлено в обслуговуванні. Системи пріоритетів також дозволяють обирати, які саме вимоги будуть частіше отримувати відмови. Таким чином, можливий сценарій коли низькопріоритетні завдання можуть ніколи не отримати доступу до ресурсу через постійний прихід високопріоритетних запитів, але при цьому важливі задачі будуть стабільно обслуговуватись. Для уникнення ситуацій, коли система починає обробляти виключно важливі запити, використовуються механізми динамічного коригування пріоритетів. За такої умови пріоритет завдань підвищується з часом, щоб гарантувати їх обробку. Це дозволяє підтримувати баланс між швидким обслуговуванням критичних запитів та забезпеченням доступу до ресурсів для менш важливих завдань.

Загалом, пріоритети у системах масового обслуговування забезпечують ефективніше управління ресурсами, допомагають оптимізувати обслуговування клієнтів та мінімізувати затримки для критичних завдань. Вони є необхідними в багатьох галузях, де час реакції або обмежені ресурси накладають свої обмеження на роботу системи.

Розрізняють наступні види пріоритетів:

– абсолютні;

– відносні;

– статичні;

– динамічні;

– на основі типу задач;

– на основі часу обслуговування;

– з часовими квотами.

Умовно їх можна поділити на дві категорії: базові та удосконалені (рис. 2.7). При виборі типу пріоритетів для системи ми маємо обрати один з базових різновидів та за необхідністю додати комбінацію з удосконалень. При цьому удосконалення можна застосовувати до кожного з трьох базових різновидів. Основна ідея базових різновидів полягає у визначені правил, що діють до пріоритезованих заявок, а вдосконалені різновиди уточнюють підходи до надання пріоритетів вимогам.

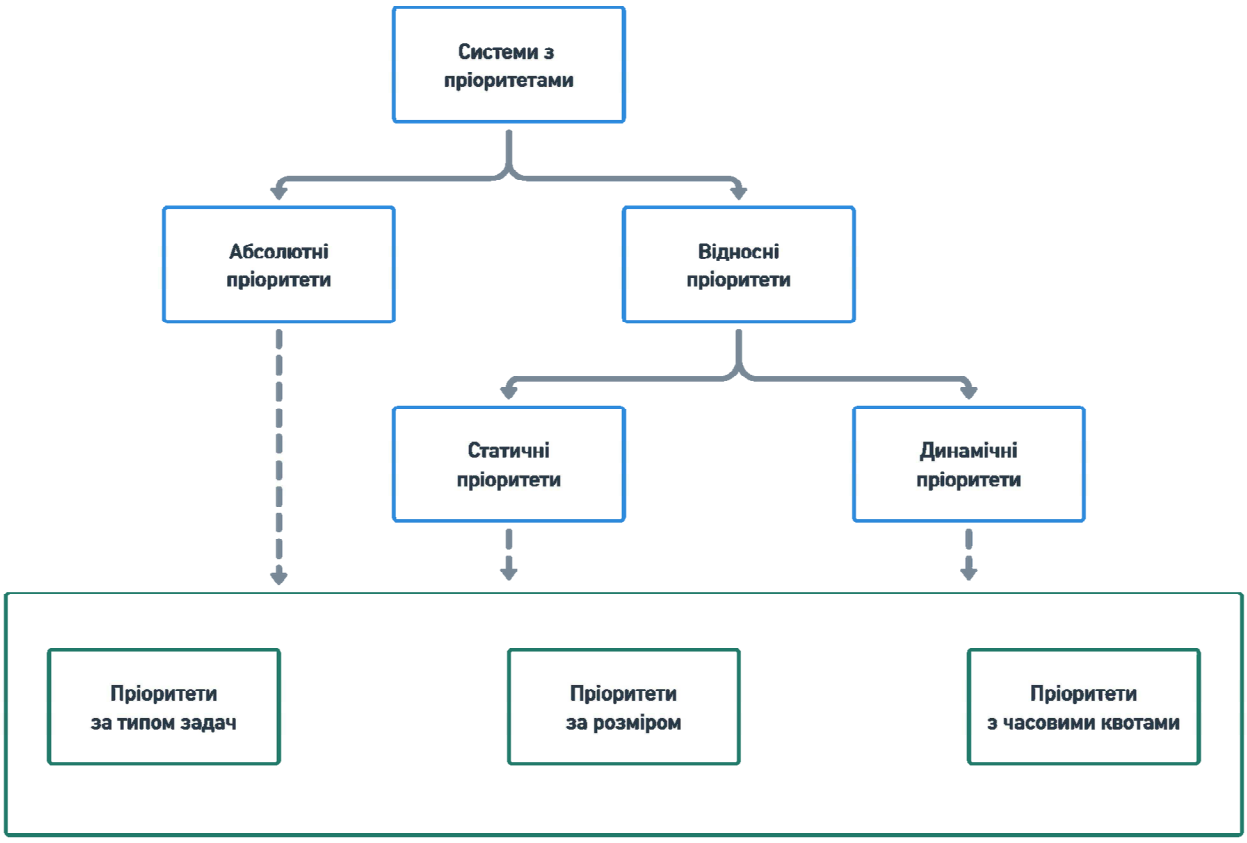

Рисунок 2.7 – Схема різновидів пріоритетів

2.3.2 Базові пріоритети

Абсолютні пріоритети є специфічним типом управління чергами в СМО, де заявки або завдання з вищим пріоритетом завжди отримують перевагу над іншими. Це означає, що завдання з найвищим пріоритетом обробляється негайно, навіть якщо в черзі чи на виконанні є інші завдання, які надійшли раніше [1]. За необхідності задача з високим пріоритетом може перервати обробку задачі, що є менш важливою. Пріоритети залишаються незмінними протягом усього життєвого циклу завдання, і система завжди обробляє більш важливі запити раніше за всі інші незалежно від часу їх надходження.

Модель абсолютних пріоритетів ґрунтується на строгому розподілі пріоритетів серед завдань або заявок. Кожній заявці присвоюється певний пріоритетний ранг, який залишається постійним, і черга завжди обробляється відповідно до цього рангу. Це реалізується шляхом організації черги в системі, де завдання з найвищим пріоритетом знаходяться на початку черги, а з нижчим – на її кінці. У разі надходження нового завдання з вищим пріоритетом воно автоматично переміщується на перше місце в черзі незалежно від поточного стану інших завдань [5].

Абсолютні пріоритети особливо корисні в системах, де є критичні завдання, що потребують негайної реакції. Прикладом такої системи може бути система управління повітряним рухом, де запити від літаків, що знаходяться в аварійній ситуації, повинні оброблятися негайно, навіть якщо інші запити перебувають у черзі на обробку. Для реалізації таких пріоритетів використовуються структури даних, як-от пріоритетні черги (priority queues) або купи (heaps), що забезпечують доступ до найвищого пріоритету в постійному часі.

Одним з основних недоліків використання абсолютних пріоритетів є потенційна проблема голодування (starvation) завдань з низьким пріоритетом, які можуть бути ніколи не оброблені, оскільки нові високопріоритетні завдання постійно надходять у систему і займають ресурси [2]. У результаті завдання з

низьким пріоритетом можуть залишатися в черзі невизначений час, що може бути неприпустимим у деяких системах, особливо там, де всі запити мають бути оброблені в межах певного часу. Також у випадку переривання виконання низькопріоритетної задачі задля того, щоб поступитись терміновій вимозі, ми втрачаємо час витрачений на частину обробки попередньої заявки або необхідно додавати логіку поновлення виконання, що у свою чергу, ускладнить алгоритм СМО.

Абсолютні пріоритети знаходять широке застосування у критичних системах реального часу. В таких системах час реакції на завдання критично важливий і від цього може залежати успішне виконання системних операцій. Абсолютні пріоритети забезпечують найкоротший шлях до обробки найважливіших запитів, однак потребують ретельного управління для уникнення негативних ефектів, таких як голодування інших задач.

Відносні пріоритети в СМО є гнучкою моделлю управління чергами, де завдання з вищим пріоритетом отримують більше ресурсів або частіше обслуговуються, але завдання з нижчим пріоритетом також мають можливість бути обробленими. Відмінністю від абсолютних пріоритетів є те, що низькопріоритетні завдання не переривають своє виконання у випадку надходження важливішої заявки [4]. Ця модель забезпечує кращий баланс між обробкою критично важливих запитів та зменшенням затримок для менш пріоритетних завдань, ніж за абсолютних пріоритетів. Даний різновид поєднує низку інших типів пріоритетів та потребує додаткових мір задля збільшення ефективності.

У системах із відносними пріоритетами завданням присвоюється числовий пріоритет, але на відміну від абсолютних систем, пріоритет впливає не на негайну обробку завдання, а на частоту або ймовірність його виконання. Це може бути реалізовано через таймшери (time-sharing), коли завдання з високим пріоритетом отримують більший відсоток часу процесора або доступу до ресурсів. Наприклад, у багатопотокових системах планувальник може виділяти 70% часу для завдань з високим пріоритетом і 30% для завдань з

низьким пріоритетом.

Також можлива реалізація через ймовірнісні підходи, де завдання з високим пріоритетом мають більше шансів бути вибраними для обслуговування під час кожної ітерації планувальника [7]. Це дозволяє системі зберігати рівновагу між швидким виконанням критичних задач і забезпеченням обслуговування для менш важливих завдань.

Алгоритми, що використовуються для реалізації відносних пріоритетів, включають:

– Round Robin з вагами: кожне завдання отримує частку часу або ресурсів відповідно до свого пріоритету;

– Lottery Scheduling: завдання отримують «квитки», кількість яких залежить від їхнього пріоритету, і завдання з більшою кількістю квитків мають більше шансів бути обраними для виконання під час кожної ітерації.

Однією з основних переваг відносних пріоритетів є те, що вони забезпечують кращий баланс між пріоритетами [37]. Завдяки цьому завдання з низьким пріоритетом не залишаються в черзі на невизначений час, як це може статися у системах з абсолютними пріоритетами. Однак однією з потенційних проблем є складність налаштування таких систем, оскільки неправильне балансування може призвести до надмірного навантаження на систему або недостатнього обслуговування низькопріоритетних завдань.

У цілому, відносні пріоритети дозволяють створювати більш адаптивні системи, що здатні забезпечувати конкурентне виконання різних типів завдань, водночас підтримуючи загальну ефективність і стабільність. Вони поділяються на статичні та динамічні пріоритети.

Статичні пріоритети є одним з найпростіших і найбільш розповсюджених підходів призначених для управління чергою завдань у системах масового обслуговування. Їх основна ідея полягає в тому, що кожному завданню або клієнту присвоюється фіксований пріоритет, який залишається незмінним протягом усього життєвого циклу задачі [39]. Цей пріоритет визначає порядок обробки заявок у черзі: завдання з вищим пріоритетом обслуговуються

першими незалежно від того, коли вони надійшли в систему. Така модель використовується для забезпечення швидкої обробки важливих запитів у системах з обмеженими ресурсами. У подібних системах кожному завданню присвоюється числовий показник, що відображає його важливість відносно інших завдань. Завдання з нижчим числовим значенням зазвичай мають вищий пріоритет. При обробці черги система спочатку перевіряє, чи є завдання з найвищим пріоритетом. Якщо такі є, вони обробляються першими, а всі інші завдання чекають на звільнення ресурсів. Після завершення обробки всіх високопріоритетних завдань система переходить до обробки завдань з нижчим пріоритетом. Це забезпечує чіткий порядок обслуговування.

Хоча система статичних пріоритетів ефективно розподіляє ресурси для критичних завдань, вона має певні недоліки [45]. Наприклад, завдання з низьким пріоритетом можуть залишатися у черзі тривалий час, якщо постійно надходять високопріоритетні завдання. Це може призвести до відмов загибелі завдань (starvation), коли деякі завдання ніколи не отримають ресурсів для обробки.

Статичні пріоритети найчастіше застосовуються в системах реального часу, де важливо, щоб критичні завдання виконувалися без затримок. Наприклад, у системах управління медичним обладнанням або авіаційних системах керування повітряним рухом використовується фіксована пріоритетна схема, щоб забезпечити обробку найважливіших завдань, таких як життєво важливі сенсори або навігаційні сигнали.

У системах, що працюють з великими обсягами даних, наприклад, у базах даних, статичні пріоритети можуть обмежено використовуватися для того, щоб забезпечити швидший доступ до даних для критичних запитів, таких як транзакції VIP-клієнтів або завдання з обмеженим часом виконання.

Динамічні пріоритети є одним з базових механізмів управління чергами завдань у СМО. Вони дозволяють системі змінювати пріоритет завдань у процесі їх виконання, що допомагає забезпечити більш гнучке управління ресурсами та уникнути проблем, таких як загибель завдань (starvation) однієї

категорії [38]. Основна відмінність динамічних пріоритетів від статичних полягає в тому, що пріоритети можуть змінюватися залежно від таких факторів, як час очікування, стан системи або специфічні умови для кожної задачі.

Динамічні пріоритети можуть базуватися на кількох параметрах, що визначають, коли і як змінюється пріоритет завдання [53]. Один із найбільш поширених підходів полягає в підвищенні пріоритету завдань, які перебувають у черзі протягом тривалого часу. Це дозволяє уникнути ситуацій, коли нові завдання з високим пріоритетом постійно займають ресурси, залишаючи старі завдання невиконаними.

Механізм динамічних пріоритетів часто включає алгоритми з балансуванням між тривалістю очікування та початковим пріоритетом завдання. Один із таких алгоритмів – AGING (старіння), який полягає у поступовому підвищенні пріоритету завдання в міру збільшення часу його очікування в черзі [44]. Завдяки цьому вдається уникнути ситуації, коли низькопріоритетні завдання ніколи не обробляються, що є типовою проблемою в системах зі статичними пріоритетами.

Динамічні пріоритети потребують ретельного налаштування для уникнення надмірних затримок і небажаного збільшення часу обробки критичних завдань [13]. Наприклад, якщо пріоритети змінюються занадто швидко або занадто часто, це може призвести до зворотної проблеми – важливі завдання можуть не отримати достатньо ресурсів, оскільки інші завдання також швидко підвищують свої пріоритети. Тому в деяких системах використовується механізм квот часу, коли завдання можуть підвищувати свої пріоритети до певного рівня, після чого їхня обробка стає обов’язковою незалежно від інших чинників.

Інші підходи можуть використовувати плаваючі пріоритети, де система оцінює пріоритети завдань залежно від зовнішніх факторів, таких як поточне навантаження на систему або доступність ресурсів. Це дозволяє гнучкіше реагувати на змінні умови в системах із нестабільним навантаженням, наприклад, у хмарних обчисленнях або веб-серверах із великим потоком запитів.

2.3.3 Вдосконалені пріоритети

Пріоритети на основі типу задачі є моделлю управління чергами у системах масового обслуговування, де кожній задачі або запиту присвоюється пріоритет залежно від його природи та важливості для загальної роботи системи. Така система дозволяє забезпечити оптимальне розподілення ресурсів між різними типами задач, що мають різні вимоги до часу виконання, критичності та інтенсивності [47]. Це важливо для забезпечення високої продуктивності системи та гарантування обслуговування найбільш важливих процесів у першу чергу.

У системах з пріоритетами на основі типу задачі кожен тип процесу або запиту має визначений пріоритет, який впливає на те, як ці задачі будуть оброблятися. Технічно це реалізується через організацію черги, в якій завдання сортуються за їх типом. Під час планування задач система перевіряє тип кожної заявки і визначає, які процеси мають бути оброблені в першу чергу. Наприклад, у багатозадачних операційних системах типи задач можуть бути класифіковані як системні процеси, інтерактивні процеси та фонові процеси, де системні завдання обробляються з найвищим пріоритетом.

Правильне налаштування системи з пріоритетами на основі типу задачі є важливим, щоб уникнути перевантаження системи критичними завданнями і забезпечити обробку менш важливих задач [9]. Неправильна конфігурація може призвести до ситуації, коли менш пріоритетні задачі отримують занадто мало ресурсів або взагалі не обробляються. Це може бути критичним у системах, де всі типи задач повинні бути оброблені в межах певного часу, як в реальних застосунках або телекомунікаційних системах.

Для вирішення цієї проблеми часто використовують динамічне коригування пріоритетів або застосування різних таймшерігових алгоритмів, які виділяють різні часові квоти для кожного типу задач. Наприклад, системні процеси отримують більший відсоток часу процесора, але інші типи задач також мають можливість отримати доступ до ресурсів.

Пріоритети на основі типу задачі широко використовуються в операційних системах, базах даних, хмарних сервісах, виробництвах та ін. У базах даних, веб-сервісах цей підхід дозволяє оптимізувати доступ до даних, де запити на читання можуть отримувати пріоритет над запитами на запис.

Пріоритети на основі часу обслуговування є політикою у СМО, за якої завданням надається пріоритет відповідно до передбачуваного або фактичного часу, необхідного для їх обробки. Метою цього підходу є мінімізація середнього часу очікування для всіх запитів, зокрема, за рахунок більш швидкої обробки коротших завдань [25]. Цей підхід може бути особливо ефективним у системах із великим потоком запитів, де швидке обслуговування коротких завдань дозволяє швидше звільнити ресурси для обробки інших задач [46]. Як правило, задачі з коротшим часом обслуговування отримують вищий пріоритет, оскільки вони можуть бути оброблені швидше і допомагають швидше «очистити» чергу. Це досягається шляхом використання алгоритмів, таких як Shortest Job First (SJF) або Shortest Remaining Time First (SRTF), які динамічно обирають завдання з найкоротшим часом виконання.

Важливо зазначити, що для цього підходу потрібна наявність точної інформації про час обслуговування завдань, що може бути складно передбачити в деяких системах. Для вирішення цієї проблеми використовуються різні оцінки або моделі передбачення, що базуються на історичних даних або поточних умовах [10]. Наприклад, у системах обробки запитів до баз даних час обробки кожного запиту може бути оцінений на основі розміру запиту або алгоритмічної складності операцій.

Одним з основних викликів при використанні пріоритетів на основі часу обслуговування є ризик голодування довгих завдань. Якщо в системі постійно надходять короткі запити, довші завдання можуть бути відкладені на невизначений час, оскільки система завжди буде надавати перевагу коротшим завданням [28]. Для уникнення цієї проблеми часто використовують модифіковані алгоритми, такі як зважене обслуговування або комбіновані підходи, де більш довгі завдання отримують пріоритет у разі тривалого

очікування.

Крім того, ефективність системи з таким підходом значно залежить від точності передбачень часу обслуговування [11]. У випадках, коли час обслуговування важко передбачити або він сильно варіюється, система може функціонувати неефективно. Тому в реальних застосуваннях пріоритети на основі часу обслуговування часто комбінуються з іншими моделями планування для досягнення кращої продуктивності.

Пріоритети на основі часу обслуговування знаходять застосування в системах, де є значні варіації у тривалості виконання завдань. Це можуть бути системи обробки запитів до баз даних, де невеликі запити можуть оброблятися дуже швидко, або веб-сервіси, де короткі запити користувачів (наприклад, завантаження веб-сторінки) можуть мати вищий пріоритет порівняно з великими файлами або складними транзакціями. Цей підхід також часто використовується у сфері телекомунікацій для управління трафіком даних, де короткі пакети можуть бути оброблені швидше, забезпечуючи кращу продуктивність мережі.

Пріоритети за часовими квотами є моделлю управління ресурсами у СМО, де завданням або процесам надаються певні часові квоти для їх обслуговування. Ця модель дозволяє збалансувати навантаження між завданнями різного пріоритету, виділяючи кожному завданню обмежену кількість часу на виконання, після чого ресурс передається наступному завданню. Часова квота може змінюватися залежно від пріоритету завдання: більш важливі процеси отримують більше часу на виконання, тоді як менш пріоритетні процеси отримують менші квоти.

Система планування на основі часових квот зазвичай працює з використанням алгоритмів кругового планування (Round Robin) або його модифікацій. У такій системі кожне завдання отримує певну квоту часу (timeslot) на обробку, після чого його виконання припиняється, і наступне завдання отримує свою квоту часу [27]. Якщо завдання не завершилося протягом виділеного вікна можливості, воно знову стає в чергу для очікування

наступної квоти. Більш пріоритетні завдання можуть отримувати часові квоти для обслуговування частіше або більші за тривалістю, що дозволяє таким вимогам швидше завершити своє виконання.

Однією з основних переваг системи планування за часовими квотами є її умовна справедливість: всі завдання отримують доступ до ресурсів, навіть якщо вони мають нижчий пріоритет. Це дозволяє уникнути проблеми загибелі завдань (starvation), які належать до однієї групи, що виникає в інших моделях, таких як абсолютні пріоритети. Однак, правильно налаштувати систему квот може бути складно, оскільки надто короткі квоти можуть призвести до частих перемикань контексту, що негативно впливає на продуктивність системи [15]. З іншого боку, занадто великі квоти можуть порушити умовну справедливість, надаючи перевагу одній категорії процесів.

Окрім того, для таких систем необхідно подбати про механізми відновлення виконання вимог. Постійне переривання та перемикання контексту негативно впливають на ефективність використання ресурсів. Також у системах з різко варіативними завданнями, у яких деякі процеси можуть вимагати значно більше часу для виконання, необхідно враховувати, що навіть з низькими пріоритетами такі завдання повинні отримувати більші часові квоти, щоб забезпечити можливість їх завершення. Тому в таких системах часто застосовують комбіновані підходи, де часові квоти можуть коригуватися в залежності від типу завдань або їх критичності.

Пріоритети за часовими квотами часто використовуються в багатозадачних операційних системах, де ресурси обмежені і система повинна умовно справедливо розподіляти час процесора між кількома завданнями. Це також корисно в реальному часі, де критично важливо, щоб кожне завдання отримало гарантований доступ до ресурсів. Наприклад, у системах обслуговування клієнтів, де важливо надати відчуття турботи (школи, садочки, лікарні). В такий спосіб жоден з відвідувачів не буде почувати себе покинутим чи проігнорованим.

2.4 Метод оптимізації систем масового обслуговування із застосуванням пріоритезації менших задач

Встановлення пріоритетів для вимог, що очікують, – один із ефективних способів керування розмірами черги та часом перебування в ній [45]. Для систем з пріоритетами всі вимоги поділяються на категорії, а заявки більш високої категорії при обслуговуванні мають певні переваги перед вимогами з нижчим пріоритетом. Структура початкової системи зображена на рис. 2.8.

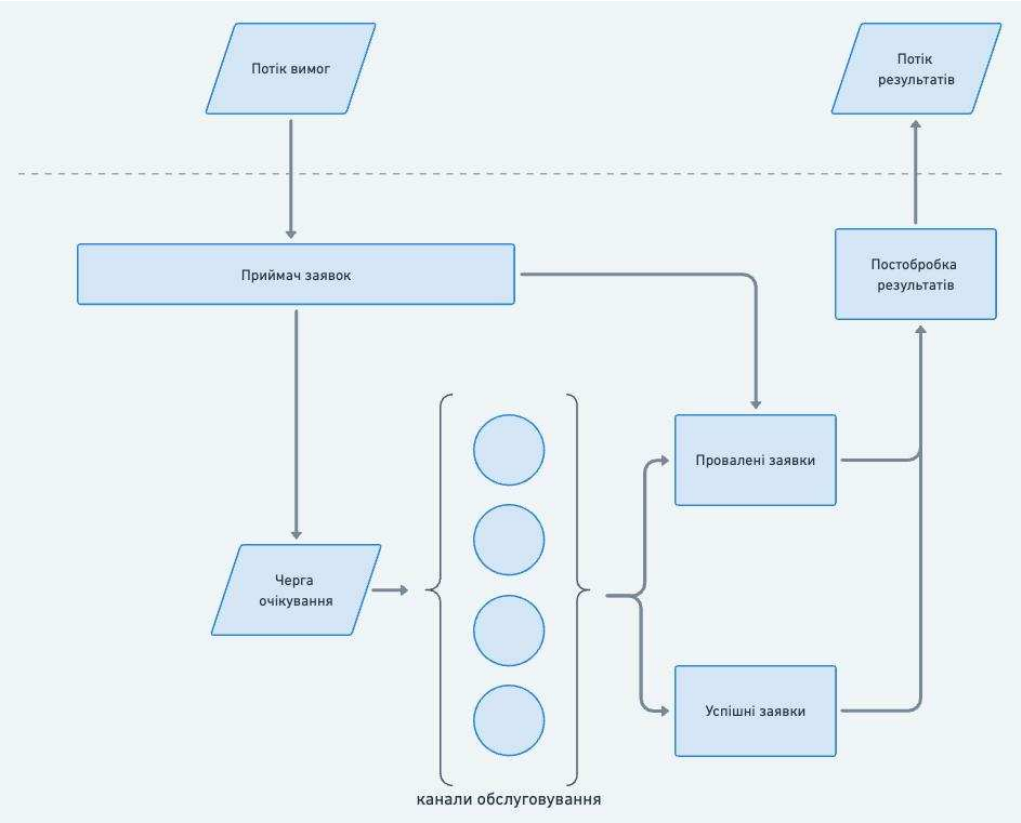

Рисунок 2.8 – Початковий алгоритм обробки заявок

Враховуючи всі особливості поставленої задачі, серед усіх варіантів у повній мірі нас не задовольняє жодний з різновидів пріоритетів. Однак, дані типи можна поєднувати, тим самим, нівелюючи недоліки одних перевагами інших. Цільова схема є комбінованою і є гібридом наступних типів:

– пріоритет на основі часу обслуговування;

– пріоритет за класами вимог;

– динамічний пріоритет;

– відносний пріоритет.

За рахунок використання пріоритетів на основі часу обслуговування ми зможемо пропускати вперед менші задачі, що у свою чергу має зменшити навантаження на чергу. При цьому за розміри задач будемо співвідносити їх до певних категорій (класів). Проблему загибелі великих завдань вирішуватимемо за рахунок застосування динамічних пріоритетів. Таким чином, ми будемо надавати шанс на виконання для складних задач, навіть якщо існують пріоритетні вимоги, тобто наявність заявок вищої категорії в черзі не забороняє взяття в роботу менш важливої задачі, а тільки надає додаткових преференцій при виборі. І, нарешті, ми не будемо переривати виконання жодних задач, адже кожне переривання означає втрату часу, що вже було витрачено на обробку заявки. Наша мета – максимально ефективне використання наявних ресурсів за рахунок грамотної оркестрації обробки вимог. Головним недоліком обраного шляху є суттєве ускладнення самої системи та тонкощі при налаштуванні балансу між класами вимог. В рамках даного дослідження не розглядається моральна сторона питання та не враховується «справедливість» надання послуг, а задача полягає у збільшенні загальної пропускної здатності.

В даному розділі розглядатимемо розв’язання поставленої задачі за рахунок розбиття загального потоку заявок на сім’ю потоків за розміром з подальшим наданням меншим задачам більшого пріоритету. Суть підходу полягає в тому, що ми від простої СМО переходимо до системи з комбінованими пріоритетами [45]. Це означає, що ми визначимо фіксовану кількість категорій заявок і сортуватимемо вимоги на вході в систему. При цьому пріоритет буде гнучко підлаштовуватись під стан системи [46]. Надання пріоритету меншим задачам дозволить зменшити загальну ймовірність відмови через переповнення черги в системі. Такий вибір ґрунтується на тому, що причина відмови залежить від кількості задач, а не від їх складності, на відміну від продуктивності системи, яка у свою чергу залежить від часу, необхідного на обробку заявки, і не залежить від кількості заявок в черзі. Відповідно, надаючи перевагу меншим задачам, ми можемо збільшити кількість опрацьованих задач

за відрізок часу і, в такий спосіб, мінімізувати кількість відмов [45, 46].

2.4.1 Модель розбиття вхідного потоку

Розвиток статистичних методів навчання та збільшення доступності даних дозволяють отримувати більше інформації з боку клієнтів. З точки зору операційного менеджменту важливо визначити, як використовувати цю інформацію для підвищення продуктивності системи. Ми розглядатимемо інформація про час обслуговування клієнтів та те, як її можна використовувати для вдосконаленого планування у сервісних системах.

Оскільки початковий потік вимог не є ранжованим, постає потреба у визначенні способу його розбиття на окремі категорії заявок. Цю задачу буде вирішувати модуль сортування заявок. Розбиття загального вхідного потоку заявок на класи за їх розміром є необхідною передумовою для подальших удосконалень моделі системи. Запропоновано декілька підходів до визначення складності вимоги, серед яких:

– самоідентифікація заявки;

– використання особливостей функції ймовірнісного розподілу часу обслуговування загального потоку заявок;

– застосування навченої моделі штучного інтелекту.

Розглянемо переваги та недоліки для кожного з вище наведених рішень. Задля підвищення якості категоризації допускаємо застосування довільних комбінацій даних підходів.

Спочатку розглянемо модель самоідентифікації розміру заявки. Найпростішим рішенням може бути делегація зобов’язання визначення розміру на саму заявку. Тобто в момент класифікації вимога має надати чисельне значення еквівалентне рівню складності її обробки. Таким чином, алгоритм категоризації може бути суттєво спрощений. Однак, цей підхід має кілька недоліків, а саме:

– заявка може надати недостовірну інформацію;

– у клієнта може бути відсутнє розуміння відносної складності задачі;

– вимоги різного походження можуть використовувати відмінні шкали оцінки відносної складності.

Для усунення першої проблеми ми можемо скомбінувати даний підхід з більш достовірними опціями. Наприклад, перевіряти співвідношення наданої складності до інших параметрів обробки заявки, які вже опрацьовані раніше. Так ми застосуємо надбаний раніше досвід обслуговування для покращення апроксимації часу виконання нових вимог. Або можна обрати інший шлях та знехтувати ймовірністю помилкової оцінки клієнтом у випадку, якщо система використовується у середовищі, де користувачі мають високий рівень кваліфікації. В такому випадку ми повністю покладаємося на їхню експертизу.

У другому випадку ми маємо замістити відсутню оцінку власною експертизою. Наприклад, якщо ми попередньо збирали статистику про залежність часу виконання вимог від вхідних параметрів, то цей досвід можна екстраполювати на чергову вимогу. Однак збір та обробка даних значно ускладнюють реалізацію, а також отримана в такий спосіб оцінка може бути нерелевантна. Якщо відомо про проблему подібного типу, бажано відмовитись від самоідентифікації складності на користь інших альтернатив.

Для третього випадку нам знадобиться напрацювати алгоритм узгодження оцінок. Його може бути реалізовано за допомогою побудови додаткового модуля приведення. Задля реалізації такої компоненти необхідно додати таблицю співвідношення оцінок для кожного клієнта до загальної системи оцінювання. Підтримка модуля приведень може бути проблематичною, але вона локалізує складність. Внаслідок чого, інші частини системи працюють виключно з загальними оцінками та не залежать від підходу клієнтів до оцінювання.

Розглянемо далі підхід, що базується на аналізі функції розподілу. Ми підкреслюємо роль розподілу часу обслуговування у визначенні вигоди, яку можна отримати від інтелектуального планування. При застосуванні функції розподілу початкового вхідного потоку вимог нам буде необхідно зібрати

певну статистику залежності часу виконання від параметрів. Наступним етапом буде визначення того, за яких вхідних даних час обслуговування потрапляє в зону математичного сподівання, а за яких – відхиляється в менший або більший бік. Завдяки цьому для подальших надходжень матимемо алгоритм категоризації за розміром та призначення пріоритету. У нашому правилі кількох класів пріоритетів саме розподіл часу обслуговування визначає відповідні порогові значення для поділу на класи. Цей поріг у свою чергу визначає середній час обслуговування для класу з певним рівнем пріоритету. Зокрема, можна показати, що значення цих порогів залежать від швидкості спадання функції кумулятивного розподілу хвоста розподілу часу обслуговування [11]. Чим повільніше спадає хвіст розподілу (тобто чим «важчий» хвіст розподілу часу обслуговування), тим більше вимог потрапляють в крайові категорії. Це свідчить про те, що системи з «важкими» хвостами розподілу часу обслуговування мають тенденцію отримувати більшу вигоду від застосування запропонованої моделі.

Важливим є питання стійкості до неточної інформації про час обслуговування. У реальних застосуваннях вимоги до часу обслуговування можуть бути невідомими або можуть містити помилки в оцінках. Існує дуже мало досліджень, які вивчають планування на основі розміру завдання з урахуванням помилок в оцінках, що було зазначено як відкриту проблему у [19]. Дослідження таких проблем стикається з кількома ключовими викликами.

По-перше, помилки оцінок можуть виникати в різних формах залежно від вибору моделі, і сама помилка оцінки може впливати на інформацію про час обслуговування. Наприклад, якщо час обслуговування оцінюється за допомогою лінійної регресійної моделі на основі спостережуваних характеристик клієнта, то розподіл часу обслуговування є згорткою варіативності спостережуваних даних та помилки оцінки. Таким чином, різні види помилок оцінки можуть вимагати різного підходу до аналізу. Це питання підіймалось також для самоідентифікації задач. Для подальшого дослідження ми знехтуємо даною похибкою, та будемо вважати клієнта експертом з оцінки

власних задач.

По-друге, аналіз динаміки системи на рівні процесу є складним, оскільки часто потрібно відстежувати процес «віку» завдань або залишковий час обробки всіх завдань у системі. Дану проблеми ми можемо оминути, оскільки ми не надаємо можливості переривання виконання, а кожну вимогу вважаємо атомарною.

По-третє, оптимізація політики планування потребуватиме оновлення оцінки часу обслуговування в процесі обробки завдання. Наша політика планування та аналітична структура надають зручний спосіб дослідити, як неточна інформація про час обслуговування впливає на продуктивність системи. Власне, помилка оцінки впливає на нашу здатність класифікувати завдання за пріоритетними класами, внаслідок чого система починає працювати як така що має базовий вхідний потік вимог.

І нарешті, розглянемо застосування моделей штучного інтелекту. Для апроксимації коефіцієнта пріоритезації можна також застосовувати штучний інтелект, модель якого попередньо має бути натренована на широкій вибірці даних. Важливо, щоб функція ймовірнісного розподілу вхідного потоку співпадала з функцією, що буде використовуватись при навчанні моделі штучного інтелекту [49]. Із поглибленням інтеграції та застосуванням глибокого навчання на великих об’ємах даних нейронної мережі, під час її застосування будуть розширюватись можливості щодо виявлення певних патернів (послідовностей) поведінки в балансуванні навантаження [48]. Це дозволить передбачати наперед надмірне завантаження системи та завчасно адаптуватись за рахунок зміни пріоритетів. Дані переваги у свою чергу в повній мірі виправдовують додаткову складність, пов’язану з використанням штучного інтелекту для вирішення проблеми збільшення пропускної здатності СМО.

2.4.2 Пріоритезація за двома класами

Опишемо пріоритезацію за двома класами. Розглядатимемо головним

чином перехід до системи з двома класами пріоритетів за часом обслуговування. Таке рішення прийнято з міркувань спрощення моделі на початковому етапі. Очікується, що навіть таке розбиття на більші та менші задачі вже надасть вагому перевагу у порівнянні з базовою системою масового обслуговування (без пріоритетів). У подальшому зберігаємо плани щодо масштабування до більшої кількості класів.

Також, з міркувань спрощення розрахунків, розглядатимемо одноканальну систему. Така зміна не завадить застосуванню здобутих результатів до загального випадку, оскільки кількість каналів обслуговування не впливає на розмір черги за умови пропорційного зменшення навантаження вхідного потоку заявок. Задля спрощення спостереження, знімемо також і обмеження на чергу. В цьому розділі ми будемо спостерігати за розміром черги для різних режимів роботи модуля пріоритезації.

Процес прибуття заявок є загальним за схемою оновлень та незалежними й однаково розподіленими часами обслуговування, тобто чергу типу $M/M/1$. Позначимо λ як швидкість прибуття заявок, а μ як швидкість обслуговування. Тоді інтенсивність трафіку ρ визначається як:

$$\rho = \frac{\lambda}{\mu},$$

де ρ – це відношення швидкості прибуття заявок до швидкості обслуговування, що показує завантаження системи. Якщо $\rho < 1$, черга має тенденцію стабілізуватися, тоді як при $\rho \geq 1$ черга може рости без обмежень, що призводить до перевантаження системи та відмов. Позначимо F як функцію кумулятивного розподілу (cdf) часу обслуговування, а $\overline{F}$ є її хвостом та дорівнює $\overline{F} = 1 - F$. Два класи пріоритетів визначаються пороговим значенням.

$$K(\rho, F) = \overline{F}^{-1}((1-\rho)^{1-\delta}), \tag{2.1}$$

де δ таке, що $0 < \delta < 1$.

Клієнти, чий час обслуговування менший або дорівнює $K(\rho,F)$, потрапляють у клас з високим пріоритетом (Клас 1), тоді як клієнти, чий час обслуговування більший за $K(\rho,F)$, належать до класу з низьким пріоритетом (Клас 2). Параметр δ регулює баланс навантаження між класами з високим та низьким пріоритетом. Рекомендується встановлювати δ малим, наприклад, у межах від 0,01 до 0,1 [11].

Поріг $K(\rho,F)$ залежить як від розподілу часу обслуговування, так і від інтенсивності трафіку системи. Він розроблений так, що $K(\rho,F)$ збільшується зі збільшенням ρ. Важливо зазначити, що якщо розподіл часу обслуговування має нескінченну підтримку, то $K(\rho,F)$ прямує до нескінченності, коли $\rho \to 1$. Однак $K(\rho,F)$ не повинен зростати надто швидко, коли ρ наближається до 1. Зокрема, його слід вибрати правильно, щоб контролювати навантаження клієнтів Класу 1.

У табл. 2.1 порівнюється продуктивність правила двох класів пріоритетів з іншими еталонними політиками для черг $M/M/1$ з різними інтенсивностями трафіку. Еталонні політики включають FCFS, shortest-job-first (SJF), де пріоритет надається задачі меншого розміру без дозволу на переривання, і SRPT, де пріоритет надається задачі з найменшим залишковим часом обробки з дозволом на переривання. Ми розглядаємо дві реалізації правила двох класів пріоритетів: одна дозволяє переривання (two-class P), а інша – ні (two-class NP).

Ми спостерігаємо, що в порівнянні з FCFS правило двох класів пріоритетів здатне скоротити середню довжину черги приблизно вдвічі або більше при різних інтенсивностях трафіку. Також зазначимо, що SRPT показує найкращі результати, як і встановлено в літературі [11]. Однак різниця між правилами двох класів пріоритетів і SRPT є незначною. Це свідчить про те, що, хоча додавання більшої кількості класів пріоритетів може покращити продуктивність системи, гранична вигода від цього буде відносно незначною.

Таблиця 2.1 – Середня довжина черги в сталому стані для черг $M/M/1$ при різних політиках планування ($\mu = 1,\ \lambda = \rho$, $K = \overline{F}((1-\rho)^{1-0,05}) = -(1-0,05)\log(1-\rho)$)

ρ	0.75	0,8	0,85	0,9	0,95
FCFS	3.43	3.92	5.55	8.80	18.55
Two-Class NP	2.31	2.63	3.37	4.68	8.11
Two-Class P	1.99	2.26	2.90	4.07	7.21
SJF	2.00	2.29	2.83	3.75	5.91
SRPT	1.62	1.86	2.33	3.16	5.18

У таблиці 2.1:

– FCFS: перший прийшов – перший обслужений;

– Two-Class NP: двокласова без пріоритету;

– Two-Class P: двокласова з пріоритетом;

– SJF: коротша робота першою;

– SRPT: залишковий час обробки найкоротший.

Для n-ї системи ми визначаємо K_n як порогове значення для двох класів, тобто клієнти з часом обслуговування меншим або рівним K_n класифікуються до Класу 1, а інші – до Класу 2. Для спрощення пояснення припустимо, що клієнти Класу 1 мають переважний пріоритет над клієнтами Класу 2. Почнемо з введення декількох додаткових позначень.

Позначимо $\lambda_1^n = \lambda^n F(K_n)$ та $\lambda_n^2 = \lambda_n \overline{F}(K_n)$ як відповідні швидкості прибуття клієнтів Класу 1 та Класу 2 у n-й системі. Також позначимо $\nu_i^n(k)$ як час обслуговування k-го клієнта Класу i у системі n. Тоді [11]

$$\mu_1^n = (\mathbf{M}[\nu_1^n(k)])^{-1} = \left(\frac{1}{F(K^n)} \int_0^{K^n} xf(x)dx \right)^{-1},$$

$$\mu_2^n = (\mathbf{M}[\nu_2^n(k)])^{-1} = \left(\frac{1}{\overline{F}(K^n)} \int_{K^n}^{+\infty} xf(x)dx \right)^{-1}.$$

Позначимо швидкості обслуговування двох класів як відповідні значення. Нехай $\rho_i^n = \frac{\lambda_i^n}{\mu_i^n}$ позначає інтенсивність трафіку для Класу i, де $i = 1, 2$. Далі ми вводимо загальну ідею, що лежить в основі нашої розробки. Виходячи з існуючих результатів для черг з пріоритетами [51], ми робимо два важливих спостереження:

1) черга Класу 1 масштабується як $(1-\rho_1^n)^{-1}$ (зазначимо, що якщо клієнти Класу 1 мають переважний пріоритет над клієнтами Класу 2, черга Класу 1 розвивається так, ніби в системі є лише клієнти Класу 1);

2) процес загального робочого навантаження масштабується як $(1-\rho_1^n)^{-1} = O(\sqrt{n})$ (дійсно, таке масштабування для процесу навантаження виконується для будь-якої політики планування, яка зберігає навантаження, тобто сервер не простоює, коли є клієнти у черзі).

Зазначимо, що якщо в «граничній» черзі залишаються лише клієнти Класу 2, то процес довжини черги масштабується як $\mu_2^n\sqrt{n}$. Щоб досягти меншого масштабування, ніж $\sqrt{n}$ для процесу довжини черги, потрібно, щоб $\mu_2^n \to 0$, коли $n \to \infty$. Однак це потрібно робити обережно. Зокрема, потрібно забезпечити, щоб за масштабування $\mu_2^n\sqrt{n}$ у черзі асимптотично не було клієнтів Класу 1, тобто $\mu_2^n\sqrt{n}(1-\rho_1^n) \to \infty$, коли $n \to \infty$.

Припущення 1. Час обслуговування має неперервний розподіл із функцією щільності ймовірності f. Існує $C > 0$, таке, що $f(x) > 0$ для будь-якого $x > C$. Існує $\delta > 0$, таке, що $\mathbf{M}[\nu^{2+\delta}] < +\infty$.

Перша частина припущення 1, по суті, вимагає, щоб розподіл часу обслуговування мав нескінченну підтримку (був необмеженим) [11]. Навіть за політики SRPT, коли розподіл часу обслуговування має обмежену підтримку, черга масштабується як $\frac{1}{1-\rho}$ [28]. У цьому випадку жодного значного покращення від «розумного» планування не відбувається. Умова щодо

моментів у припущенні 1 є стандартною для встановлення дифузійної межі для процесів довжини черги (див., наприклад, [52]).

Згідно з припущенням 1, ми позначаємо $\sigma_s^2 = \mathbf{D}[\nu]$. Також ми накладаємо наступне припущення щодо «легкого хвоста» для τ_∞. Ключовим елементом оптимізації є вибір порогового значення K_n відповідним чином. Наш вибір K_n задовольняє наступну умову.

Припущення 2. $K_n \to +\infty$, і існує $\delta \in \left(0, \frac{1}{2}\right)$, таке, що $n^{\frac{1}{2}-\delta}\overline{F}(K_n) \to +\infty$, коли $n \to \infty$.

Прокоментуємо припущення 2. Нехай $\gamma_n := K_n\mu_n^2$. Згідно з припущенням 1, $\frac{2}{3} \le \gamma_n \le 1$. Тоді припущення 2 означає, що $\mu_n^2 \to 0$, коли $n \to \infty$ [11]. Це допомогає забезпечити, що черга Класу 2 масштабується повільніше, ніж $\sqrt{n}$. Також зазначимо, що

$$\rho_n^1 = \frac{\lambda_n^1}{\mu_n^1} = 1 - \frac{\beta}{\sqrt{n}}\int_0^{K_n} \overline{F}(K_n)xf(x)dx = 1 - \mu_n^2 + O\left(\frac{1}{\sqrt{n}}\right).$$

Таким чином, згідно з припущенням 2, існує $\delta \in \left(0, \frac{1}{2}\right)$ таке, що

$$\frac{n^{1/2-\delta}}{K^n}(1-\rho_1^n) \to +\infty, \text{ коли } n \to \infty,$$

або

$$n^{1/2-\delta}\mu_2^n(1-\rho_1^n) \to +\infty, \text{ коли } n \to \infty.$$

Це допомагає забезпечити, що черга Класу 1 масштабується ще повільніше, ніж черга Класу 2.

Нарешті, нагадаємо, що в (2.1) ми пропонуємо встановити

$K_n = F^{-1}((1-\rho_n)^{1-\delta})$ для деякого $0<\delta<1$. Такий вибір K_n задовольняє припущення 2, коли розподіл часу обслуговування задовольняє припущення 1.

Наведемо числові експерименти для правила двох класів пріоритетів, щоб проілюструвати продуктивність цього правила до досягнення асимптотичного стану. Зокрема, ми досліджуємо вплив розподілу часу обслуговування на продуктивність системи. Ми також розглядаємо залежність продуктивності від класів у рамках правила двох класів пріоритетів.

Спершу зазначимо, що широкий діапазон значень $K(\rho,F)$ відповідає припущенню 2. Тому ми починаємо з аналізу чутливості, щоб дослідити, як різні значення $K(\rho,F)$ впливають на продуктивність правила двох класів пріоритетів. На рис. 2.9 показано середню довжину черги в сталому стані для різних значень порогу для правила двох класів пріоритетів з і без переривання.

Слід зазначити, що очікувана довжина черги дійсно змінюється з різними значеннями $K(\rho,F)$. Однак, масштаб варіації дуже малий для діапазону значень $K(\rho,F)$ поданих на графіку. Це свідчить про те, що правило двох класів пріоритетів є відносно нечутливим до вибору порогу в межах розумного діапазону.

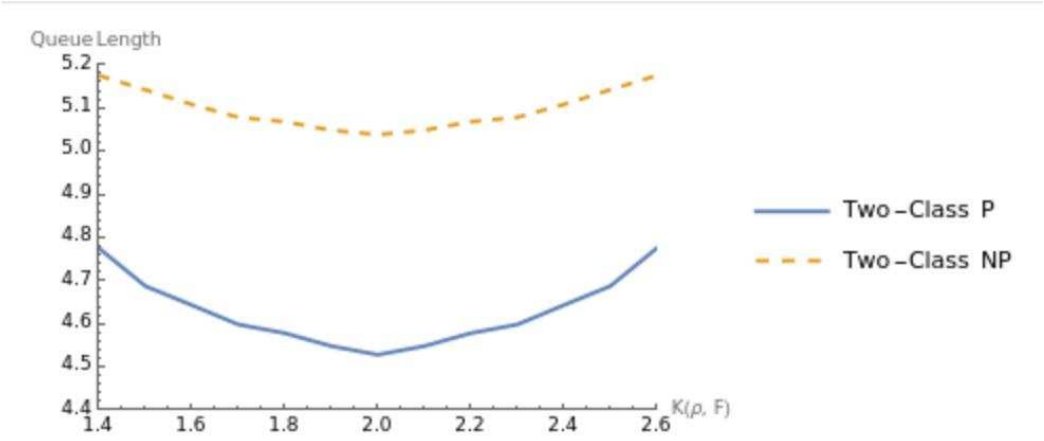

Рисунок 2.9 – Середня довжина черги в сталому стані
для черги типу $M/M/1$ за правила двох класів пріоритетів (P)
з різними значеннями порогу

У табл. 2.2 ми порівнюємо середню довжину черги в сталому стані для черг $M/GI/1$ з однаковою швидкістю обслуговування одиниці, але з різними розподілами часу обслуговування. При цьому розподіл часу обслуговування визначається законом Парето (m_α, α) для різних значень α ($\mu = 1$, $\lambda = \rho$, $K(\rho, F) = F((1-\rho)^{1-0.05})$).

Таблиця 2.2 – Середня довжина черги в сталому стані для черг типу $M/GI/1$

	FCFS			Two-class (P)			SRPT		
ρ	0,85	0,9	0,95	0,85	0,9	0,95	0,85	0,9	0,95
$\alpha = 2.5$	5,12	8,19	23,20	3,32	4,43	9,56	2,20	3,05	7,34
$\alpha = 5$	4,22	5,22	16,00	3,01	4,34	7,65	2,55	3,83	6,80
$\alpha = 7.5$	4,05	5,05	15,30	3,13	4,57	7,80	2,64	4,16	6,94
$\alpha = 10$	4,00	5,00	15,20	3,16	4,70	8,00	2,70	4,35	7,00

Ми розглядаємо клас розподілу Парето з параметром α, де ми встановлюємо $m_\alpha = \frac{\alpha - 1}{\alpha}$, а хвостова функція розподілу визначається як $\overline{F}(x) = \left(\frac{m_\alpha}{x}\right)^\alpha$. Зазначимо, що чим більше значення α, тим швидше спадає $\overline{F}$, і, відповідно, тим легшим є хвіст розподілу часу обслуговування.

Ми спостерігаємо, що при порівнянні правила двох класів пріоритетів із FCFS, правило двох класів завжди забезпечує меншу довжину черги. Зменшення довжини черги при застосуванні правила двох класів пріоритетів стає більшим, коли інтенсивність трафіку зростає. Наприклад, для розподілу часу обслуговування Парето з $\alpha = 2{,}5$, коли $\rho = 0{,}8$, правило двох класів зменшує середню довжину черги на 32%; коли $\rho = 0{,}9$, зменшення становить 46%; а в крайньому випадку, коли $\rho = 0{,}99$, зменшення досягає 76%. Ці спостереження узгоджуються з асимптотичним аналізом, наведеним у роботі [11]. Зокрема, процес довжини черги масштабується повільніше при збільшенні

ρ за правилом двох класів пріоритетів порівняно з FCFS, і ця різниця у масштабуванні стає більш помітною, коли ρ наближається до 1.

Далі ми детальніше розглядаємо вплив розподілів часу обслуговування. Порівнюючи правило двох класів пріоритетів із FCFS, ми зазначаємо, що більший ефект зменшення довжини черги досягається для розподілів часу обслуговування з важчим хвостом, тобто коли α менше. Наприклад, для $\rho = 0,9$, коли $\alpha = 2,5$, ми досягаємо зменшення довжини черги на 46% за допомогою правила двох класів пріоритетів; коли $\alpha = 5$, зменшення становить 17%; а коли $\alpha = 10$, зменшення складає лише 6%.

Ми також спостерігаємо, що при використанні FCFS, коли хвіст розподілу часу обслуговування стає легшим (тобто α збільшується), середня довжина черги зменшується. Це очікувано, оскільки для черги типу $M/GI/1$ з одиничною швидкістю обслуговування середня очікувана довжина черги в сталому стані за правилом FCFS має вигляд:

$$\rho + \frac{(1+\sigma_s^2)\rho^2}{2(1-\rho)},$$

де σ_s^2 зменшується при збільшенні α. Зазначимо, що для правила двох класів пріоритетів

$$\mathbf{M}[Q(\infty)] \approx \frac{\gamma(1+\sigma_s^2)}{2(1-\rho)K(\rho, F_\alpha)},$$

де $K(\rho, F_\alpha) = m_\alpha (1-\rho)^{-\frac{1-\delta}{\alpha}}$.

Зазначимо, що коли α збільшується, і σ_s^2, і $K(\rho, F_\alpha)$ зменшуються. Оскільки σ_s^2 змінюється разом із ρ, ефект від $K(\rho, F_\alpha)$ буде домінувати, коли ρ достатньо близьке до 1. Однак, коли ρ мале, незрозуміло, чи домінуватиме менше значення σ_s^2 або менше значення K_n. З табл. 2.2 видно, що при $\rho = 0,8$ середня довжина черги спочатку зменшується, а потім зростає зі збільшенням

α. У цьому режимі ефект σ_s^2 відіграє роль. Однак, коли $\rho = 0{,}99$, середня довжина черги збільшується зі збільшенням α. У цьому режимі домінує ефект від $K(\rho, F_\alpha)$.

Нарешті, порівнюючи SRPT із правилом двох класів пріоритетів, ми зазначаємо, що коли α мале, SRPT може досягти значно коротшої середньої довжини черги, ніж правило двох класів пріоритетів. Однак, очікувані довжини черг за цими двома політиками планування стають ближчими зі збільшенням α. Наприклад, для $\rho = 0{,}9$, коли $\alpha = 5$, оптимальний розрив для правила двох класів пріоритетів становить 13%; коли $\alpha = 10$, розрив зменшується до 8%. Це спостереження узгоджується з асимптотичним аналізом [11].

Продемонструємо залежну від класу продуктивність за правилом двох класів пріоритетів. На рис. 2.10 порівнюється середня довжина черги в сталому стані для Класу 1 та Класу 2 у черзі типу $M/M/1$ за правилом двох класів пріоритетів.

З рис. 2.10 бачимо, що оскільки Клас 1 має пріоритет, його черга значно коротша, ніж у Класу 2.

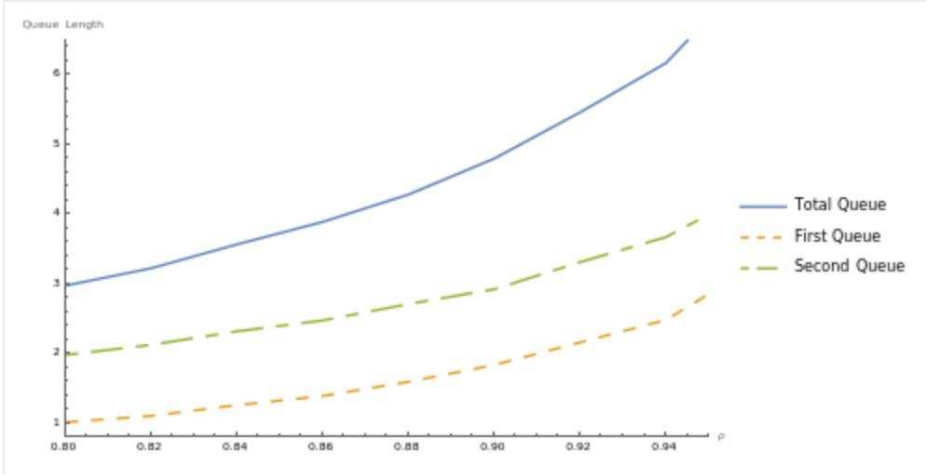

Рисунок 2.10 – Середня довжина черги в сталому стані для черг типу $M/M/1$ за різних інтенсивностей трафіку за правилом двох класів пріоритетів (P)

У табл. 2.3 порівнюються середні часи очікування в сталому стані для

Класу 1, Класу 2 та всіх клієнтів разом (All) за правилом двох класів пріоритетів і FCFS.

Таблиця 2.3 – Середній час очікування в сталому стані для черг типу $M/M/1$ за правилом двох класів пріоритетів (P) та FCFS ($\mu = 1,\ \lambda = \rho,\ K(\rho, F) = F((1-\rho)^{1-0,05})$)

	Two-class (P)			FCFS		
ρ	Клас 1	Клас 2	Разом	Клас 1	Клас 2	Разом
$\alpha = 0,8$	0,84	10,28	2,89	4,63	6,60	5,05
$\alpha = 0,85$	1,08	15,76	3,51	6,38	8,55	6,74
$\alpha = 0,9$	1,54	29,27	4,64	9,82	12,31	10,10
$\alpha = 0,95$	2,82	90,23	7,76	20,02	23,08	20,22

Ми спостерігаємо, що в порівнянні з FCFS, незважаючи на значне покращення середнього часу очікування для клієнтів Класу 1, клієнти Класу 2 стикаються з суттєвим збільшенням часу очікування за правилом двох класів пріоритетів. Ці спостереження узгоджуються з асимптотичним аналізом, проведеним у роботі [11].

2.4.3 Модель балансування пріоритетів

Розглянемо оцінку поточного завантаження. Важливим етапом оптимізації є контроль поточного завантаження системи. При цьому нас не цікавить кількість одночасно працюючих каналів обслуговування, оскільки з точки зору модуля пріоритезації всі сценарії, за яких система працює з недобором роботи, означають відсутність необхідності розділяти вхідний потік [44]. Для правильного функціонування необхідно збирати інформацію щодо поточного складу черги. Таким чином, ми будемо розуміти відносно чого ми пріоритезуємо задачі. Відповідно, ширина часових діапазонів (складності вимог) для категорій задач та загальна кількість класів заявок залежить від

розміру черги та характеристик вхідного потоку.

Проте, при впровадженні цієї політики в сервісні системи виникає проблема: важко управляти чергою, де порядок клієнтів може змінюватися після кожного прибуття. Унаслідок цього для використання пріоритету за класами обслуговування під кожен клас виділяється окрема черга [47, 48]. Це забезпечить збереження послідовності обслуговування заявок, що були розподілені до однієї категорії. Водночас, в будь-який момент часу можна отримати інформацію щодо кількості заявок надісланих до певної групи за розміром. Розміри цих черг залежать від функції ймовірнісного розподілу, що регулює час обробки заявок загального потоку вимог та визначаються перед початком роботи системи. Разом з тим, наповненість кожної з черг є важливою ознакою. За такої будови, наша система буде давати відмову тільки тим заявкам, які неможливо додати в чергу згідно з визначеною категорією [49]. Відповідно, ми маємо неперервно слідкувати за наповненістю кожної з черг категорій задля розуміння завантаженості СМО. Для цього використовуватимемо метрику – кількість вільних місць на певний клас заявок. Таким чином, ми отримуємо метод стійкий до змін балансу між складними та легкими задачами. Він є інформативним та дозволяє збирати детальну інформацію про навантаження системи, для подальшого покращення алгоритму.

Опишемо тепер балансування пріоритетів. За наявності точної інформації про час обслуговування відомо, що політика Shortest Remaining Processing Time First (SRPT) забезпечує високу продуктивність. Однак вона не забезпечує всі вимоги поставленої задачі. Відповідно, ми вдосконалили модель пріоритезації під наші потреби. Головним завданням, яке вирішує модуль пріоритезації, є прийняття рішення про обрання класу заявок, що використовуватиметься як джерело для постанови в чергу, з якої беруть собі роботу канали обслуговування. З технічної точки зору це означає, що даний елемент системи окрім черги, яка використовується для подачі вимог на виконання, взаємодіє з окремими чергами для задач кожного з класів.

Опишемо запропоноване вдосконалення у кортежному записі:

$$CMO = (I,\ P,\ GQ,\ SC), \tag{2.2}$$

де I – потік вимог, $I = (\lambda, C)$;

P – модуль пріоритезації, $P = (PS, CQ, PP)$;

GQ – загальна черга очікування, $Q = (m, QR)$;

SC – канали обслуговування, $SC = (n, \mu, O)$.

Тоді, розгортаючи перший (2.2) рівень абстракції, отримаємо вираз (2.3):

$$CMO = ((\lambda,\ C),\ (PS,\ CQ,\ PP),\ (m,\ QR),\ (n,\ \mu,\ O)), \tag{2.3}$$

де λ – інтенсивність потоку заявок;

C – множина категорій заявок, $C = \{S, M, L\}$;

PS – сортувальник заявок, $f_{copm}(I) \to \{PQS, PQM, PQL\}$;

CQ – множина черг під кожну категорію заявок, $CQ = \{PQS, PQM, PQL\}$;

PP – подавач заявок, $PP = (QS)$;

m – максимальна довжина загальної черги;

QR – правило роботи черги (наприклад, FIFO);

n – кількість каналів обслуговування;

μ – інтенсивність обслуговування каналу;

O – вихідний потік результатів, $O = (OS, OF)$.

Продовжуємо розписувати (2.3) рівень абстракції, отримаємо рівняння:

$$CMO = ((\lambda,\ \{S,\ M,\ L\}),\ (PS,\ \{PQS,\ PQM,\ PQL\},\ PP),$$
$$(m,\ QR),\ (n,\ \mu,\ \{OS,\ OF\})), \tag{2.4}$$

де S – категорія маленьких задач;

M – категорія середніх задач;

L – категорія великих задач;

PQS – черга для менших задач, $PQS = (m_s, QR_s)$;

PQM – черга для середніх задач, $PQM = (m_m, QR_m)$;

PQL – черга для великих задач, $PQL = (m_L, QR_L)$;

OS – заявки, що було успішно оброблено;

OF – заявки, яким було відмовлено або які отримали помилку під час обробки.

Врешті, розгортаючи (2.4) в останній раз виводимо фінальний вираз:

$$CMO = ((\lambda, \{S, M, L\}), (PS, \{(m_s, QR_s), (m_m, QR_m), (m_L, QR_L)\}, PP),$$
$$(m, QR), (n, \mu, \{OS, OF\})),$$

де m_s – довжина черги менших задач;

QR_s – правило роботи черги менших задач (FIFO);

m_m – довжина черги середніх задач;

QR_m – правило роботи черги середніх задач (FIFO);

m_L – довжина черги більших задач;

QR_L – правило роботи черги більших задач (FIFO);

QS – правило вибору черги категорії для передачі заявки в загальну чергу (наприклад, стратегія пріоритетів).

Таким чином, з початкової послідовності взаємодій:

$$I \to PS \to SC \to O.$$

Отримуємо удосконалену послідовність:

$$I \to PS \to CQ \to PP \to GQ \to SC \to O.$$

Таким чином, на рис. 2.11 можемо побачити оновлений алгоритм для заданої СМО. Зміни торкнулися передусім механізму прийому заявок в систему, оскільки черга виконання залишилась тою самою. Але тепер їй

передує аналіз та сортування заявок по чергах відповідних розмірів. Після чого, враховуючи поточний склад черги виконання, а також наповненість черг по категорії заявок, відбувається прийняття рішення про постанову вимоги певного розміру до черги виконання. Черга виконання може бути зменшена до розміру, що дорівнює кількості каналів обслуговування. Це пов'язано з тим, що її роль в оновленій моделі зводиться до буферу, ціль якого полягає у безперервній подачі задач на канали, а сама черга очікування розподіляється між чергами відповідних класів.

Застосування такого рішення дозволяє уникнути додаткової категорії відмов та збільшити пропускну здатність системи.

Задля чисельного вимірювання успішності механізму пріоритезації, введемо метрику, яка буде відображати пропорційність добору задач. Вона представлятиме собою співвідношення відмов великим заявкам до загальної кількості відмов за проміжок часу t

$$V(t) = \frac{V_L}{V_{\text{заг.}}},$$

де V_L – це відсоток відмов для великих задач;

$V_{\text{заг.}}$ – загальна кількість відмов у системі за проміжок часу t.

Визначивши розмір проміжку часу, за який вестимуться спостереження, ми отримаємо періоди коригування балансу [48]. Після завершення збору аналітики по відмовах у системі, настає корекція коефіцієнту пріоритезації, який змінить певним чином пропорцію великих, середніх та маленьких заявок, що перебувають у черзі.

Сортувальник заявок має повідомляти відправнику заявок про переповнення черги певного розміру шляхом обміну сигналами. Це необхідно робити з метою зміни пріоритетів та уникання подальших відмов через переповнення черги категорії. Окрім того, ми маємо пропускати великі задачі до системи, щоб не припинити їх обслуговування [50]. Про це має дбати модуль

пріоритезації.

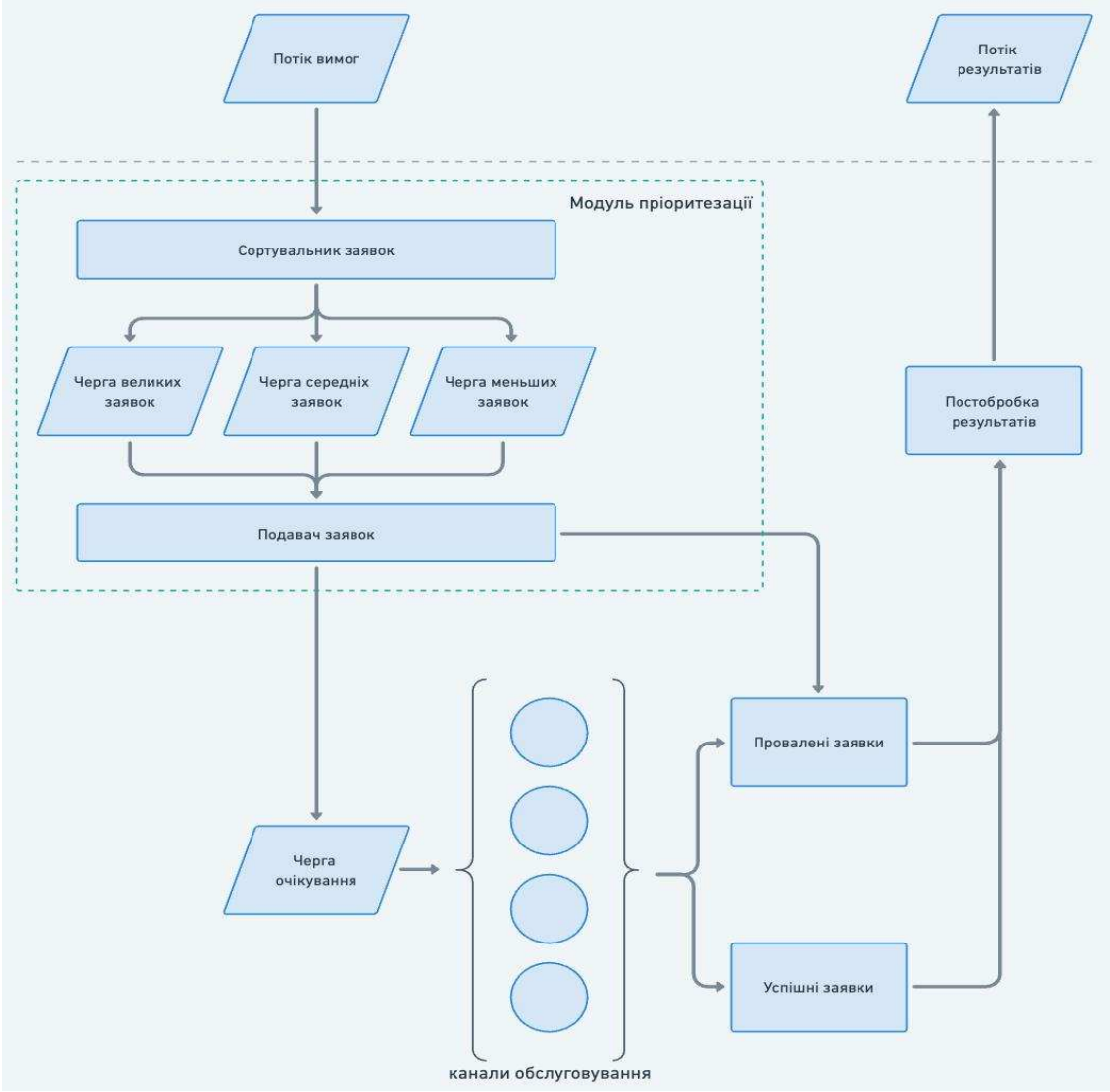

Рисунок 2.11 – Алгоритм обробки заявок з розбиттям на класи

За таких обставин наш алгоритм не гарантує співпадіння послідовності надходження заявок до системи з послідовністю надходження на виконання. Але ми зберігаємо цю послідовність в рамках кожної окремої категорії задач. Послідовність надходження результатів обслуговування при кількості каналів більше за один не має обмежень щодо відповідності послідовності через асинхронну та випадкову природу процесу обслуговування.

Наступним кроком буде розробка алгоритму, який даватиме стабільне потрапляння в зону балансу.

Для кількісної оцінки ефективності системи масового обслуговування з

пріоритетами використовують ті самі характеристики, як і для простої системи, але, на додачу до загальних показників, це робиться для кожної категорії окремо.

Оскільки черга змінила свою структуру, маємо зміни у формулі, яка описує загальну ймовірність відмови в системі. Кількість місць очікування в системі залишилась незмінною, однак тепер частина черги розподілилась між класами. Для найгіршого сценарію, коли всі заявки, що надходять до системи мають один розмір отримаємо наступний вираз:

$$P_{\text{відм.}} = p_{n+m+m_j} = \frac{n^n R_{\text{заг.}}^{n+m+m_j}}{n!} p_0, \tag{2.5}$$

де m – кількість місць очікування у загальній черзі, з якої отримують заявки канали обслуговування;

m_j – розмір черги для j-ї категорії, $j = 1, 2, ..., r$.

При цьому

$$p_0 = \left[\sum_{i=0}^{n-1} \frac{n^i R_{\text{заг.}}^i}{i!} + \frac{n^n}{n!} \cdot \frac{R_{\text{заг.}}^n (1 - R_{\text{заг.}}^{m+m_j+1})}{1 - R_{\text{заг.}}} \right]^{-1}. \tag{2.6}$$

Враховуючи (1.17) та підставивши (2.6) у (2.5) матимемо наступний вираз:

$$P_{\text{відм.}} = \frac{n^n R_{\text{заг.}}^{n+m+m_j}}{n!} \left[\sum_{i=0}^{n-1} \frac{n^i R_{\text{заг.}}^i}{i!} + \frac{n^n}{n!} \cdot \frac{R_{\text{заг.}}^n (1 - R_{\text{заг.}}^{m+m_j+1})}{1 - R_{\text{заг.}}} \right]^{-1}.$$

Для оцінки результатів дослідження пропонується використовувати декілька чисельних метрик, які мають якнайкраще характеризувати наслідки застосування оптимізації. Головною метрикою буде загальна ймовірність відмови новоприбулій довільній заявці через переповнення черги очікування. Для глибшого розуміння природи оптимізації розраховуватимемо показники ймовірності відмови за умови повної пріоритезації по кожному з розмірів

вимог. На рис. 2.12 – рис. 2.14 зображено графіки залежності ймовірності відмови для певної категорії від загального навантаження, що відповідають різним стратегіям:

– без застосування пріоритезації (рис. 2.12);

– пріоритет надається меншим задачам (рис. 2.13)

– пріоритет надається великим задачам (рис. 2.14).

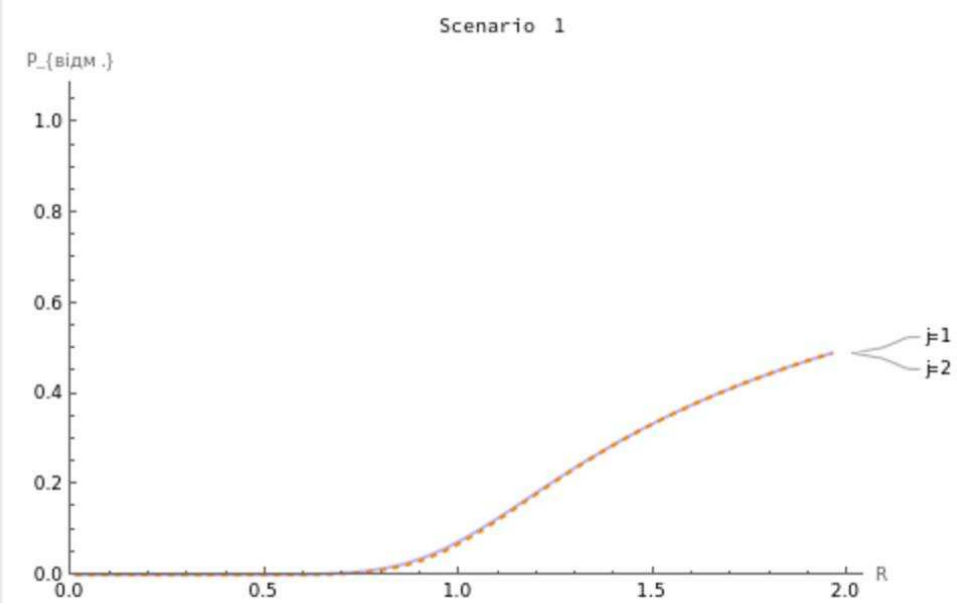

Рисунок 2.12 – Залежність ймовірності відмови від загального навантаження по категоріях при $n=2$, $m=10$, $m_1=0$, $m_2=0$

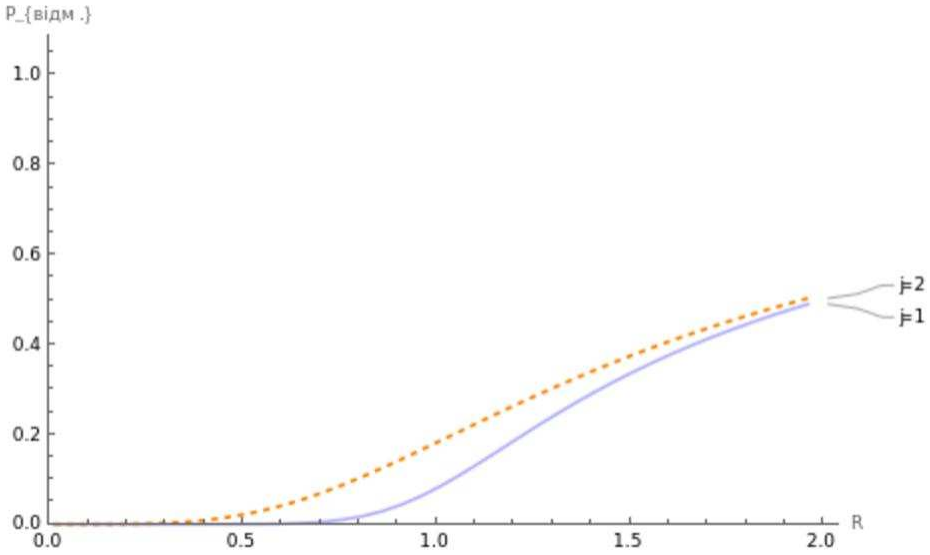

Рисунок 2.13 – Залежність ймовірності відмови від загального навантаження по категоріях при $n=2$, $m=0$, $m_1=9$, $m_2=1$

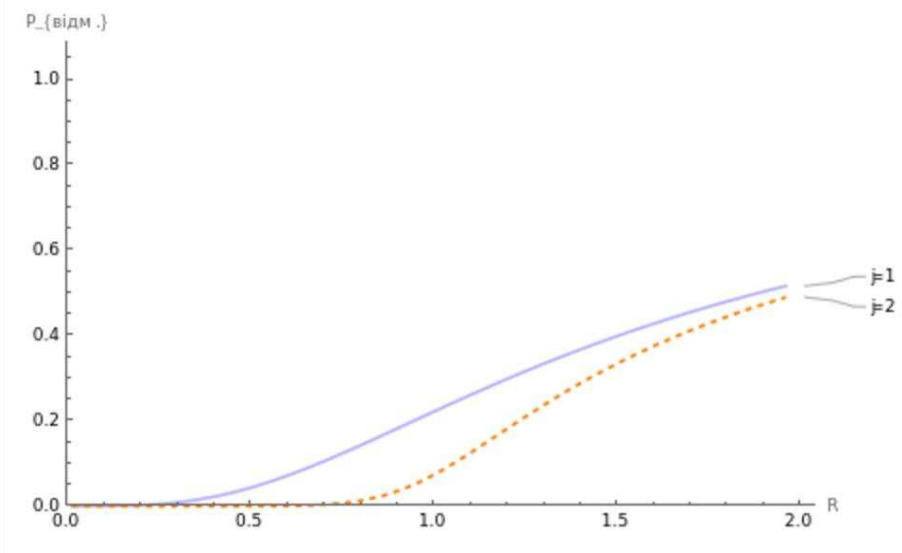

Рисунок 2.14 – Залежність ймовірності відмови від загального навантаження по категоріях при $n=2$, $m=0$, $m_1=1$, $m_2=9$

На рис. 2.15 наведено залежності ймовірності відмови від навантаження для стратегії пріоритезації у порівнянні з базовим вхідним потоком.

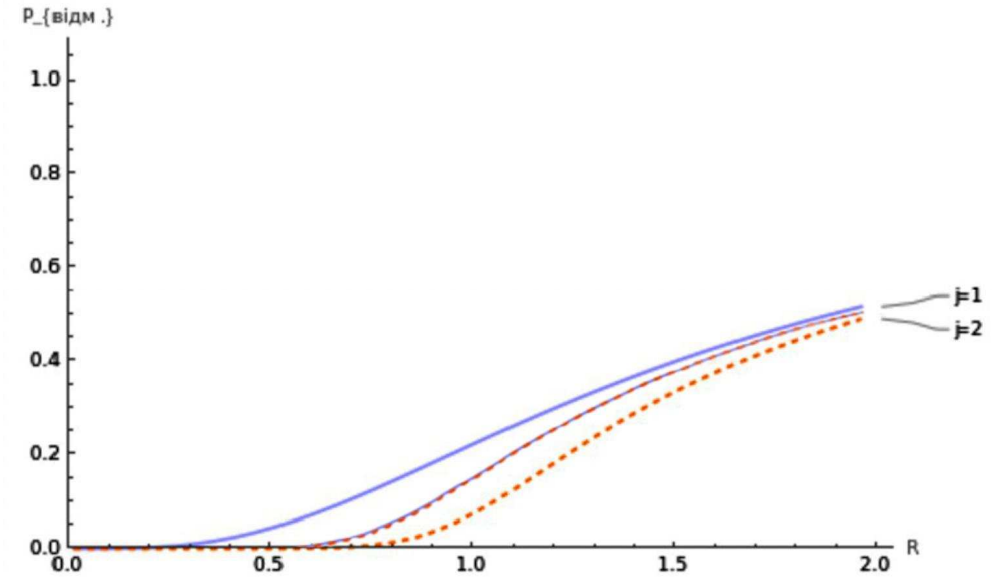

Рисунок 2.15 – Залежність ймовірності відмови від загального навантаження по категоріях при $n=2$, $m=0$, $m_1=1$, $m_2=9$

Як бачимо, за умов коли інтенсивність надходження задач суттєво перевищує пропускну здатність системи, пріоритезація певної категорії заявок призводить до суттєвих змін пропускної здатності системи. Це пояснюється

витісненням непріоритетних задач, а також збільшенням відсотку відмов для такої категорії. Відповідно, коли ми надаємо пріоритет меншим задачам, ми майже повністю перестаємо обслуговування великих задач, і навпаки.

Оскільки ми зацікавлені у збережені балансу розміру виконаних задач, щоб оптимізація не призвела до повного усунення всіх заявок, розподілених до класу більших, будемо розглядати балансні політики. Вони призводять до потрапляння в проміжок між лініями графіків пріоритету менших задач та без пріоритету [53]. Це дозволить збільшити загальну пропускну здатність системи, надаючи при цьому можливість оброблятись для більших задач. Модель штучного інтелекту може додатково покращити результати балансування навантажень завдяки використанню отриманого досвіду обробки подібних даних в минулому. Для застосування до складних задач та великих об’ємів даних варто прибігти до використання глибокого навчання нейронних мереж.

Висновки за розділом 2

1. Розглянуто алгоритм роботи багатоканальної системи масового обслуговування з обмеженою чергою.

2. Було розглянуто підходи до збільшення пропускної здатності системи масового обслуговування зі сталим потоком вхідних заявок. Проведено аналіз відповідності до потреб поставленої задачі. З наявних підходів обрано застосування системи пріоритетів до черги очікування. Такий підхід дозволить раціональніше використати вже наявні ресурси. Розібрані розповсюджені різновиди політик пріоритетів, а також їхні переваги та недоліки.

3. Задля забезпечення потреб та розв’язування поставленої задачі, було обрано комбіновану схему, що складається з: часу обслуговування, класів вимог, динамічного пріоритету, відносного пріоритету. Було запропоновано використання пріоритезації за розміром задач, що у свою чергу зменшує навантаження на чергу, адже кількість місць в черзі не залежить від часу обробки однієї задачі на відміну від процесу обслуговування. Відносні

пріоритети та заборона на переривання задачі, що вже виконується, мінімізують втрати часу, а також, дозволяють уникнути потреби відновлення виконання.

4. Перехід від початкової системи до такої що має пріоритети досягається завдяки розбиттю початкового вхідного потоку на класи пріоритетів за розміром. Введено термін сортувальника заявок, відповідальність якого полягає у визначенні розміру вимоги. Розглянуто два основних підходи до пріоритезації: самоідентифікація, оцінка на основі функції ймовірнісного розподілу. Оголошено переваги та недоліки обох підходів.

5. Було запропоновано розділити загальну чергу на окремі черги для різних класів розмірів, а також залишити буфер для безперебійної роботи багатоканальних систем. Завдяки такому рішенню, вдається уникнути заповнення черги великими задачами чи регулярної загибелі задач з меншим пріоритетом. Досягається баланс між загальним збільшенням продуктивності системи та наданням можливості виконуватись великим задачам. Використання пріоритету на основі часу виконання вимог, дозволило пропускати вперед менші задачі, але при цьому ми зберігаємо можливість обслуговування більших вимог. За таких обставин наш алгоритм не гарантує співпадіння послідовності надходження заявок до системи з послідовністю їх надходження на виконання, проте ми зберігаємо цю послідовність в рамках кожної окремої категорії задач. Неперервний контроль за станом черг дозволяє гнучко реагувати на зміни в поведінці вхідного потоку заявок. Запропоновано метрику, що дозволить спостерігати зсув балансу.

6. Було детально розібрано математичну модель для спрощеної системи: одноканальної, з необмеженою чергою, та за двома класами розмірів заявок. Виведено рівняння для знаходження граничних значень політик балансу. Розраховано умови масштабування, та описано сприятливі умови для застосування даного метода. Велика увага приділена пошуку порогового значення, за якого необхідно змінювати категорію для заявки.

7. Було проведено чисельні експерименти з замірами результатів застосування різних політик щодо пріоритетів. Порівняно продуктивність

правила двох класів пріоритетів з іншими еталонними політиками для черг $M/M/1$ з різними інтенсивностями трафіку. Ми спостерігаємо, що при порівнянні правила двох класів пріоритетів із FCFS, правило двох класів завжди забезпечує меншу довжину черги. Зокрема, процес довжини черги масштабується повільніше при збільшенні ρ за правилом двох класів пріоритетів порівняно з FCFS, і ця різниця у масштабуванні стає більш помітною, коли ρ наближається до 1. Що у свою чергу означатиме меншу ймовірність відмови для систем з обмеженою чергою.

Основні результати другого розділу опубліковано у роботах [80, 82, 83].

Список джерел, які використано у даному розділі, наведено у повному списку використаних джерел [1, 2, 5, 7, 9 – 11, 13, 15, 25, 27, 28, 37 – 39, 44 – 53].

3 ЗАСТОСУВАННЯ ВЗАЄМОДІЮЧОГО СПІВВИКОНАННЯ ДЛЯ МОДЕЛЮВАННЯ СИСТЕМ МАСОВОГО ОБСЛУГОВУВАННЯ

3.1 Комп’ютерне моделювання

3.1.1 Вимоги до програмного забезпечення

Теорія масового обслуговування виникла як окремий напрямок та породила цілий розділ теорії ймовірностей внаслідок свого практичного значення. Вона була покликана описати поведінку складних систем та надати математичні моделі для прогнозування обробки великих об’ємів заявок. Усі компоненти системи працюють незалежно один від одного, чітко розділяючи обов’язки. В момент виникнення певних подій, вони взаємодіють між собою шляхом обміну повідомленнями. Така комунікація може бути викликана, наприклад, надходженням заявки до системи, готовністю вимоги до прийняття в обробку або успішним опрацюванням заявки. При цьому у випадку багатоканальної системи, з’являється необхідність одночасної обробки вимог [53] на відміну від одноканальної, де всі дії, теоретично, можуть виконуватись послідовно.

При спробі програмно змоделювати поведінку СМО, ми стикаємось з необхідністю відтворення взаємодії між незалежними процесами, що комунікують між собою із подальшим масштабуванням до паралельної обробки вимог.

Метою даного розділу є обґрунтування доцільності застосування сучасних комп’ютерних інформаційних технологій, а саме – підходу CSP із використанням мови програмування Go для моделювання систем масового обслуговування. Також маємо на меті аналіз спорідненості інструментарію паралельного виконання даної технології до математичних узагальнень. Це, надалі, може відкрити шлях до значного вдосконалення інструментарію та

покращення ефективності програмних моделей, а також, наблизити програмну реалізацію до процесу, що моделюється.

Для початку необхідно сформулювати вимоги до програмного забезпечення (ПЗ), яке претендує на використання як інструмента при моделювання систем масового обслуговування.

Оскільки цифровізація світу неспинно набирає оберти, виникає потреба інтегрувати логіку моделювання СМО у вже існуючі рішення. Більшість сучасного програмного забезпечення пишеться популярними та сучасними мовами програмування високого рівня. Відповідно, першою вимогою буде наявність можливості використання ПЗ в якості залежності у цільовому програмному забезпеченні. Така взаємодія може бути досягнута кількома шляхами:

– підключення в якості бібліотеки або модуля;

– взаємодія за допомогою API (REST, JSON, RPC, GraphQL).

При цьому перевага надається можливості вбудувати логіку моделювання СМО безпосередньо до програми. Це зумовлено бажанням зменшити вплив мережевої взаємодії на результат моделювання.

Задля забезпечення можливості моделювання багатоканальних систем нам необхідна підтримка багатопотоковості на всіх рівнях виконання програми:

– рівні процесора, на якому запускається ПЗ;

– рівні операційної системи, на якій запускається ПЗ;

– рівні мови програмування, якою реалізоване ПЗ;

– рівні програмної реалізації.

Оскільки дослідження проводяться для навантажених систем, виникає потреба в раціональному використанні всіх ресурсів процесора. Забезпечивши оптимальне завантаження кожного окремого потоку операційної системи, ми забезпечимо збільшення ефективності використання багатоядерного процесора загалом. Зазвичай, такий ефект досягається за рахунок використання внутрішнього планувальника операцій на рівні мови програмування. Його задача полягає у балансуванні задач під час потоку виконання програми.

Кожна з операцій обробки заявки має бути подана у вигляді незалежного процесу, який за необхідності може бути масштабовано або замінено на аналог. Така модульність дозволить легко порівнювати різні стратегії застосовані до системи масового обслуговування.

Необхідною умовою також є підтримка обміну повідомленнями між кількома незалежними процесами. Така взаємодія має бути захищена від одночасного доступу до пам’яті задля уникнення випадків неузгодженості стану (datarace) [60, 68].

Більшість прикладних задач можуть використовувати складні структури даних. Відповідно, наступною вимогою стає підтримка композитних типів. Задля чистоти експерименту, а також оптимізації ресурсів нам необхідно мати жорстку типізацію на рівні мови програмування. Це дозволить виділяти не більше ніж необхідну кількість оперативної пам’яті та прискорить операції над даними. При цьому для забезпечення гнучкості нам необхідно мати у розпорядженні динамічні типи, які дозволять писати універсальний код.

Оскільки ми не можемо наперед знати логіку виконання досліджуваної системи, нам необхідна підтримка абстракцій та динамічних функцій, щоб логіку обробки даних визначав клієнт. Процес моделювання СМО має бути незалежним від логіки обробки заявки.

Нарешті, для проведення досліджень, нам знадобиться потужний набір інструментів вимірювання, оцінки та аналізу продуктивності програми. Мова йде про механізми, що збирають статистику про те, який процес, як довго оброблявся та на яких потоках. Також корисною буде інформація про об’єм задіяної оперативної пам’яті та кількість її алокацій. Такий функціонал дозволить нам вивчати та чисельно характеризувати кожен етап обробки заявки. Завдяки цьому ми матимемо підґрунтя для оптимізації алгоритму.

На даний момент існує багато інструментів для програмного моделювання систем масового обслуговування. Розглянемо найбільш розповсюджені варіанти та проведемо оцінку їхньої відповідності описаним критеріям.

3.1.2 Категорії існуючих рішень

Самостійні програми із власним графічним інтерфейсом відрізняються глибоко розвинутим функціоналом з моделювання різноманітних складних систем, підтримують трьохвимірні візуалізації відтворюваних процесів, мають велику кількість попередньо налаштованих заготовок для систем, що часто підлягають моделюванню. Однак в більшості випадків не мають можливості інтеграції в існуючі програми, оскільки, зазвичай, написані застарілими мовами програмування із слабкою підтримкою багатопотоковості. Велика кількість вбудованого функціоналу разом з можливостями приносить з собою і надмірну складність, збільшуючи поріг входження та зменшуючи швидкість розрахунків. Яскравими представниками цього класу застосунків є: AnyLogic, Arena Simulation та Simul8. Програмні рішення з цієї групи не підходять за кількома критеріями для нашого дослідження.

Другу групу утворюють математичні пакети та розширення для них (Mathcad, Matlab, Maple, Mathematica). Значною перевагою є потужне математичне ядро, яке дозволяє використовувати чисельні методи при розв’язуванні прикладних математичних задач. Також в них підтримуються побудова схем, та графіків. Однак вони програють у швидкодії, не мають можливості інтеграції та суттєво обмежують можливості оптимізації виконання. Зазвичай підтримка паралельних обчислень обмежується запуском асинхронних розрахунків без можливості впливати на сам процес.

Третя категорія включає в себе бібліотеки та пакети, які можна підключити у власну програму. Даний різновид рішень сильно залежить від мови програмування котрою реалізовані розширення. Тому розглядати кожен приклад необхідно разом з контекстом використання у результуючому програмному забезпеченні. Більшість з них були реалізовані довгий час тому і обмежені тими підходами та технологіями, що були доступні на момент створення. В наслідок чого, жоден з них в повній мірі не відповідає оголошеним раніше вимогам.

3.2 Паралелізм

3.2.1 Сутність паралелізму

Паралелізм – це властивість системи або алгоритму виконувати кілька операцій одночасно, розділяючи задачу на менші підзадачі, які можуть оброблятися незалежно. Паралелізм буває двох основних видів:

а) у математичному контексті – це поділ завдання на підзадачі, які можуть виконуватись незалежно або частково незалежно з метою зменшення часу розв’язування (такий підхід використовується в розподілених обчисленнях та великих симуляціях);

б) у програмуванні – це підхід до обробки, при якому завдання розбиваються на окремі процеси або потоки, що виконуються одночасно на багатоядерних процесорах, і паралелізм включає багатопотоковість, розподілені системи та асинхронне виконання, причому використовується для підвищення швидкодії програм, зокрема для складних обчислювальних задач.

При застосуванні багатоядерних процесорів кожне ядро може виконувати свою частину роботи паралельно з іншими ядрами. Це дозволяє суттєво прискорити виконання задач, що складаються з незалежних операцій, таких як обробка великих обсягів даних або моделювання фізичних процесів.

Види паралелізму:

– даних: однакові операції виконуються на різних фрагментах даних;

– задач: різні операції або задачі виконуються паралельно.

Паралелізм підвищує продуктивність шляхом максимального використання ресурсів багатоядерної архітектури процесора, що зменшує час виконання складних операцій [64].

Станом на сьогоднішній день кількість навантажених систем, що обслуговують потоки заявок, які описуються випадковими законами неперервно зростає. Така тенденція була присутня завжди, але довгий час постійне вдосконалення одноядерних процесорів за рахунок збільшення

кількості транзисторів перекривало потреби. Певний час тому збільшувати продуктивність окремих ядер процесорів стало вкрай проблематично – закон Мура перестав діяти. Логічним рішенням стала ідея паралельного виконання задач на окремих ядрах процесора та збільшення загальної продуктивності пристрою таким чином. Внаслідок цього виробники почали фокусувати увагу конструкторів на збільшені енергоефективності ядер, а також на одночасному розміщені більшої їх кількості на одному чипі. Одноядерні процесори були майже повністю витіснені з ринку багатоядерними аналогами. Тренд на зменшення технічного процесу виробництва кристалів процесорів, а також поступовий перехід на ARM архітектуру зумовлені бажанням збільшити автономність портативних пристроїв. Багатопотоковість при цьому продовжує відігравати важливу роль як в енергоефективному, так і в продуктивному режимах роботи.

Втім, сам по собі багатоядерний процесор не надає жодних переваг. Оскільки для ефективного використання його потенціалу необхідна підтримка багатопотоковості тим програмним забезпеченням, що буде на ньому виконуватись [79]. При цьому застосунки не мають прямої взаємодії з процесором. Натомість операційна система (ОС), що виконує роль медіатора, має також підтримувати багатопотоковість. За умови використання багатоядерного процесора та ОС з гарною підтримкою багатопотоковості при спробі запустити програму, що працює виключно в одному потоці та написану більшістю мов програмування, ми будемо спостерігати приблизно такі самі показники швидкодії, як і у випадку з одноядерним процесором. Структура взаємодії компонентів багатопотоковості зображена на (рис. 3.1).

З моменту появи багатоядерних процесорів, об’єми запитів почали зростати із значно більшою швидкістю, ніж можливості з масштабування ресурсів, що використовуються при обробці даних, поновлюючи тим самим запит на оптимізацію програмного продукту для більш ефективного використання наявних ресурсів із урахуванням паралелізму.

Рисунок 3.1 – Піраміда багатопотоковості

Відмінністю сучасних мов програмування є те, що вони не взаємодіють з процесором напряму. Натомість вони використовують високорівневі абстракції. Самі по собі програми є набором команд до операційної системи для подальшої обробки процесором. Відповідно підхід та рівень підтримки багатопотоковості мови програмування залежить від тієї філософії та інструментарію, що було закладено в її основу. Під інструментами розуміються наступні компоненти:

– співвідношення потоків програми до потоків операційної системи;

– механізми синзронізації;

– спосіб комунікації між потоками.

3.2.2 Планування потоків

Серед мов програмування поширені три підходи взаємодії з потоками операційної системи:

– один до багатьох;

– один до одного;

– багато до багатьох.

Один до багатьох – N потоків програми виконуються на одному потоці

операційної системи (рис. 3.2). Перевага такого способу взаємодії полягає у тому, що вся необхідна пам’ять виділяється в одному кеші процесора. Відповідно, коли ядро відновлює виконання задачі після паузи, вся необхідна інформація знаходиться у швидкому доступі. Завдяки цьому ми отримуємо надзвичайно швидке переключення контексту виконання програмних потоків, але за такого підходу ми обмежуємо можливість програми ефективно використовувати багатоядерні процесори.

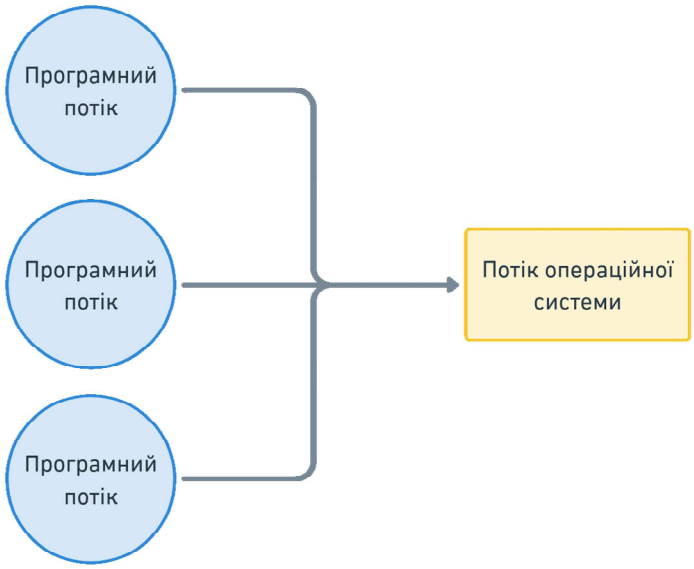

Рисунок 3.2 – Планування потоків один до багатьох

Один до одного – один потік програми обслуговується одним потоком ОС (рис. 3.3). Внаслідок цього ми можемо використовувати багатоядерні процесори, але переключення контексту виконання вимагає суттєвих часових витрат, що уповільнює виконання програми. Такий підхід може краще працювати, коли є відносна невелика кількість потоків та задачі є достатньо складними, щоб виправдати їх паралельну обробку.

Багато до багатьох – довільне число програмних потоків може виконуватись на довільному числі потоків операційної системи (рис. 3.4). Дана методика планування навантаження є відносно новою. Вона підтримується у сучасних мовах програмування.

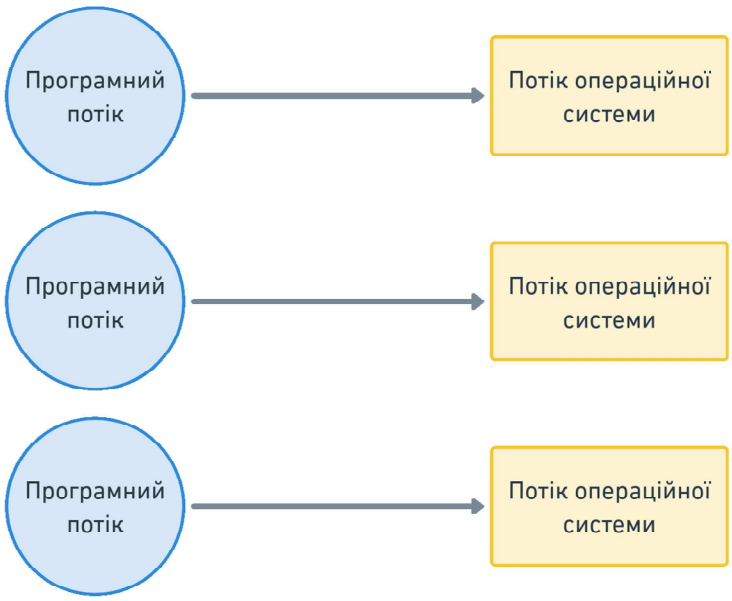

Рисунок 3.3 – Планування потоків один до одного

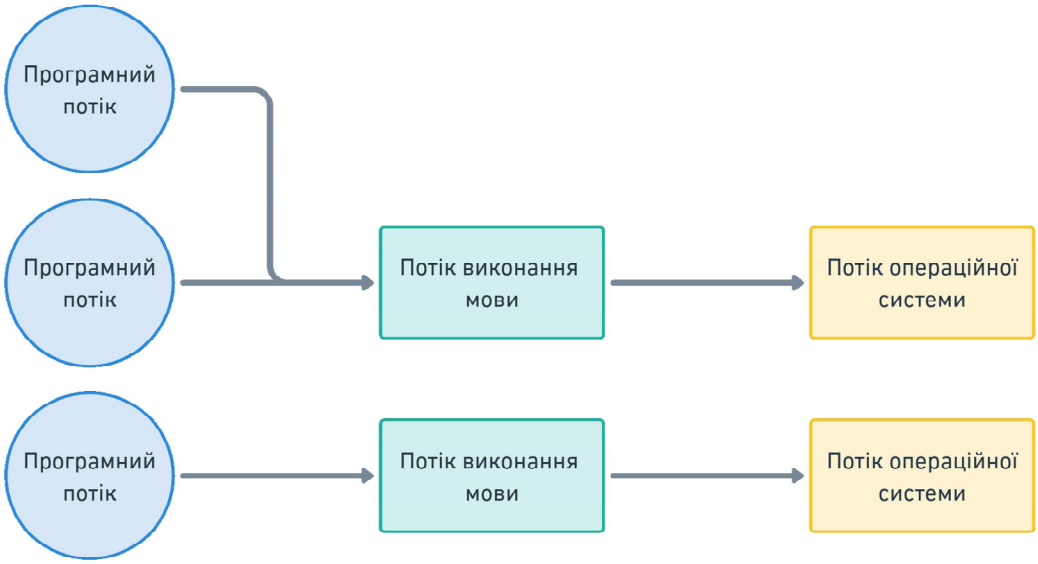

Рисунок 3.4 – Планування потоків багато до багатьох

Головна ідея полягає у спробі поєднати переваги двох попередніх підходів, при цьому уникнувши їх недоліків. Для такого підходу характерна можливість швидко перемикатись між контекстом виконання програмних потоків при повноцінному використанні усього потенціалу багатоядерного процесора. Завдяки цьому програма, написана із використанням мови з даним типом багатопотоковості, має виконуватись значно швидше за умови використання менших програмних потоків.

3.2.3 Механізм синхронізації

В різновидах паралелізму, згаданих раніше, з’являється необхідність обмінюватись даними між програмними потоками. Така взаємодія може бути необхідною для забезпечення наступних потреб:

– подача даних, необхідних для обробки в потоках;

– повернення результатів обробки або помилок;

– обмін проміжними результатами (коли різні етапи обробки виконаються паралельно).

При застосуванні асинхронної обробки ми втрачаємо можливість передбачати послідовність виконання паралельних потоків. Відповідно при звичайному (не захищеному) обміні даних отримаємо невизначені стани. Це призведе до відсутності повторюваності результатів програми, а також до отримання помилкових результатів.

Дана проблема вирішується за допомогою різноманітних підходів синхронізації:

– організований спільний доступ до пам’яті;

– блокуючий доступ до пам’яті;

– блокуючі сигнали;

– обмін повідомленнями.

Організований доступ до пам’яті є відносно швидким, але вимагає величезних зусиль до забезпечення вірної організації взаємодії з пам’яттю. Разом з тим він є вкрай небезпечним та вразливим до помилок. Такий підхід не рекомендований до використання в більшості випадків.

Блокуючий доступ до пам’яті є найрозповсюдженішим підходом до взаємодії між кількома асинхронними процесами. Більш того він лежить в основі усіх наступних рішень. Використання полягає в тому, що на час взаємодії з блоком пам’яті даний ресурс блокується для доступу іншими процесами. Як результат маємо одну операцію над станом в один момент часу.

Блокуючі сигнали є надбудовою навколо блокуючого доступу до пам’яті.

Ідея полягає в тому, щоб мати спосіб сповіщати асинхронні процеси про певну подію в системі. Така синхронізація є безпечною та дозволяє оркеструвати велику кількість паралельних потоків одночасно.

Обмін повідомленнями, технічно, поєднує в собі плюси усіх раніше згаданих підходів і є високорівневим інструментом синхронізації. Він дозволяє надсилати повідомлення з даними разом зі сповіщенням про асинхронну дію. Даний підхід є рекомендованим до використання у сучасних програмах завдяки своїй гнучкості та безпечності.

При цьому всі інструменти синхронізації вимагають їх вірного застосування. Результатом помилкової організації синхронізуючих алгоритмів може бути стан взаємного блокування (deadlock) програми. Так може статися у випадку, якщо усі асинхронні процеси очікують розблокування і жоден з них не може продовжити своє виконання.

3.2.4 Категоризація мов

Важливим чинником при програмному моделюванні систем масового обслуговування є те, у якій мірі механізми мови програмування відповідають критеріям математичного апарату. Одна з основних розбіжностей між мовами програмування, що має вплив на доцільність використання для моделювання СМО, криється у підході до паралельного виконання. Розглядатимемо виключно популярні мови програмування, адже важливу роль відіграє можливість розвивати та підтримувати програмний продукт у довготривалій перспективі. Згідно з рейтингом, побудованим шляхом аналізу вакансій на роль програмного інженера за 2023 р., в табл. 3.1 наведено 10 мов, придатних для написання серверів, що мають долю більше 1%. Серед наведених технологій можна виділити кілька груп:

– підтримка асинхронності в рамках одного потоку операційної системи;

– підтримка роботи з потоками операційної системи напряму;

– підтримка взаємодіючого співвиконання за допомогою розширень,

доданих нещодавно у якості сторонньої бібліотеки;

– повна підтримка взаємодіючого співвиконання та інтеграція ідей CSP на рівні ядра мови.

Мови програмування, які донедавна використовуються для написання програм, у тому числі і для моделювання систем масового обслуговування, не мали підтримки CSP та взаємодіючого співвиконання. Зараз впроваджується часткова підтримка, яка, тим не менш, не надає повноти функціоналу [60]. Такий стан справ надає можливість застосувати нові підходи до програмного моделювання процесів із використанням сучасних мов програмування.

Таблиця 3.1 – Класифікація популярних мов програмування.

Назва	Доля	Підхід до багатопотоковості
JavaScript / TypeScript	29,8%	асинхронність
Python	19,64%	багатопотоковість за моделлю один до багатьох
Java	17,78%	багатопотоковість за моделлю один до одного (з версії 21 додано підтримку взаємодіючого співвиконання за моделлю один до одного без підтримки обміну повідомленнями)
C#	12,21%	багатопотоковість за моделлю один до одного
PHP	9,39%	асинхронність
C / C++	9,14%	багатопотоковість за моделлю один до одного
Ruby	4,37%	асинхронність
Go	1,91%	повна реалізація CSP, взаємодіючого співвиконання за моделлю багато до багатьох з підтримкою обміну повідомленнями

Головною ідеєю, що була закладена у сучасні мови програмування, впроваджених за останні 10 років, є забезпечення інженерів можливістю розкрити весь потенціал процесорів нового покоління. Підтримка ефективного

паралелізму є одним з ключових напрямків покращень в мовах програмування задля збільшення продуктивності кінцевого програмного забезпечення.

Як можна помітити, підтримка багатопотоковості за моделлю багато до багатьох є недостатньо поширеною серед популярних мов програмування. Однією з основних причин є відносна новизна проблеми яку даний підхід покликаний вирішити. Першою мовою яка зробила підхід взаємодіючого співвиконання популярним стала мова Go. Її успіх підштовхнув розробників інших мов до введення аналогічних рішень. Однак мова Go так і залишається лідером у сфері багатопотоковості серед сучасних технологій.

3.3 Взаємодіюче співвиконання

3.3.1 Визначення процесів

Процеси є фундаментальними математичними абстракціями, що описують взаємодію систем як між собою, так і з зовнішнім середовищем. Вони є послідовністю подій, які система виконує відповідно до певного сценарію. Процеси можуть використовуватись для моделювання таких об’єктів, як годинники, лічильники або торгові автомати, які виконують визначені дії у взаємодії з зовнішнім середовищем та ін.

Основною характеристикою кожного процесу є його алфавіт – це множина всіх подій, які можуть відбутися під час його виконання [55]. Ці події є атомарними діями, що не мають тривалості, але є ключовими моментами взаємодії між процесом і середовищем. Події можуть бути ініційовані як самим процесом, так і зовнішнім середовищем. Це визначає поведінку системи та її реакції на зовнішні стимули. Алфавіт процесу є фіксованим і включає всі можливі варіанти дій.

Подія – це основна одиниця взаємодії в межах процесу, яка не має тривалості й є миттєвою дією. Вона може бути викликана як внутрішніми факторами процесу, так і зовнішніми подіями в середовищі, з яким процес взаємодіє. Це

забезпечує структуру поведінки системи, де кожна подія може впливати на наступні кроки процесу.

Для опису взаємодії між процесами використовується кілька важливих операцій, таких як префіксація, рекурсія та вибір.

Префіксація – це оператор, який вказує на послідовність дій, що здійснює процес. Він дозволяє описати, як одна дія відбувається перед іншою в рамках певної послідовності [58]. Це особливо важливо для визначення порядку взаємодій у складних системах, де важливо чітко фіксувати послідовність дій. Наприклад, процес може спочатку виконати одну подію, а потім перейти до іншої дії.

Рекурсія використовується для моделювання процесів, які можуть повторювати свої дії тривалий час або нескінченно. Вона дозволяє описати процеси, які безперервно виконують певні дії, як, наприклад, годинник, що постійно відраховує час. Рекурсія є важливим інструментом для опису систем, які виконують циклічні дії або працюють у тривалих сценаріях.

Вибір дозволяє процесу приймати рішення на основі зовнішніх подій або взаємодії з середовищем. Це забезпечує адаптивну поведінку процесу, де система може реагувати на різні зовнішні стимули та змінювати свою поведінку відповідно до обставин [58]. Наприклад, у системах, що взаємодіють із користувачем, вибір дозволяє виконувати різні дії залежно від введених даних.

Процеси можуть взаємодіяти один з одним у рамках складних систем, де кілька підпроцесів виконуються паралельно та обмінюються подіями через певні канали зв’язку. Це дозволяє моделювати великі системи, де окремі компоненти можуть працювати незалежно, але при цьому мають взаємодіяти один з одним для досягнення спільної мети.

Описані операції, такі як префіксація, рекурсія та вибір, забезпечують чітке формальне моделювання процесів. Алгебраїчні закони, що застосовуються до цих операцій, дозволяють уникнути проблем паралелізму, таких як взаємне блокування чи змагання за ресурси. Це забезпечує передбачуваність поведінки системи та підвищує її надійність [62].

Таким чином, концепція процесів дозволяє будувати стабільні, надійні та передбачувані системи, що працюють у паралельних середовищах. Моделювання таких систем забезпечує узгоджену поведінку процесів навіть у складних умовах багатозадачності.

3.3.2 Визначення взаємодіючого співвиконання

Взаємодіюче співвиконання (concurrency) у математиці та інформатиці визначається як властивість систем, в яких кілька обчислень виконуються одночасно та потенційно взаємодіють один з одним. Термін, введений C. A. R. Hoare у рамках його роботи з “Communicating Sequential Processes” (CSP) [58], став основоположним для розуміння комплексних систем, де багато процесів відбуваються одночасно та асинхронно. CSP моделює поведінку системи за допомогою алгебри процесів, дозволяючи описати взаємодію між паралельними процесами через події комунікації.

Розвиток теорії CSP привів до впровадження поняття взаємодіючого співвиконання у високорівневих мовах програмування та архітектурах систем. Поняття паралелізму в CSP та його математична модель дозволяють проєктувати й аналізувати складні системи, які включають паралельні процеси та взаємодію через обмін повідомленнями. Теорія взаємодіючого співвиконання має застосування в багатьох галузях, включно із розробкою операційних систем, розподіленими обчисленнями та проєктуванням мікросхем. Вона допомагає забезпечити взаємну незалежність, синхронізацію та уникнення взаємоблокувань між паралельними процесами. Даний аспект математики та комп’ютерних наук постійно розвивається, оскільки нові парадигми та методології надходять для кращого розуміння та ефективнішого використання паралелізму в обчислювальних системах [61].

Взаємодія процесів є ключовим елементом взаємодіючого співвиконання. Коли два або більше процесів виконуються одночасно, вони можуть взаємодіяти через спільні події, які вимагають синхронної участі всіх задіяних

процесів. Така взаємодія полягає у ініціалізації одним процесом події та подальшої одномоментної реакції на подію іншими процесами. Це означає, що взаємодія відбувається лише тоді, коли всі процеси-учасники готові до виконання спільної події. Виключенням є ініціалізація події зовні системи, коли для взаємодії достатньо, щоб приймаюча сторона була готова до обробки події.

Наприклад, у сценарії автоматизованої системи продажу товарів, подія видачі товару відбудеться тільки тоді, коли і автомат готовий видати товар, і покупець готовий його отримати [55]. Це приклад точок синхронізації, де процеси взаємодіють одночасно. Така координація подій дозволяє створювати складні системи, де підпроцеси взаємодіють один з одним, обмінюючись інформацією або сигналами через спільні події [62].

При цьому важливою властивістю взаємодії процесів є те, що вони можуть зберігати незалежність до моменту спільної події. Це дозволяє паралельно працювати над власними завданнями, але в критичні моменти узгоджувати свої дії із застосуванням подій. Важливою умовою успішної взаємодії процесів є відсутність конфліктів або взаємного блокування, оскільки гармонійна синхронізація дозволяє уникати таких проблем. У разі відсутності узгодженості дій можливі проблеми, такі як взаємне блокування (deadlock) або конкуренція за ресурси (data race) [60, 68], що може призвести до того, що процеси не зможуть завершити свої дії.

Таким чином, взаємодія процесів у конкурентних системах вимагає чіткої координації через спільні події, які забезпечують точну синхронізацію дій. Це дозволяє уникати типових проблем паралелізму, таких як взаємне блокування конкуренція за ресурси або загублені процеси (проблема за якої втрачаються всі способи зв’язку з процесом), забезпечуючи ефективну роботу всієї системи.

Наявність взаємодіючого співвиконання у системі означає, що кілька процесів можуть бути запущені одночасно і виконувати свої завдання незалежно. Це дозволяє підвищити гнучкість і ефективність системи, адже різні компоненти можуть виконуватися паралельно, але узгоджувати свої дії в критичні моменти через спільні події. Наприклад, модель «обідаючих

філософів» [58] демонструє типовий випадок роботи процесів за моделлю взаємодіючого співвиконання, де кожен філософ є незалежним процесом, але всі вони мають узгоджено використовувати спільні ресурси (виделки). Це ілюструє типові проблеми, пов’язані з синхронізацією в паралельних системах, такі як блокування ресурсів або нескінченне очікування, коли один із процесів постійно перебуває в очікуванні ресурсу, який заблоковано іншим процесом.

Математична теорія детермінованих процесів надає формальну основу для опису поведінки систем, де всі події мають чітко визначений порядок виконання. Детерміновані процеси характеризуються передбачуваністю: для кожної події існує лише один можливий результат, що дозволяє уникати станів невизначеності. Це особливо важливо для систем, що вимагають точної послідовності дій, таких як банківські системи або критичні інфраструктури, де кожна складна операція повинна виконуватись строго за визначеним сценарієм.

Для опису процесів із повторюваними або рекурсивними діями використовується концепція фіксованих точок. Вона дозволяє формалізувати процеси, що повторюються нескінченну кількість разів або працюють циклічно, і гарантує їхню передбачувану поведінку. Це критично важливо для багатьох систем, де повинна зберігатися стабільність і надійність виконання.

Таким чином, взаємодіюче співвиконання забезпечує можливість паралельної роботи процесів у системі, де вони можуть виконувати свої завдання незалежно, але водночас взаємодіяти через спільні події. Для підтримки передбачуваної поведінки в таких системах використовується детермінізм, а математичні методи, такі як фіксовані точки, дозволяють формалізувати поведінку рекурсивних процесів.

3.3.3 Комунікація та синхронізація

Комунікація між процесами глибоко пов’язана з концепцією синхронізації. Синхронізація забезпечує узгоджене виконання дій між процесами, що є критичним для коректної взаємодії, особливо у системах, де

процеси комунікують через канали передачі даних. Синхронізація розглядається як фундаментальний механізм для забезпечення послідовності та узгодженості в асинхронних системах.

У реалізації комунікації через канали, операції надсилання та отримання повинні бути синхронізовані. Це означає, що операція передачі повідомлення (надсилання) і операція прийому (отримання) не можуть відбутися незалежно одна від одної. Подія передачі даних блокується до того моменту, коли інший процес буде готовий прийняти повідомлення. Ця важлива властивість забезпечує гарантію того, що процеси обмінюються даними у визначеному порядку, що запобігає втратам або некоректній обробці інформації. Наприклад, при передаванні повідомлення через канал, подія надсилання не завершиться до того часу, поки інший процес не здійснить прийом цього повідомлення. В такий спосіб забезпечується повна синхронізація між процесами.

Розглядається кілька основних механізмів для досягнення синхронізації між процесами:

– комунікаційні канали (channels) є основним механізмом для синхронізації процесів (канали забезпечують синхронізоване введення і виведення між процесами, коли один процес відправляє дані, а інший – отримує, і оскільки події відправлення і отримання взаємозалежні, це забезпечує узгоджене виконання дій між процесами);

– труби (pipes): труби, як механізм передачі даних між послідовними процесами, також вимагають синхронізації, при цьому кожен процес у ланцюжку труб обробляє дані лише після того, як попередній процес завершує їхню передачу, що забезпечує послідовність дій і ефективну передачу даних без втрат;

– механізми префіксації також служить для забезпечення порядку виконання подій у рамках процесів і дозволяють чітко визначати, які події повинні відбутися спочатку, а які – пізніше, забезпечуючи правильну послідовність взаємодій між процесами;

– субординація: у цьому випадку один процес підпорядковується іншому,

що також може бути формою синхронізації, при цьому підпорядкований процес діє у межах дозволених йому подій і не може виконати жодної дії без дозволу або ініціації головного процесу, що дозволяє чітко структурувати взаємодію і забезпечувати синхронізацію між підлеглим і головним процесами.

Вводяться поняття трасерів (tracers) та відмов (failures), що дозволяють формально описувати поведінку процесів під час синхронізації. Трасери відслідковують послідовність подій у процесах і дозволяють визначити, чи правильно синхронізовані процеси в системі. Відмови ж описують сценарії, коли синхронізація між процесами не відбувається через певні порушення або несподівані зміни у процесі взаємодії. Це дозволяє формалізувати поведінку системи та виявляти потенційні проблеми синхронізації на ранніх етапах проєктування системи.

Синхронізація є ключовим компонентом при комунікації між процесами. Вона забезпечує послідовне виконання дій і запобігає небажаним результатам, таким як втрати даних або неправильна обробка. Різні механізми, такі як комунікаційні канали, труби, префіксація та субординація, дозволяють досягти чіткої і надійної синхронізації в паралельних системах. Формальні методи, зокрема трасери та відмови, дають змогу строго аналізувати поведінку системи і забезпечувати її передбачувану роботу.

3.3.4 Взаємодіюче співвиконання в мовах програмування

Взаємодіюче співвиконання є важливою складовою сучасного програмування, що забезпечує ефективне використання обчислювальних ресурсів і можливість виконання кількох завдань одночасно. З розвитком багатоядерних процесорів і розподілених систем цей підхід став особливо актуальним, оскільки дозволяє програмам безпечніше та ефективніше масштабуватися та виконувати паралельні операції. Взаємодіюче співвиконання застосовується для розв’язання багатьох практичних задач: обробка великих обсягів даних, асинхронні операції з мережею, додатки з

обчисленнями у реальному часі та багатопотокові обчислення.

У різних мовах програмування були розроблені різні моделі реалізації взаємодіючого співвиконання. Вони варіюються від класичних потоків до більш легковагових підходів, таких як coroutine (у мові Kotlin) чи goroutine (у мові Go). Головна мета таких моделей – дозволити програмам безпечно виконувати кілька завдань одночасно без конфліктів при доступі до спільних ресурсів, запобігаючи таким проблемам, як гонки даних або блокування. Це робить взаємодіюче співвиконання незамінним інструментом для розробки масштабованих та ефективних програм у сучасних умовах.

Серед сучасних та розповсюджених мов програмування можна виділити три технології, що реалізують ідеологію взаємодіючого співвиконання: Go, Java, Kotlin.

Structured concurrency (структуроване взаємодіюче співвиконання) у Java 21 – це підхід до асинхронного програмування, що зберігає природні відносини між задачами та підзадачами. Це призводить до здобуття більш зрозумілого, підтримуваного та надійного коду для роботи з паралельними процесами. Термін «structured concurrency» був введений M. Sustrik та популяризований N. J. Smith. Ідеї з інших мов програмування, таких як ієрархічні супервізори в Erlang, вплинули на дизайн обробки помилок у structured concurrency. Даний підхід базується на простому принципі: якщо задача розділяється на підзадачі, то всі вони повертаються в одне й те саме місце, а саме в блок коду задачі.

У structured concurrency підзадачі працюють від імені основної задачі. Основна задача очікує результатів від підзадач і відстежує їх на предмет помилок. Подібно до структурних методів програмування для коду в одному потоці, сила structured concurrency для багатопотокових систем походить із двох ідей:

а) чітко визначених точок входу та виходу для виконання коду в блоці;

б) суворого вкладення часу життя операцій, яке віддзеркалює їх синтаксичне вкладення у коді.

Оскільки точки входу та виходу коду добре визначені, час життя підзадачі

обмежується синтаксичним блоком її батьківської задачі. Оскільки час життя підзадач вкладений у час життя їхньої батьківської задачі, вони можуть розглядатися та управлятися як єдиний блок. Відповідно, час життя батьківської задачі також вкладений у час життя її батьківської задачі, система часу виконання може перетворити ієрархію задач на дерево, яке є асинхронним аналогом стеку викликів у одному потоці. Це дозволяє застосовувати політики, такі як дедлайни, до цілого піддерева задач і дозволяє інструментам спостереження представляти підзадачі як підпорядковані їхнім батьківським задачам.

Structured concurrency відмінно підходить для віртуальних потоків, які є легкими потоками, реалізованими у JDK. Багато віртуальних потоків можуть використовувати один і той же системний потік, що дозволяє використовувати величезну кількість віртуальних потоків. Окрім чисельності, віртуальні потоки є достатньо дешевими, щоб представляти будь-яку асинхронну одиницю поведінки, навіть ту, що включає операції вводу та виводу. Це означає, що серверний додаток може використовувати structured concurrency для обробки тисяч або мільйонів вхідних запитів одночасно: він може присвятити новий віртуальний потік для обробки кожного запиту, і коли задача розгалужується, запускаючи підзадачі для асинхронного виконання, вона може присвятити новий віртуальний потік кожній підзадачі. У процесі відносини між завданням і підзадачею формуються в дерево шляхом створення посилання на унікального батька для кожного віртуального потоку, подібно до того, як кадр у стеку викликів посилається на свого унікального викликача.

У підсумку віртуальні потоки забезпечують велику кількість потоків. Structured concurrency може правильно і надійно їх координувати, а інструменти спостереження можуть показувати потоки так, як їх розуміє розробник. Наявність API для structured concurrency у JDK полегшить створення підтримуваних, надійних та контрольованих серверних додатків.

Coroutine у Kotlin є механізмом для роботи з асинхронним і неблокуючим кодом, заснованим на ідеї взаємодіючого співвиконання. Вони концептуально схожі на класичні потоки у тому сенсі, що виконують блок коду, який працює

одночасно з рештою коду. Однак coroutine не прив’язана до певного потоку: вона може призупинити своє виконання в одному потоці та продовжити в іншому. Coroutine можна розглядати як легкий потік, але є кілька важливих відмінностей, через які їхнє використання в реальному житті суттєво відрізняється від потоків. Основна перевага coroutine полягає в тому, що написання асинхронного коду виглядає майже так само, як і синхронного, з використанням відомих конструкцій, таких як цикли і обробка виключень, без необхідності освоювати нові специфічні API. Coroutine дотримуються принципу structured concurrency, що означає, що нові coroutine можуть запускатися тільки в межах певного контексту або області видимості (coroutine scope), який обмежує час життя coroutine та визначає їхній життєвий цикл. Ця модель гарантує, що всі дочірні коррутини завершаться, перш ніж вийти з зовнішнього контексту. Завдяки цьому вдається уникнути витоків ресурсів і забезпечує чітке управління життєвим циклом усіх підзадач. За допомогою coroutine Kotlin може забезпечувати ефективне управління асинхронними завданнями без прив’язки до потоків операційної системи. Це робить coroutine потужним інструментом для масштабованих і продуктивних додатків, оскільки вони дозволяють одночасно запускати безліч задач із мінімальними витратами на системні ресурси.

Канали в Kotlin Coroutines надають спосіб передавати потік значень між coroutine. Вони схожі на BlockingQueue, але мають деякі ключові відмінності. Наприклад, замість того, щоб додавати елемент у чергу, відправник надсилає його в канал, а приймач отримує елемент не з черги, а з каналу.

Flow – це тип у Kotlin, який може послідовно видавати кілька значень на відміну від функцій із підтримкою призупинення, які повертають лише одне значення. Це «холодний» потік, що означає, що код у конструкторі потоку не виконується, поки потік не буде зібраний.

Оскільки Kotlin працює на базі JVM, як і Java, йому також доступне використання structured concurrency починаючи з версії JVM 21. Хоч coroutine у Kotlin з’явились раніше ніж structured concurrency у Java, вони все ще є

бажаним підходом до імплементації багатопотоковості в Kotlin.

Goroutine в Go є легковаговими потоками виконання, які управляються потоком виконання програми Go. Вони дозволяють запускати кілька функцій асинхронно без створення нових системних потоків, що робить їх набагато дешевшими у створенні та управлінні порівняно з класичними потоками. Для створення goroutine використовується ключове слово go, яке дозволяє функції виконуватись паралельно з іншими частинами програми. При створені нових goroutine вони, як і у випадку structured concurrency в Java та Kotlin, утворюють дерева залежностей. Це дає можливість ефективно використовувати обчислювальні ресурси та створювати високомасштабовані системи. Однією з основних та відмінних складових моделі взаємодіючого співвиконання Go є канали, які слугують засобом для комунікації та синхронізації між goroutine. Канали дозволяють безпечно обмінюватися даними між goroutine, забезпечуючи синхронізацію їхньої роботи без необхідності явно використовувати м’ютекси або інші засоби блокування потоків. Це дозволяє розробляти програми на базі взаємодіючого співвиконання без ризику гонок даних або інших типових проблем багатопоточних систем. Більш того, це максимально наближає мову Go до реалізації підходів CSP, завдяки наявності ефективного механізму комунікації та нотифікації між незалежними процесами. Модель взаємодіючого співвиконання в Go є простою у використанні, але водночас надзвичайно ефективною для вирішення реальних задач. Поєднання goroutine і каналів дозволяє писати безпечний і масштабований код, який може використовувати багатоядерні процесори для паралельного виконання завдань. Мову Go відрізняють від Java та Kotlin ряд особливостей таких як: простота, відсутність JVM, перевага ефективності goroutine та каналів.

3.3.5 Взаємодіюче співвиконання в CMO

Розглянемо застосування підходів взаємодіючого співвиконання до

вирішення поставленої задачі. Згідно із термінологією CSP багатоканальна система масового обслуговування з обмеженою чергою та відмовами може бути подана у вигляді мережі процесів, що комунікують між собою. Кожен етап, або стан, у якому перебуватиме заявка під час обробки, є процесом. Перехід між станами відбуватиметься під впливом відповідних подій або сигналів. При цьому, згідно із визначенням взаємодіючого співвиконання, всі ці процеси можуть відбуватися асинхронно для кількох заявок в один момент часу. Для початку виділимо наступні незалежні процеси, що супроводжують обробку заявки у СМО:

– надходження заявок може мати різні розподіли ймовірностей та інтенсивність;

– утримання в черзі може мати додатковий функціонал визначення пріоритетів;

– обробка каналом обслуговування може мати випадкову тривалість, а також ймовірність помилки;

– покидання заявкою системи, що не залежить від результату обслуговування.

Формалізуємо взаємодію між даними процесами через обмін повідомленнями за допомогою подій, які описуватимуть переходи між станами заявки:

– надходження (arrive): подія, що означає прибуття чергової заявки до системи та є початковою подією;

– взяття до черги (onqueue): у випадку, якщо у черзі є вільні місця, заявка успішно стає на очікування;

– покидання черги (dequeue): у разі звільнення хоча б одного каналу обслуговування заявка, що стоїть наступною у черзі, може покидати чергу, звільнюючи місце для подальших надходжень;

– початок обробки (start): канал розпочинає обслуговування заявки;

– успішне закінчення обробки (success): опрацювання заявки пройшло у штатному режимі;

– помилка під час обробки (fail): з певних причин канал не зміг надати очікувану послугу;

– відмова (reject): у черзі на момент надходження не було вільних місць;

– базова подія закінчення обробки для кінцевих алгоритмів (STOP).

Користуючись визначеною множиною подій, сформулюємо наступний алфавіт процесу:

$$\alpha SMO = \{\text{arrive, onqueue, dequeue, start, success, fail, reject}\}.$$

Тоді процес обробки заявки матиме наступний вигляд [62]:

$$SMO = \text{arrive} \to (\text{reject} \to STOP \mid \text{onqueue} \to \text{dequeue} \to \text{start} \to$$
$$\to (\text{success} \to STOP \mid \text{fail} \to STOP)).$$

Таким чином, ми бачимо, що системи масового обслуговування, які розглядаються в рамках дослідження, можуть бути цілісно описані за допомогою CSP та взаємодіючого співвиконання. При цьому за рахунок абстракцій, а також комунікації через події, ми отримуємо можливість гнучко масштабувати та розширювати взаємодію компонентів без необхідності змін в самих процесах. Даний процес може бути запущено паралельно, що не змінить його загальну будову.

Висновки за розділом 3

1. Було сформульовано вимоги до програмного забезпечення, необхідного для вирішення поставленої задачі оптимізації системи масового обслуговування. В процесі аналізу перелічені категорії програмного забезпечення, обговорені переваги та недоліки кожного. Серед оголошених рішень жодне в повній мірі не задовольняє критеріям. Прийнято рішення запропонувати новий підхід.

2. Одна з ключових властивостей для моделювання багатоканальних систем масового обслуговування є паралелізм. Ми підняли питання розвитку про-

грамного паралелізму, проаналізували його структуру та різновиди. Було введено класифікацію. При цьому було приділено особливу увагу процесу планування потоків виконання та механізмам синхронізації. Проведено категоризацію мов програмування за ознаками паралелізму, що вони реалізують.

3. Було детально розглянуто суть підходу взаємодіючого співвиконання. Визначені терміни: процесів, комунікації, синхронізації в рамках методології взаємодіючого співвиконання. Проаналізована спорідненість взаємодіючого співвиконання з процесами, що відбуваються при моделюванні СМО. Запропоновано застосування взаємодіючого співвиконання для моделювання систем масового обслуговування. Розглянуто мови програмування, в яких реалізується дана парадигма, а також функціонал, що вони надають для реалізації синхронізації та комунікації.

Основні результати третього розділу опубліковані у роботах [79, 80, 81].

Список джерел, які використано у даному розділі, наведено у повному списку використаних джерел [55, 58, 61 – 70, 79].

4 РОЗРОБКА АРХІТЕКТУРНОГО РІШЕННЯ ТА ІМПЛЕМЕНТАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Постановка вимог

Першим, і безперечно базовим кроком при роботі над якісним програмним забезпеченням є його архітектурне рішення. Адже, як дім не будують без креслень – програмний продукт не пишуть без архітектури. Саме архітектура системи визначає компоненти, що прийматимуть участь в обробці даних, їхню будову, а також спосіб взаємодії [73]. Вона є чітким рецептом, що буде однаково сприйнятий будь-яким інженером і дозволить уникнути непорозумінь при реалізації задуманого. Відповідно, ми уникаємо залежності від конкретних особистостей, розблоковуючи потенціал до розширення команди розробки, а також нівелюємо ризики, пов’язані зі зміною ключових членів колективу інженерів.

Разом з тим, на початку роботи над новим продуктом у більшості випадків існує тільки часткове розуміння потреб та призначення майбутнього програмного забезпечення. Це пов’язано зі складністю проєктованих систем та незліченною кількістю сторонніх чинників, що впливають на них. У зв’язку з цим розуміння реальних потреб виникає на етапі проєктування під впливом дослідження проблематики, а також правильної постанови запитань до бізнес доменів [74].

Не можна забувати й про подальший розвиток програмного забезпечення. Згодом дуже ймовірно постане потреба в одному або декількох одночасно підкласів масштабування:

– збільшення навантаження та пропускної здатності окремих частин програми;

– зміна або розширення користувацьких потреб з подальшою надбудовою функціоналу;

– розширення команди або декомпозиція зон відповідальності з подальшою делегацією повноважень між кількома автономними командами розробки.

Кожен з наведених різновидів масштабування потребує завчасно закладеного підґрунтя, що надає можливість безшовної, поетапної розбудови програмного продукту без необхідності глибокої переробки вже існуючого функціоналу [75].

Разом з тим передбачити абсолютно всіх потенційних сценаріїв розвитку подій неможливо. Відповідно, пошук балансу між гнучкістю та складністю є черговою задачею, що закладено у процес побудови архітектури. Зазвичай вирішення полягає у завчасному застосуванні абстракцій, що відокремлюватимуть конкретну реалізацію від функціоналу, що очікується.

Гарна ізоляція компонентів дозволяє підвищити надійність та стійкість програмного забезпечення до помилок виконання, а також знизити потенційний вплив змін, що вносяться в окремі частини програми, локалізуючи їх у рамках фіксованих абстракцій [73].

Додатковим приводом до того, щоб приділити достатньо уваги проєктуванню перед початком роботи, є ціна виправлення помилки. На етапі побудови архітектури задля виправлення недоліків проєктування необхідно переписати документацію та внести зміни до діаграм і блок-схем на відміну від кількості часу та зусиль, що вимагаються для модифікації вже існуючої програми. І чим більш пізня фаза розробки розглядається – тим складніше буде виправити недолік [73]. Отже, приділивши більше часу й зусиль до розробки архітектури програмного продукту, ми значно економимо час та ресурси, необхідні на внесення правок у майбутньому.

Окрім того, цілком можливо, що під час проєктування або незабаром, після закінчення процесу побудови архітектури, проєкт буде визнаний недоцільним. Адже, знайдеться інший шлях до вирішення проблеми, суттєво дешевший або швидший. Відповідно, задача архітектурного рішення полягає, зокрема, у загальній оцінці витрат, ризиків, а також в обґрунтуванні

необхідності саме такого шляху до забезпечення потреб.

Роль маяку доповнює перелік задач, що покликана вирішувати архітектура програмного забезпечення. Доволі часто трапляється, що під час роботи над імплементацією проєкту виникають нові виклики та необхідно підлаштовуватись під змінені потреби [74]. Архітектура системи забезпечує орієнтир, що має використовуватись для звірки відповідності чергового рішення загальному курсу й стратегії розвитку.

За визначенням архітектура програмного забезпечення зобов'язана задовольняти потреби чітко визначеної задачі. Саме тому є вкрай важливим приділити достатню увагу специфікації проблеми, що вирішується. У нас на меті стоїть побудова програмного забезпечення для моделювання поведінки систем масового обслуговування (СМО). При цьому є велика кількість різновидів СМО, що поділяються за наступними критеріями [54]:

– наявність одного чи більшої кількості каналів обслуговування, що надають послугу;

– наявність або відсутність черги очікування для новоприбулих заявок на випадок, якщо всі наявні канали обслуговування зайняті;

– скінченність або необмежений розмір черги очікування;

– політика пріоритизації вимог, що визначає, яка саме вимога з очікуючих буде приступати до обслуговування в разі звільнення каналу;

– наявність обмежень часу перебування в системі або часу обслуговування.

У рамках дослідження нас цікавить у першу чергу моделювання багатоканальної системи з обмеженою чергою та відмовами новоприбулим вимогам у разі відсутності місця для очікування. Але при цьому ми закладатимемо можливість розширення та підтримки всіх інших різновидів СМО у майбутньому. Кінцевою метою програмного продукту є універсальна програма, що надає можливість моделювати різноманітні процеси, порівнювати поведінку систем зі змінними режимами та параметрами [43]. Не останню роль гратиме швидкодія моделювання, яка має надати можливість отримувати

швидше результат відтворення поведінки складних систем та оптимізувати процес завдяки результатам експериментів. Допоміжним підходом у досягненні мети буде застосування паралельних обчислень, а залучення підходу взаємодіючого співвиконання наблизить поведінку моделі до природнього досліджуваного процесу.

У даному розділі ми перелічимо ті чіткі вимоги, які ми встановлюємо як ключові для результуючого програмного забезпечення.

1) Можливість перед початком роботи програми визначити основні характеристики системи, що описуватимуть її поведінку, а саме – кількість каналів обслуговування, ємність черги. На даному етапі моделюються СМО з обмеженою чергою.

2) Можливість визначати максимальний час перебування заявки у системі, а також граничну тривалість обслуговування. У випадку перевищення даного часу обробка переривається та вимога покидає систему з помилкою.

3) Обслуговування полягає у виконанні функції, заданої перед початком роботи програми. Вона є єдиною для всіх вимог і використовує для розрахунків вхідні параметри, що визначаються заявкою. Сама функція може мати довільну логіку, але має очікувати вхідні параметри й повертати результат обробки або помилку.

4) Забезпечити підрахунок та збір інформації щодо кількості невиконаних задач задля подальшого аналізу ефективності системи.

5) Обслуговування черги заявок у послідовності їхнього надходження (FIFO).

6) Співпадіння послідовності повернення результатів виконання вимог із їхнім надходженням не гарантується системою.

Сформульовані таким чином вимоги вписуються у визначення систем масового обслуговування та мають надавати вичерпний функціонал для моделювання широкого спектру процесів [43].

4.2 Мова програмування Go

4.2.1 Походження мови Go

Go – це сучасна компільована мова програмування з відкритим кодом, яка полегшує створення простого, надійного та ефективного програмного забезпечення. Мова Go була розроблена у вересні 2007 року Robert Griesemer, Rob Pike та Ken Thompson, усі з яких працювали в Google, і була анонсована в листопаді 2009 року. Метою мови та її інструментів було забезпечення виразності, ефективності як при компіляції, так і при виконанні, а також ефективності в написанні надійних та стійких програм [63]. Вона була створена задля задоволення нагальних потреб та в якості відповіді на виклики у сфері комп'ютерної інженерії, що постали перед розробниками програмного продукту у XXI столітті, а саме:

– неперервного надходження значної кількості одночасних запитів із подальшою паралельною їх обробкою;

– необхідності підтримки великої різноманітності процесорів, платформ та операційних систем;

– потреби швидко реалізовувати бізнес ідеї та раніше отримувати відгук про відповідність очікувань;

– бажанні спростити написання, розширення, підтримку, а також читання коду програм;

– ідеї децентралізованої спільноти розробників і популяризації пакетів із відкритим кодом.

На поверхневому рівні Go нагадує мову C і, як і C, є інструментом для професійних програмістів, що досягає максимального ефекту з мінімальними засобами. Проте Go набагато більше, ніж просто оновлена версія C [67]. Вона запозичує та адаптує вдалі ідеї з багатьох інших мов, уникаючи тих особливостей, які призводять до складності та ненадійного коду. Її можливості для паралельного програмування є новими та ефективними, а підхід до

абстракції даних і об’єктно-орієнтованого програмування вирізняється незвичайною гнучкістю. Go має автоматичне керування пам’яттю або збір сміття.

Також вона особливо добре підходить для створення інфраструктури, як-от мережевих серверів, інструментів і систем для програмістів, проте вона є справді універсальною мовою і знаходить застосування в таких різних галузях, як графіка, мобільні додатки та машинне навчання. Go набуває популярності як заміна нетипізованих скриптових мов, оскільки вона поєднує виразність із безпекою: програми на Go зазвичай працюють швидше, ніж програми, написані на динамічних мовах, і зазнають значно менше аварій через неочікувані помилки типів [67].

Go – це проєкт з відкритим кодом, тому вихідний код його компілятора, бібліотек та інструментів є вільно доступним для кожного. Внески в проєкт робляться активною міжнародною спільнотою.

Go працює на системах, схожих на Unix – Linux, FreeBSD, OpenBSD, Mac OS X – а також на Plan 9 та Microsoft Windows. Програми, написані в одному з цих середовищ, зазвичай працюють без змін на інших.

Як і біологічні види, успішні мови породжують нащадків, які включають переваги своїх предків; схрещування іноді приводить до несподіваних переваг; а час від часу з’являється радикально нова особливість, що не має прецеденту. Ми можемо багато дізнатися про те, чому мова є такою, якою вона є, і для якого середовища вона була адаптована, досліджуючи ці впливи.

На рис. 4.1 наведено схему, яка показує найважливіші впливи попередніх мов програмування на розробку Go.

Go іноді описують як «мову, схожу на C» або «C для 21 століття» [63]. Від мови C мова Go успадкувала синтаксис виразів, оператори керування потоком, базові типи даних, параметри, що передаються за значенням, вказівники та, перш за все, акцент на програми, які компілюються в ефективний машинний код і природно взаємодіють з абстракціями сучасних операційних систем.

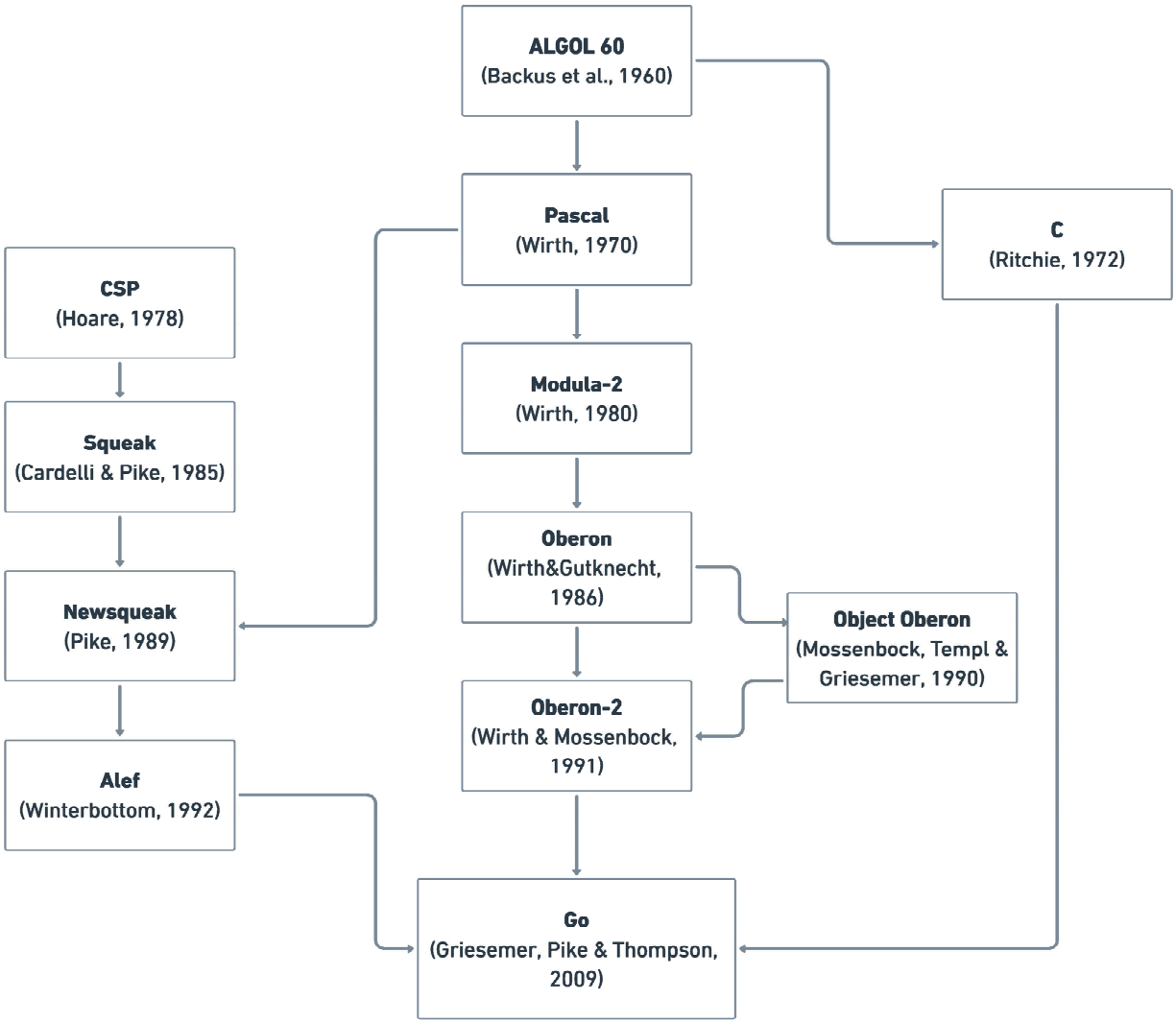

Рисунок 4.1 – Дерево еволюції до Go

Однак у родоводі Go є й інші предки. Один із важливих потоків впливу походить від мов Ніклауса Вірта, починаючи з Pascal.

Modula-2 надихнула концепцію пакетів. Oberon усунула розподіл між файлами інтерфейсів модулів та файлами реалізації модулів. Oberon-2 вплинула на синтаксис для пакетів, імпорту та оголошень, а Object Oberon забезпечила синтаксис для оголошень методів.

Інша важлива лінія серед предків Go, що робить його унікальним серед сучасних мов програмування, – це послідовність маловідомих дослідницьких мов, розроблених у Bell Labs, усі з яких були натхнені концепцією комунікацій послідовних процесів (CSP) з основоположної роботи C. A. R. Hoare 1978 року про основи паралельного програмування.

У CSP програма є паралельною композицією процесів, які не мають спільного стану; процеси комунікують і синхронізуються за допомогою каналів.

Але CSP, що розроблена C. A. R. Hoare [58], була формальною мовою для опису фундаментальних концепцій паралелізму, а не мовою програмування для написання виконуваних програм.

Rob Pike та інші почали експериментувати з реалізаціями CSP як справжніми мовами програмування. Першою була мова під назвою Squeak («Мова для спілкування з мишами»), яка забезпечувала обробку подій миші та клавіатури за допомогою статично створених каналів. За нею пішла Newsqueak, яка мала синтаксис операторів і виразів, схожий на C, та нотацію типів, подібну до Pascal. Це була чисто функціональна мова зі збором сміття, орієнтована на керування подіями клавіатури, миші та вікон. Канали стали значеннями першого класу, які динамічно створювалися та зберігалися у змінних [63].

Операційна система Plan 9 розвинула ці ідеї у мові під назвою Alef. Мова Alef намагалася зробити Newsqueak життєздатною мовою системного програмування, але відсутність автоматичного збирання сміття зробила роботу з паралельністю занадто складною.

Інші конструкції в Go показують вплив генів, не пов’язаних із прямими предками; наприклад, iota походить від APL, а лексичний обсяг із вкладеними функціями – від Scheme (та багатьох інших мов).

Тут також є нові мутації. Інноваційні зрізи (slices) в Go забезпечують динамічні масиви з ефективним довільним доступом, а також дозволяють складні схеми спільного використання, подібні до зв’язаних списків. Крім того, оператор defer є нововведенням Go.

4.2.2 Кроскомпіляція

При створенні Go розробники хотіли мати можливість запускати програми, написані нею, на різних платформах без змін у кодовій базі. Але при

цьому вони прийняли рішення відійти від розповсюдженого підходу використання віртуальної машини. Натомість було запропоновано підхід кроскомпіляції. Ідея полягає у двоетапній компіляції, коли спочатку програма перетворюється на GoAssembly, а потім вона перекладається на асемблер під необхідну комбінацію операційної системи та архітектури процесора. Наразі підтримуються усі розповсюджені комбінації платформ. У табл. 4.1 наведений перелік найпопулярніших з них, але він не є повним. Більшість екзотичних архітектур процесорів підтримуються під ОС Linux. Що виправдано відсутністю користувачів інших операційних систем для такого обладнання. Також, цікавою особливістю мови є надміру швидкий процес компіляції у порівнянні з іншими мовами.

Таблиця 4.1 – Підтримувані комбінації архітектури процесорів та ОС

	Android	Darwin	Ios	Js	Linux	Windows
386	+				+	+
amd64	+	+	+		+	+
amd64p32						
arm	+				+	+
arm64	+	+	+		+	+

4.2.3 Модель пам’яті

Мова Go належить до жорстко типізованого виду. Умовно можна поділити всі типи змінних на три категорії:

– примітиви;

– складні структури;

– абстракції (динамічні типи).

При цьому абстракції у свою чергу бувають як інтерфейсами (абстракція поведінки), так і дженеріки (абстракція пам’яті), а складні структури можуть включати в себе як поля примітиви, абстракції або інші складні структури [67].

Така модель повністю задовольняє потреби сучасного програмування.

Водночас, в мові Go відсутній класичний підхід ООП. Його було принесено в жертву спрощенню. Справа в тому, що Go створювалась з ідеєю простоти, тобто надати рівно стільки функціональності, скільки необхідно для вирішення поставленої задачі. Світові тенденції свідчать про поступове відмирання застосування наслідування на користь композиції. Отже, в Go сформовано новий погляд на старі проблеми та спрощено підхід ООП як такий, що додає невиправдану складність програмам.

Підхід Go до моделі пам’яті аналогічний загальним принципам мови: мета полягає в тому, щоб зберігати семантику простою, зрозумілою та корисною. У цьому розділі наводиться загальний огляд підходу, який буде достатнім для більшості програмістів. Формальна специфікація моделі пам’яті представлена у наступному розділі.

Гонки даних визначаються як одночасний запис у комірку пам’яті під час читання або запису до цієї ж комірки, якщо всі ці операції не є атомарними операціями доступу до даних, що забезпечуються пакетом sync/atomic [68]. Як вже зазначалося, програмістам настійно рекомендується використовувати відповідні механізми синхронізації для уникнення гонок даних. За відсутності гонок даних програми Go поводяться так, ніби всі goroutine працювали на одному процесорі, що іноді називають DRF-SC: програми без гонок даних виконуються в послідовно узгоджений спосіб.

Хоча програмісти повинні писати програми Go без гонок даних, існують обмеження щодо того, як Go може реагувати на гонки даних [72]. Реалізація мови може завжди реагувати на гонку даних шляхом її фіксації та завершення програми. В іншому випадку кожне читання пам’яті розміром у одне слово або менше має спостерігати значення, яке дійсно було записане в цю комірку (можливо, паралельно виконуваною goroutine) і яке ще не було перезаписане. Ці обмеження роблять Go більш подібним до Java або JavaScript, де більшість гонок даних мають обмежену кількість результатів, і менш подібним до C та C++, де сенс будь-якої програми з гонками даних повністю невизначений, а

компілятор може виконати що завгодно. Підхід Go спрямований на те, щоб зробити помилкові програми більш надійними та легшими для налагодження, водночас наполягаючи на тому, що гонки є помилками, які можуть бути виявлені та зафіксовані інструментами.

Наступне формальне визначення моделі пам'яті Go тісно повторює підхід, представлений H. J. Boehm та S. V. Adve у статті «Основи моделі пам'яті для паралельного програмування на C++», опублікованій у PLDI 2008. Визначення програм без гонок даних та гарантії послідовної узгодженості для програм без гонок даних є еквівалентними наведеним у цій роботі.

Модель пам'яті описує вимоги до виконання програм, які складаються з виконання goroutine, що у свою чергу складаються з операцій пам'яті.

Операція пам'яті моделюється за допомогою чотирьох аспектів [70]:

– її типу, який вказує, чи є це звичайне читання даних, звичайний запис даних чи синхронізуюча операція, така як атомарний доступ до даних, операція з м'ютексом або операція з каналом;

– її місця у програмі;

– комірки пам'яті або змінної, до яких здійснюється доступ;

– значень, які операція читає або записує.

Деякі операції пам'яті є подібними до операцій читання, зокрема читання, атомарне читання, блокування м'ютекса та отримання даних з каналу. Інші операції є подібними до операцій запису, зокрема запис, атомарний запис, розблокування м'ютекса, надсилання через канал та закриття каналу. Деякі операції, такі як атомарна операція порівняння і обміну (compare-and-swap), поєднують у собі властивості операцій читання та запису.

Виконання goroutine моделюється як набір операцій пам'яті, які виконує одна goroutine.

Вимога 1: операції пам'яті у кожній goroutine повинні відповідати правильному послідовному виконанню цієї goroutine з урахуванням значень, які читаються і записуються в пам'ять. Це виконання повинно бути узгодженим із відношенням «впорядковане до» (sequenced before), яке визначається як

частковий порядок, заданий специфікацією мови Go для конструкцій керування потоком, а також для порядку обчислення виразів.

Виконання програми на Go моделюється як набір виконань goroutine разом із відображенням W, яке визначає операцію запису, з якої кожна операція читання отримує дані (різні виконання однієї програми можуть мати різні виконання програми) [63].

Вимога 2: для заданого виконання програми відображення W, обмежене лише синхронізуючими операціями, повинно бути пояснене деяким неявним повним порядком синхронізуючих операцій, який є узгодженим із порядком виконання та значеннями, що читаються та записуються цими операціями.

Відношення «синхронізоване до» є частковим порядком синхронізуючих операцій пам'яті, виведеним із W. Якщо операція читання синхронізації r отримує значення з операції запису синхронізації w (тобто, якщо $W(r) = w$), то w синхронізується перед r. Неофіційно відношення «синхронізоване до» є підмножиною неявного повного порядку, зазначеного в попередньому пункті, обмеженого інформацією, яку безпосередньо спостерігає W.

Відношення «відбувається перед» (happens before) визначається як транзитивне замикання об'єднання відношень «впорядковане до» та «синхронізоване до».

Вимога 3: для звичайної (несинхронізуючої) операції читання r на комірці пам'яті x $W(r)$ повинна бути операція запису w, яка є видимою для r, де видимість означає, що обидві наступні умови виконуються:

а) w відбувається перед r;

б) w не відбувається перед жодною іншою операцією запису w' (на x), яка відбувається перед r.

Гонка читання-запису на комірці пам'яті x складається з операції читання r на x та операції запису w на x, принаймні одна з яких не є синхронізуючою, і які не впорядковані за відношенням «відбувається перед» (тобто ні r не відбувається перед w, ні w не відбувається перед r) [68].

Гонка запис-запис на комірці пам'яті x складається з двох операцій

запису w та w' на x, принаймні одна з яких не є синхронізуючою, і які не впорядковані за відношенням «відбувається перед».

Зазначимо, що якщо немає гонок читання-запису або запис-запису на комірці пам'яті x, то будь-яке читання r на x має лише одне можливе $W(r)$: єдину операцію w, яка безпосередньо передує йому в порядку «відбувається перед».

Загалом можна показати, що будь-яка програма на Go, яка є вільною від гонок даних, тобто не має жодного виконання програми з гонками читання-запису або запис-запису, може мати результати, які можна пояснити деяким послідовно узгодженим чергуванням виконань goroutine [68]. Ця властивість називається DRF-SC.

Метою формального визначення є забезпечення гарантії DRF-SC для програм без гонок даних, яка надається іншими мовами, включаючи C, C++, Java, JavaScript, Rust та Swift.

Деякі операції мови Go, такі як створення goroutine та виділення пам'яті, діють як операції синхронізації. Їхній вплив на частковий порядок «синхронізоване до» задокументований у розділі «Синхронізація» нижче. Окремі пакети несуть відповідальність за надання аналогічної документації для власних операцій.

Обговорюючи модель пам'яті, обрану для мови, неможна забувати про те, як ресурс, виділений під зберігання даних, вивільнюється, коли в ньому немає більше потреби. У мові Go присутні два типи пам'яті: стек та купа (рис. 4.2).

Стек (stack) – є ресурсом швидкого доступу, він виділяється локально на кожен програмний потік та вивільнюється автоматично як тільки закінчує виконання блоку, в рамках якого був наданий. Об'являючи змінні як можна ближче до фактичного їх місця використання, ми дозволяємо компілятору зменшувати час оренди пам'яті [66]. Оскільки стек є швидким та легко прогнозованим, компілятор за замовченням намагається виділити ресурс під зберігання змінних саме на ньому.

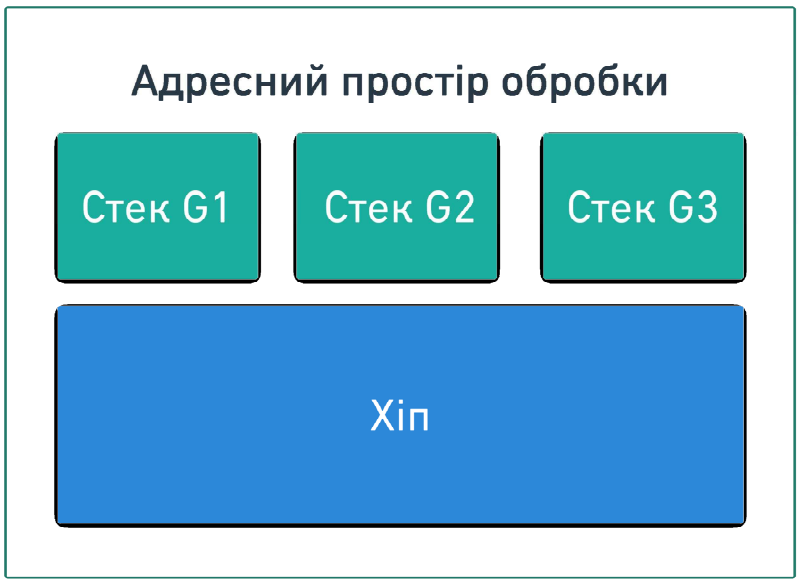

Рисунок 4.2 – Схема пам’яті в Go

В мові Go, об’єм пам’яті, що має стек goroutine, при старті роботи дорівнює всього двом кілобайтам. Однак в подальшому він зростає за необхідністю. В такий спосіб вдається мінімізувати пам’ять та час при створенні великої кількості маленьких goroutine, що і є однією з ідей закладених в мову.

Купа (heap) – це загальний простір пам’яті. На ньому виділяється пам’ять у випадку, коли важко або неможливо спрогнозувати межі використання змінної. Так може статись у разі, якщо замість значення у функцію передається покажчик. В результаті чого компілятору важко передбачити як довго буде потреба в доступі до значення змінної [63]. Іншим випадком може бути виділення пам’яті під ресурси спільного користування.

Купа сама по собі не відчищається, натомість існує процедура, яка виконується в потоці виконання програми: збірник сміття (garbage collector). Його роль полягає в тому щоб вивільнювати та дефрагментувати пам’ять, до якої більше немає потреби звертатись. У мові Go збірник сміття працює за алгоритмом mark & sweep (помітити та звільнити). Як тільки кількість виділеної на купі пам’яті досягає 200% від об’єму після завершення попередньої очистки, запускається процес розмітки. Його задача полягає у виявленні пам’яті, посилань на яку більше не залишилось в програмі. Після

закінчення даної фази збірник сміття призупиняє на мить виконання програми для остаточного видалення даних, після чого відновлює штатну роботу. Графік відчистки пам’яті застосунку зображено на рис. 4.3.

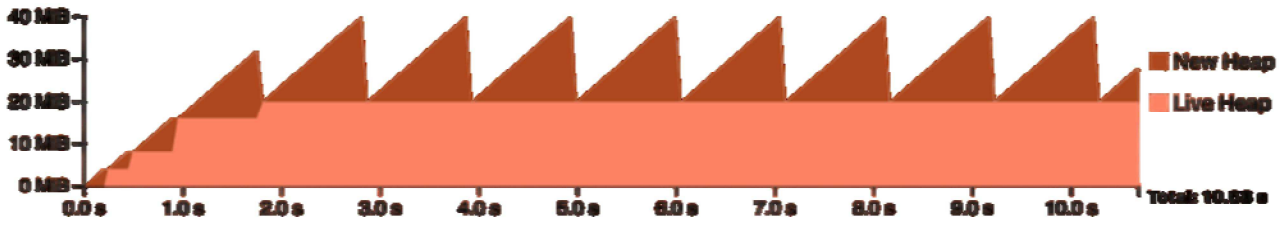

Рисунок 4.3 – Процес прибирання сміття

4.2.4 Багатопотоковість та потік виконання програми

Однією з проблем, що постає під час дизайну мови програмування, є задача про багатопотоковість. Оскільки, врешті решт, логіка буде виконуватись на потоках операційної системи, популярним підходом було пряме поєднання програмних потоків із потоками ОС. Останнім часом все більше мов додають абстракцію пула потоків. Цей прошарок відповідальний за розподіл задач між потоками в залежності від їхнього завантаження. Однак взаємодія все ще відбувається на рівні потоків ОС. Серед плюсів такого рішення – його відносна простота. Однак програми стають залежними від того, яким чином операційна система виділяє їм процесорний час. У мові Go розробники взяли за основу ідею про легкі, маленькі програмні потоки, які не мають прямого виходу на рівень операційної системи [63]. Значною відмінністю мови програмування Go є її підхід до організації потоку виконання програми (runtime) та планувальник (scheduler) – механізм оркестрації goroutine. Система планування багатопотоковості має наступні компоненти:

– потік ядра процесора (Core);

– потік операційної системи (M), який взаємодіє з потоком процесора;

– програмний потік (P) – абстракція рівня потоку виконання програми

Go, що зазвичай відповідає одному потоку операційної системи;

– goroutine (G) – маленький автономний процес, який описує бізнес логіку;

– допоміжні процеси потоку виконання програми – операції, що є goroutine самі по собі, і забезпечують життєдіяльність програми;

– планувальник – алгоритм, який обирає коли та якій goroutine надати можливість виконання, а також який тред (потік) операційної системи буде її обслуговувати.

Механізм взаємодії компонентів планувальника (рис. 4.4) працює наступним чином:

– при використанні у програмному коді директиви go із подальшим вказанням, яка функція має виконуватись, потік виконання програми створює маленьку goroutine з власним стеком пам’яті мінімального розміру;

– планувальник визначає локальну чергу програмного потоку, до якої потрапить goroutine на очікування;

– коли потік буде готовий взяти на виконання чергову goroutine – він бере її в роботу;

– є перелік умов, за яких виконання goroutine може бути тимчасово призупинено з міркувань надання часу іншим гору тинам, при цьому goroutine, що була попередньо в обробці, «паркується» до наступного вікна можливостей;

– у випадку закінчення очікуючих goroutine у локальній черзі програмного потоку він виконає спробу забору половини черги goroutine у іншого більш завантаженого програмного потоку [72];

– раз у певний час (або якщо всі локальні черги пусті) забір відбувається з глобальної черги.

Головна ідея такого підходу – надання додаткового рівня абстракції для щільнішого планування роботи потоків операційної системи. Даний механізм забезпечує дію підходу взаємодіючого співвиконання навіть за наявності лише одного ядра процесора [64].

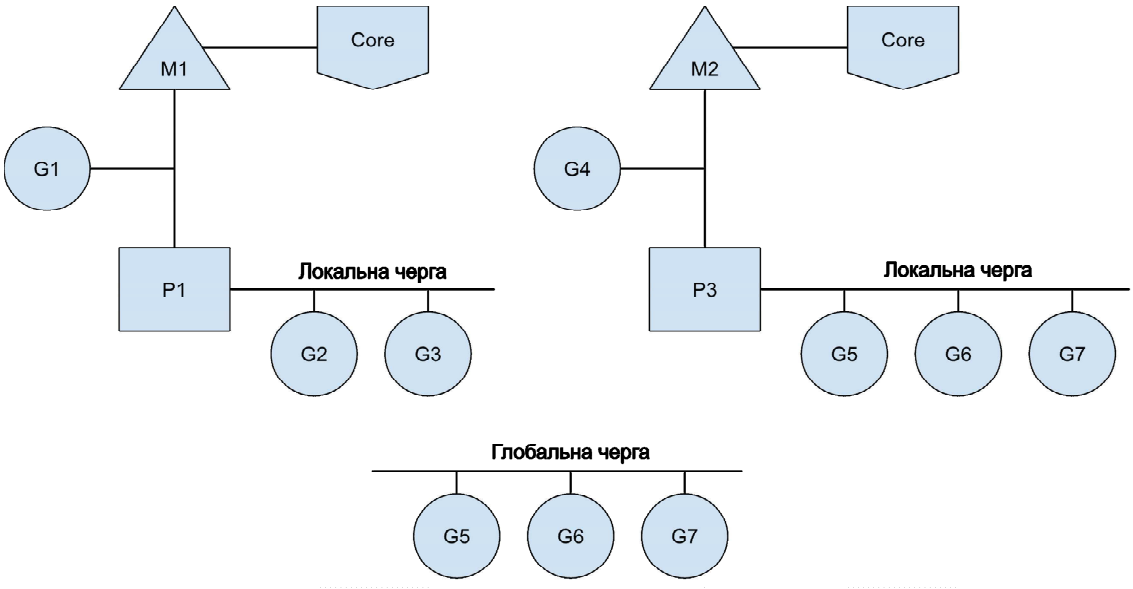

Рисунок 4.4 – Компонентна схема планувальника

Найважливішою перевагою у сукупності факторів є загальна простота взаємодії елементів, масштабування та потенціал до паралельного виконання на багатоядерних процесорах [65]. Аналогічним чином можна розширювати можливості [66].

При застосуванні паралельних обчислень завжди присутні накладні витрати. Мова йдеться про час та ресурс необхідний для переключень контексту виконання між виконуваними задачами. Чим більше необхідно витратити процесорного часу та пам’яті для зміни контексту, тим більшою має бути задача, щоб виправдати затрати на паралелізм. Однією з ключових відмінностей реалізації потоку виконання програми Go є орієнтованість на обробку менших задач паралельно. Зменшивши ціну запуску однієї goroutine, архітектори мови надали інженерам більше простору для маневру при виборі підходів до обробки задач.

4.2.5 Синзронізація в Go

Спосіб взаємодії між процесами є вкрай важливим, адже самі по собі вони неефективні. Існує декілька способів комунікації:

– спільний доступ до пам’яті (мьютекси);

– не блокуючі алгоритми;

– обмін повідомленнями через канали.

З вище зазначених підходів саме обмін повідомленнями за допомогою каналів найкраще вписується у контекст систем масового обслуговування. Оскільки при цьому підході, на відміну від інших, акцент йде не на доступ до даних, а на взаємодію між компонентами [68]. В результаті від проблеми обробки даних ми переходимо до проблеми моделювання взаємодії та визначення станів компонентів системи, що добре описується математичним апаратом.

Хоча Go є молодою мовою та має запозичення ідей з інших, старших мов програмування, вона має і свої унікальні властивості, що роблять програми написані на Go ефективнішими та відмінними за характером від програм, написаних на споріднених мовах. Це єдина сучасна технологія, яка реалізує принципи взаємодіючого співвиконання, що описані у роботах C. A. R. Hoare “CSP” на рівні ядра [69]. Тому прямий переклад з таких мов, як, наприклад, Java або C++, малоймовірно дасть очікуваний результат. Відповідно, для ефективного використання мови програмування Go, важливо розуміти її особливості, ідіоми, а також практики, що дозволяють розкрити потенціал. У Go чітко сформульовані стандарти щодо форматування, найменування, будови програми, які дозволяють уніфікувати рішення, а також зробити їх більш зрозумілими для інших розробників і стійкими до змін. Також ця технологія перебуває у активній фазі розробки та вдосконалення, і до неї регулярно надходять оновлення, надаючи додаткові можливості та переваги.

Ядром програми є її потік виконання програми – набір процесів, що відповідальні за обробку та виконання алгоритму програми, а також забезпечення допоміжних операцій, що відбуваються за замовчанням на рівні мови [70].

Мова програмування Go перезавантажує підхід до паралельного виконання програм за рахунок інтеграції методології взаємодіючого співвиконання у філософію та інструментарій мови. Вона є новатором, оскільки

вперше серед популярних мов програмування саме у Go було реалізовано концепцію CSP на рівні базових механізмів.

В Go механізми синхронізації реалізовані через пакет sync, який забезпечує різні примітиви для управління доступом до спільних ресурсів і координації роботи goroutine [72]. Основна ідея полягає в тому, щоб створювати безпечне середовище для паралельного виконання, де goroutine можуть взаємодіяти без конфліктів і збереження даних відбувається послідовно.

Основні примітиви синхронізації у Go:

– Mutex (Mutual Exclusion): це базовий примітив, що використовується для захисту доступу до спільних ресурсів, sync.Mutex дозволяє заблокувати доступ до ресурсу для однієї goroutine, поки інші чекають на звільнення, а методи Lock() і Unlock() забезпечують взаємне виключення, запобігаючи одночасному доступу до ресурсу та уникаючи станів гонки;

– RWMutex: це розширений варіант Mutex, що дозволяє одночасне читання кільком goroutine, але виключає будь-які операції, коли ресурс блокується для запису, що є ефективним способом підвищити продуктивність, коли операції читання переважають над записами;

– WaitGroup: використовується для синхронізації goroutine, які виконують паралельні завдання, за допомогою sync.WaitGroup можна очікувати завершення всіх goroutine перед тим, як продовжити виконання основної goroutine, при цьому методи Add(), Done() і Wait() допомагають керувати кількістю активних goroutine і блокувати виконання, доки всі вони не завершаться;

– Cond (Condition Variable): цей примітив допомагає керувати взаємодією goroutine на основі певних умов, sync.Cond використовується для вирішення класичних задач синхронізації, таких як проблема продюсер-споживач, він дозволяє goroutine очікувати певних подій, наприклад, коли ресурс стає доступним, і сигналізувати про зміни за допомогою методів Signal() і Broadcast();

– Atomic Operations: пакет sync/atomic забезпечує низькорівневі операції для безпечного читання та модифікації спільних змінних без використання замків, що дозволяє реалізовувати синхронізацію з мінімальними накладними витратами, що особливо корисно для простих інкрементів або перевірки значень, але ці операції потрібно використовувати з обережністю, оскільки вони можуть ускладнювати читабельність і розуміння коду.

Go пропонує широкий вибір інструментів для синхронізації, але загальна рекомендація – вибирати найбільш виразний і простий спосіб для конкретної задачі. У багатьох випадках використовувати канали для передачі повідомлень між goroutine може бути зручніше, ніж застосування замків, оскільки канали дозволяють природно впорядковувати доступ до даних.

4.2.6 Канали в Go

Головною особливістю Go є канали, які використовуються для синхронізації та комунікації між goroutine. У Go канали забезпечують засіб комунікації та синхронізації між goroutine, дозволяючи їм обмінюватися даними без використання явних замків або змінних стану. Канали функціонують як черги FIFO (First In, First Out) і передають копії об’єктів, а не самі об’єкти. Ця властивість дозволяє уникати конфліктів при паралельному доступі до пам’яті.

Типи каналів:

– буферизовані канали (асинхронні канали): такі канали мають буфер фіксованого розміру, що дозволяє відправнику надсилати дані, навіть якщо приймач ще не готовий отримувати їх, що є зручним, коли goroutine працюють із різною швидкістю, наприклад, коли відправник надсилає дані пакетами, а приймач обробляє їх у своєму темпі, при цьому важливо правильно вибрати розмір буфера, бо занадто великий буфер може споживати більше пам’яті, тоді як занадто малий не забезпечить оптимальної продуктивності;

– небуферизовані канали (синхронні канали): ці канали використовуються

для точної синхронізації між гору тинами, вони гарантують, що дані, надіслані відправником, одразу отримуються приймачем, при цьому важливо, щоб і відправник, і приймач були готові одночасно, інакше операція буде блокуватися, доки обидві сторони не синхронізуються, такі канали ідеальні для ситуацій, коли потрібна миттєва передача даних і синхронізація без затримок;

– сигнальні канали: використовуються для асинхронного зв'язку та координації, де необхідно лише сигналізувати подію або стан, наприклад, канал із нульовим розміром може використовуватись для передачі сигналу між goroutine без блокування.

Властивості каналів:

– блокування: канали за замовчуванням блокують операції відправки та прийому, поки обидві сторони не будуть готові, що дозволяє автоматично синхронізувати goroutine;

– очікування: якщо goroutine намагається отримати дані, але відправник ще не надає їх, канал буде утримувати goroutine в стані очікування, і навпаки;

– копіювання даних: канали передають копії об'єктів, що запобігає зміні стану об'єкта кількома goroutine одночасно.

Реалізація каналів в Go може бути уявлена як проста черга, захищена блокуванням мьютексів. По суті, Go використовує схожий підхід, і це втілено у структурі hchan (рис. 4.5). Ця структура відіграє центральну роль у реалізації каналів. Реалізація каналів у Go зосереджена на трьох структурах: hchan, waitq і sudog [72]. Зараз ми зосередимося на hchan, а інші розглянемо пізніше. Структура hchan є основою функціонування каналів. Вона розроблена для обробки різних аспектів поведінки каналів, забезпечуючи надійну комунікацію між goroutine. Ми розглянемо роль кожного поля структури в деталях пізніше. Ось як виглядає структура hchan (рис. 4.5). Окрім полів sendq та recvq, усі інші поля є самодостатніми та зрозумілими. Поки що відкладемо ці поля та розглянемо їх детальніше пізніше, коли буде зрозуміло їхнє використання.

```
type hchan struct {
 qcount   uint           // Total data in the queue
 dataqsiz uint           // Buffer size of the channel
 buf      unsafe.Pointer // Pointer to an array of data elements
 elemsize uint16         // Size of each element. Decide by type of elements
 closed   uint32         // Flag indicating whether the channel is closed
 elemtype *_type         // Type of the elements sent on the channel
 sendx    uint           // Index of the next slot to send data
 recvx    uint           // Index of the next slot to receive data
 recvq    waitq          // Queue of waiting receivers
 sendq    waitq          // Queue of waiting senders
 lock     mutex          // Mutex for protecting the channel
}
```

Рисунок 4.5 – Структура даних каналу

При створенні каналу Go виділяє структуру hchan у купі пам’яті та повертає вказівник на неї. Таким чином, канал – це просто вказівник на змінну типу hchan.

Як пояснено раніше, при ініціалізації буферизованого каналу створюється буфер із зазначеною довжиною каналу, і він чекає на додавання (enqueue) та видалення (dequeue) елементів. Наприклад, якщо відбувається enqueue, елементи розміщуються у буфері, а dequeue витягує елемент з буфера.

Далі на рис. 4.6 наведено процес наповнення буфера каналу за рахунок додавання послідовно відповідної кількості елементів. Оскільки подальші надсилання будуть блокувати виконання goroutine, проведемо операцію читання з каналу, для звільнення місця в буфері.

Важливий момент полягає в тому, що операція додавання (enqueue) включає копіювання пам’яті. Це копіювання завдання розміщується у буфері. Ключовим аспектом є те, що механізм копіювання в буфері забезпечує безпеку пам’яті. Єдина пам’ять, яку спільно використовують обидві goroutine, – це структура hchan, захищена м’ютексом. Все інше, як-от самі завдання, обробляється шляхом копіювання пам’яті.

«Не комунікуйте, розділяючи пам’ять; замість цього діліться пам’яттю через комунікацію» [63].

Тепер розглянемо ситуацію з небуферизованим каналом або буферизованим каналом, що досяг свого ліміту.

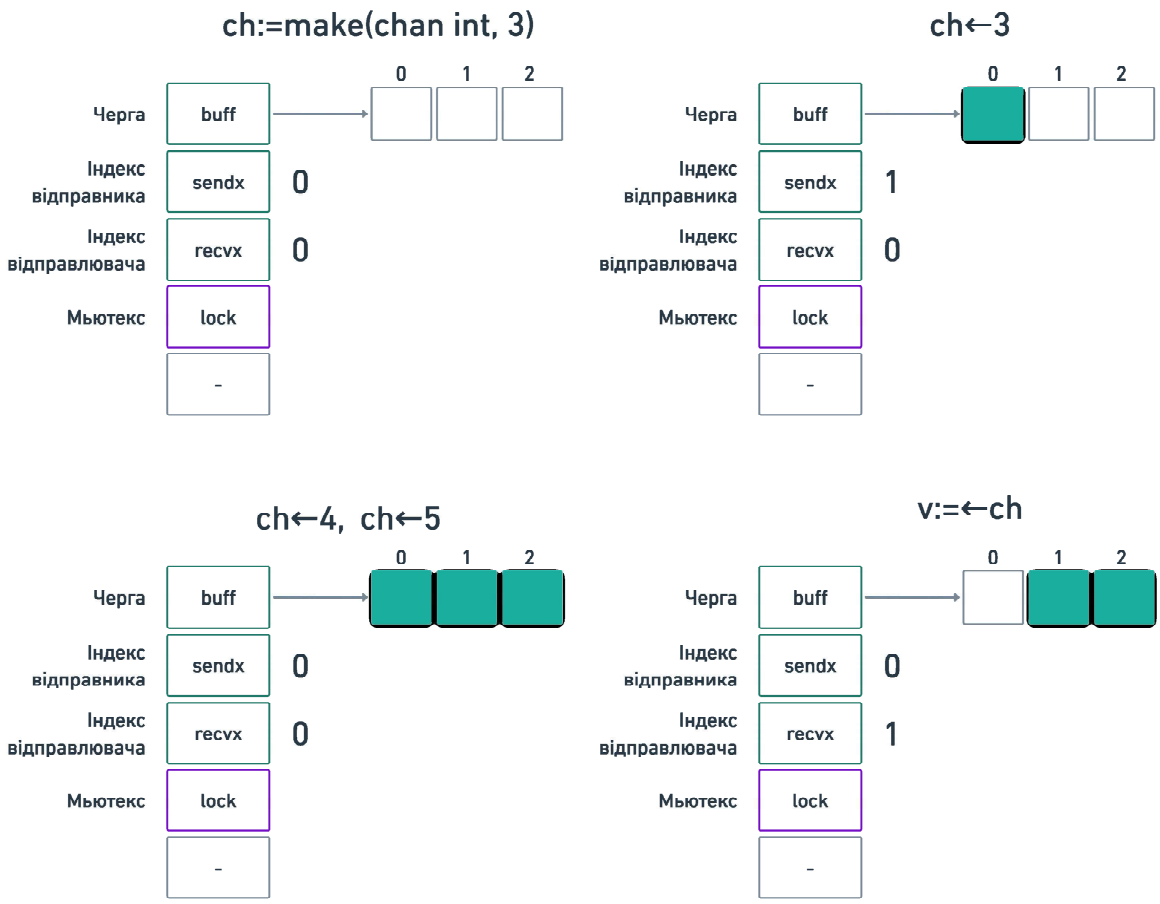

Рисунок 4.6 – Схема роботи структури даних буферизованого каналу

Уявімо, що goroutine G2 витрачає багато часу на обробку одного завдання, через що вона не може прийняти більше завдань з асинхронного каналу. У той же час, goroutine G1 продовжує надсилати більше завдань у канал без блокування.

Однак, якщо ми маємо буферизований канал, що досяг своєї місткості, G1 не зможе надсилати більше завдань у нього. Як наслідок, виконання G1 зупиняється або блокується, і воно відновиться тільки після того, як відбудеться операція прийому [70].

Ми розуміємо, що goroutine блокуються, коли або відправник, або приймач не готові, або черга заповнена. Тепер постає питання, як канал знає про наявність goroutine відправки та прийому. Тут використовуються поля recvq та sendq структури hchan. Поле sendq використовується для зберігання заблокованих goroutine, які намагаються надіслати дані в канал. Поле recvq використовується для зберігання заблокованих goroutine, які намагаються

прочитати дані з каналу.

Канали в Go є унікальним та потужним механізмом для організації паралельного виконання, що відтворює ідеї, закладені у парадигму Communicating Sequential Processes (CSP) [58], яку розробив C. A. R. Hoare. CSP наголошує на тому, що комунікація між паралельними процесами відбувається не через спільну пам’ять, а через обмін повідомленнями, і Go реалізує цю концепцію з витонченою простотою.

Унікальність та технічна перевага каналів у Go:

– інтуїтивний спосіб синхронізації та комунікації: канали Go забезпечують природний спосіб синхронізації між goroutine, вони дозволяють обмін даними та сигналізацію без використання явних замків або змінних стану, а взаємодія через канали робить паралельні програми чистими та легкими для розуміння, зберігаючи при цьому безпеку паралельного виконання;

– відтворення парадигми CSP: основна ідея CSP полягає у комунікації через повідомлення, а не через спільну пам’ять, канали в Go дозволяють чітко розділяти goroutine, де кожна з них може безпечно взаємодіяти через канали, забезпечуючи відсутність гонок і конфліктів, що відповідає концепції CSP, де паралельні процеси «спілкуються» один з одним через передання даних, а не безпосередньо звертаються до спільних ресурсів;

– асинхронна і синхронна комунікація: у Go є можливість використовувати як буферизовані (асинхронні), так і небуферизовані (синхронні) канали, що надає розробникам гнучкість у виборі підходу залежно від потреб їх програми: чи потрібна точна синхронізація, чи асинхронна комунікація з буфером для підвищення ефективності.

Канали в Go не лише реалізують принципи CSP, але й виводять їх на новий рівень завдяки вбудованій підтримці простоти та ефективності. Вони дозволяють передавати дані та сигнали між goroutine так, що код залишається чистим і передбачуваним. Це забезпечує ідеальний баланс між зручністю для розробника та продуктивністю, що робить Go популярним вибором для сучасних паралельних систем.

Таким чином, канали в Go є не просто засобом комунікації між паралельними процесами; вони є сутністю, що втілює найкращі принципи CSP, роблячи паралельне програмування безпечним, зрозумілим та ефективним.

4.3 Архітектура програмного забезпечення

4.3.1 Обрання типу архітектури

Першим значним роздоріжжям на шляху до побудови архітектурного рішення є відповідь на запитання про тип програмного забезпечення. Існує кілька підходів до того, як буде викликатись та виконуватись застосунок [76]. Умовно їх можна оцінити за наступними критеріями:

– чи має програмне забезпечення свою незалежну область пам’яті та планувальник виконання;

– чи є застосунок доступним неперервно, чи він запускається за вимогою на час обробки;

– яким саме чином забезпечується взаємодія з функціоналом і передача параметрів.

За даними ознаками можна поділити підходи на кілька груп, що наведені у табл. 4.2.

Таблиця 4.2 – Класифікація видів програмного забезпечення

	Окремий потік виконання програми	Доступність	Пряма взаємодія
Утиліта	+	-	-
Серверлес	+	-	-
Сервер (HTTP/RPC)	+	+	-
Пакет	-	+	+

Командна утиліта має свій потік виконання програми, але при цьому її необхідно запускати кожен раз, коли з’являється необхідність взаємодії, що погіршує її доступність. Крім того, з недоліків маємо ускладнену комунікацію через передачу параметрів та результатів у вигляді параметрів командного рядку.

Серверлес – це технологія, що пропонується представниками великих хмарних провайдерів. По суті є гібридом командної утиліти та серверу. За такого підходу відповідальність за запуск програми та комунікацію беруть на себе відповідні сервіси провайдера. З суттєвих недоліків такого підходу – велика кількість посередників, слабко типізовані контракти та нестабільність часу очікування через те, що функції, які певний час не викликаються, переводяться у гібернацію. І наступний раз необхідно чекати на їхній перезапуск. Проблему контактів можна вирішити із застосуванням додаткових сервісів, але це в значній мірі ускладнює систему [77].

Наступною групою є сервери. Вони працюють неперервно й незалежно від клієнтського застосунку, що спрощує раптову взаємодію та робить час відповіді прогнозованим. Однак, і в цьому випадку для використання у контрактах ми обмежені типами даних, що можуть приймати текстовий формат. У випадку з функцією такий підхід вимагатиме перетворення тексту запиту на програмний код та підключення його до виконання сервером. Такі дії значно збільшують складність та додають вразливостей системі.

У мові програмування Go найменший рівень абстракції, а також інкапсуляції забезпечується пакетом [63]. При цьому пакети є частиною потоку виконання програми загального застосунку та на них поширюються всі можливості по використанню системи типів мови програмування Go. Отже, ми можемо визначити тип параметра як функцію та застосувати дженерік типи для абстрагування від фактичних значень зі збереженням жорсткої типізації.

Кожен з перелічених підходів має свої переваги та недоліки. Однак, враховуючі вимогу щодо гнучкості функції, яка описуватиме логіку обслуговування, перед нами постає необхідність визначати саму функцію на

боці клієнтського коду й передавати до функціоналу, що обслуговуватиме процес моделювання. Це можна реалізувати за допомогою кодогенерації. Однак за такого підходу ми значно ускладнюємо реалізацію. Отже, наша стратегія полягатиме в тому, щоб реалізувати найпростіший і найбільш природній для мови програмування Go варіант із забезпеченням ізоляції відповідальності. Це дозволить відокремити спосіб комунікації в окремі абстракції та зробити можливим перевикористання однієї логіки за потреби у багато різних способів. Ми приймаємо рішення про розробку архітектури програмного продукту як пакету з делегацією відповідальності за спосіб комунікації рівню взаємодії.

4.3.2 Дизайн архітектури

Для початку необхідно визначитись із загальними компонентами, що приймають участь у побудові алгоритму [78], а саме у наступному.

Приймач заявок – по суті є публічним API пакетом. Задача полягає в прийомі вимог, загальній валідації та постанові до черги. У випадку переповнення черги – заявці буде відмовлено в обслуговуванні.

Черга очікування – приймає та утримує вимогу до звільнення каналу обслуговування. Надає перевагу заявкам, що надійшли раніше.

Канал обслуговування – безпосередньо обробник заявок. За раз може опрацьовувати не більше однієї вимоги. Перед початком виконання перевіряє чи не вичерпано час перебування заявки у системі. У випадку виникнення помилки при роботі або перевищення часу обслуговування має переривати виконання. Одразу після звільнення переходять у режим очікування чергової заявки.

Пост-обробник результатів – відокремлює вдалі заявки від таких, якім було відмовлено в обслуговуванні. У подальшому реєструє причини невиконання, а також може виконати інші допоміжні задачі такі, як ведення обліку подій.

Потік результатів – повертає користувачу API результат обробки заявок з

додаванням додаткової інформації щодо причин відмови на випадок помилки під час обслуговування.

Такий розподіл компонентів повністю накладається на теорію масового обслуговування і забезпечує вичерпну функціональність щодо моделювання СМО [43]. Загальну схему компонентів та послідовність обробки заявки зображено на рис. 4.7.

Наступним етапом необхідно підготувати цикл діаграм, що описуватимуть майбутню імплементацію з інженерної точки зору. Для цього скористаємось моделлю C4 – простим, прозорим і легким у використанні інструментом для розробників та архітекторів. Він покликаний стандартизувати поетапне проєктування складних систем, де ми починаємо занурення із загального вигляду системи та наближаємось з кожним наступним етапом ближче до реалізації. Така модель включає в себе всього 4 кроки.

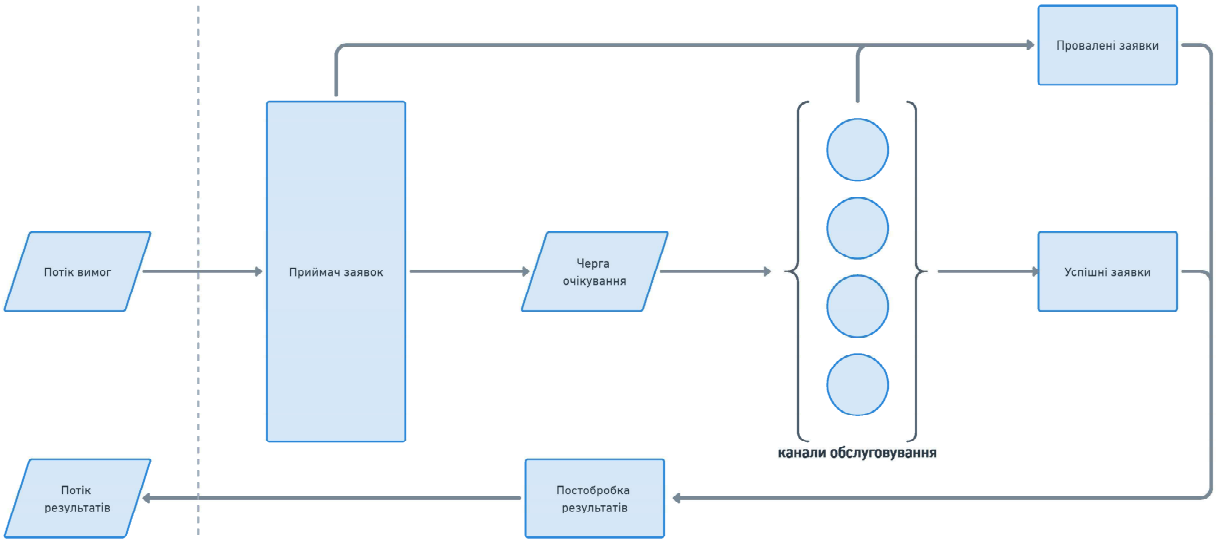

Рисунок 4.7 – Загальна схема алгоритму

Найвищій рівень абстракції – це C1. На ньому відображається контекстна діаграма із загальним зображенням системи. На рис. 4.8 можемо спостерігати дві сутності: користувача та кінцевий програмний продукт, що містить експеримент. Пакет, архітектуру якого ми закладаємо в рамках дослідження, на даному етапі не має значення оскільки він інтегрований всередину експерименту.

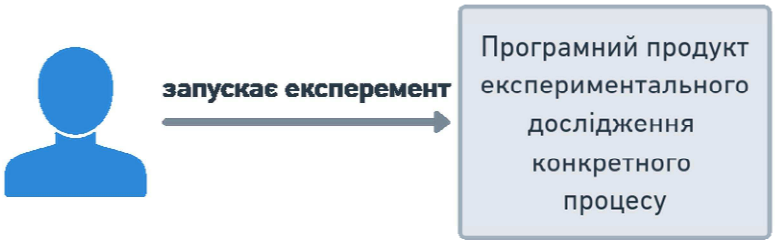

Рисунок 4.8 – Рівень моделі С1

На наступному рівні, також відомому як контейнерний, нас вже цікавить загальна будова системи без надмірних подробиць, але на рівні, необхідному для розуміння взаємодії глобальних складових. Дану діаграму зображено на рис. 4.9.

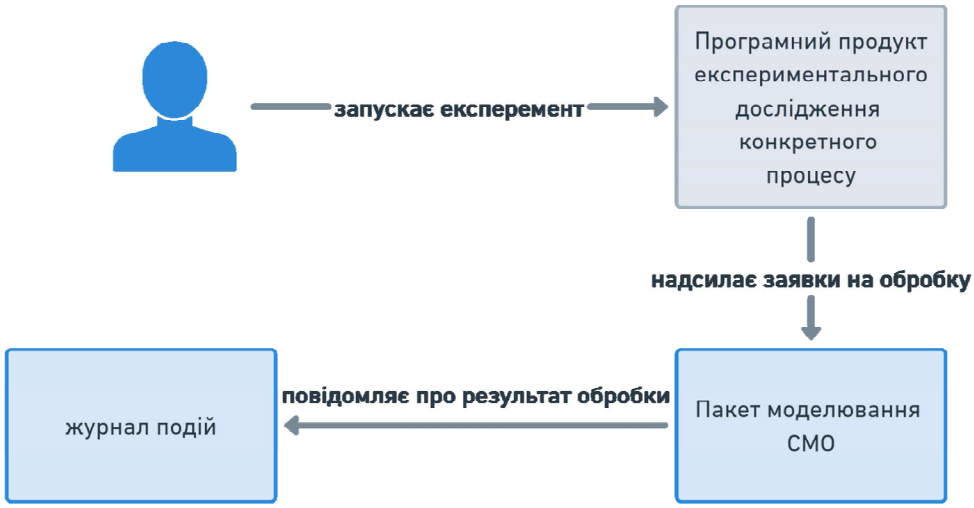

Рисунок 4.9 – Рівень моделі С2

Нарешті, час перейти до компонентної діаграми. Її роль в описі системи полягає в зображенні структурної будови досліджуваних компонентів. Нас найбільш цікавить саме пакет, архітектуру якого ми і будуємо. З рис. 4.10 випливає, що ключову роль системи відіграватиме канал обслуговування. Саме в цьому вузлі буде відбуватися обробка заявки, а також всі додаткові умови щодо часу перебування. Роль оркестрації та застосування взаємодіючого співвиконання має опрацьовувати окремий пакет [71]. Відокремлення реалізації паралелізму дозволить перевикористати даний функціонал для інших потреб, а також обмежити доступ користувачам.

Наступним рівнем абстракції буде діаграма коду, на ній будуть зображені подробиці реалізації специфічні для конкретної мови програмування Go. На рівні С4 найбільш цікавими для нас є дві зони відповідальності: симуляція СМО та організація паралельних обчислень шляхом застосування підходу взаємодіючого співвиконання [58].

Основна ідея полягає в чіткому розподілені обов'язків і побудова абстракцій на місцях перетину. Відповідно до визначених вимог будуватимемо два окремих пакети, кожен з яких буде надавати публічне API, і при цьому ховати всі деталі реалізації. При цьому сама заявка має бути обгорнута опціями, які будуть додавати додаткові властивості процесу без прямого в нього втручання. Дана діаграма наведена на рис. 4.11.

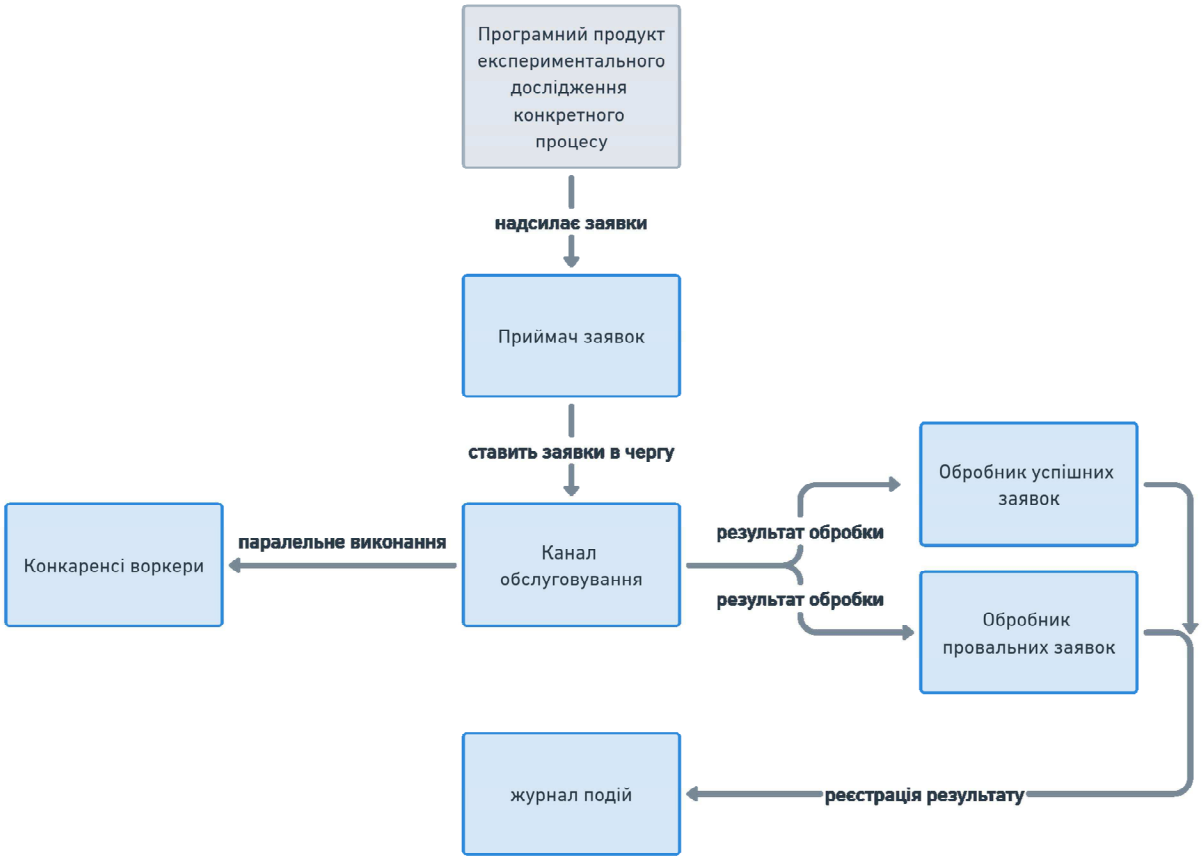

Рисунок 4.10 – Рівень моделі С3

У мові програмування Go асинхронна комунікація може бути забезпечена за рахунок вбудованого типу «канал» [63]. Ця структура даних надає можливість організувати безпечний доступ до заявок, які очікують у буфері й

будуть поступати одержувачам у послідовності, що співпадає з надходженням до черги.

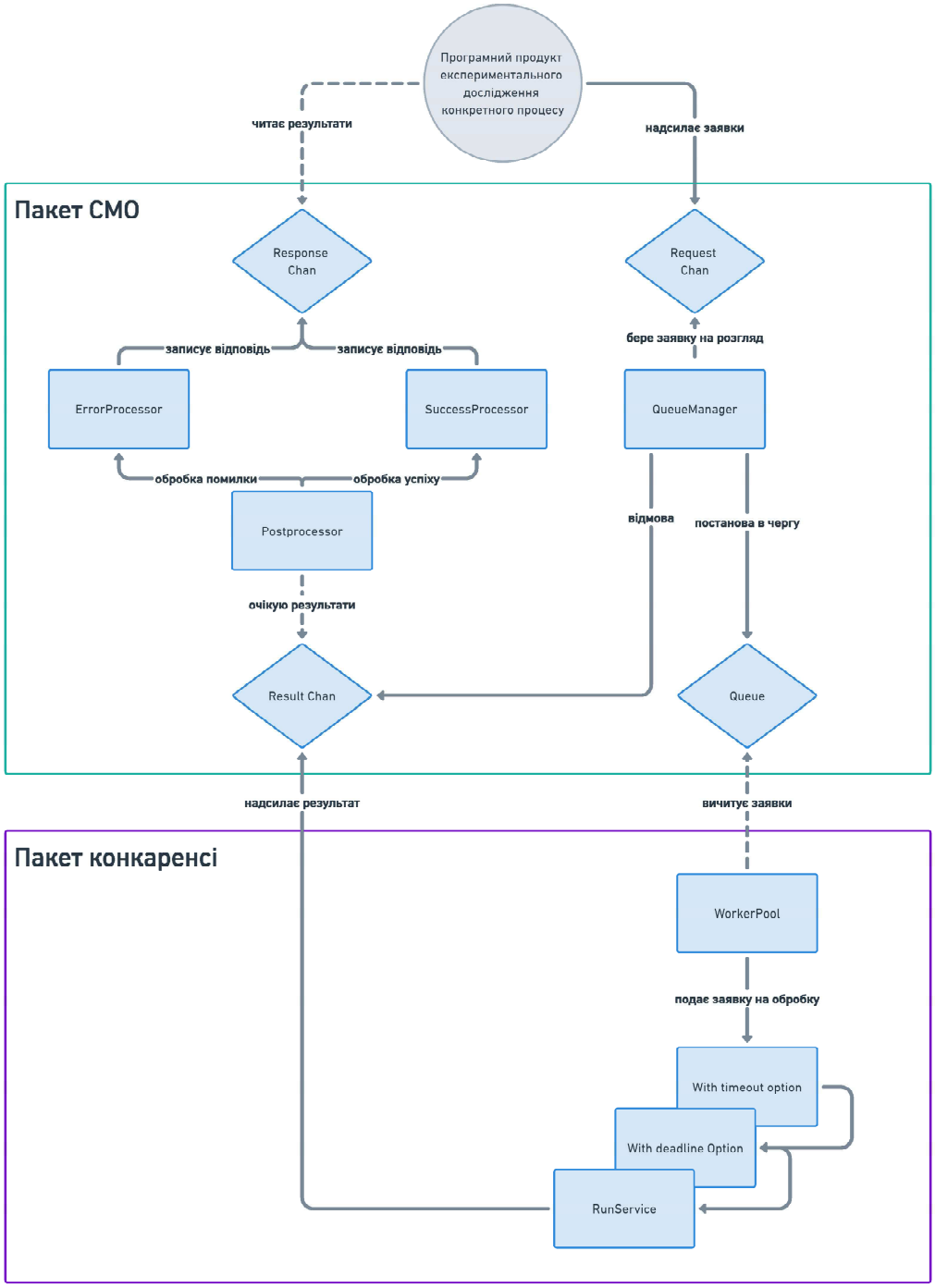

Рисунок 4.11 – Рівень моделі C4

Оскільки запис у переповнений канал у мові Go неможливий, це дає нам зручну обставину для відмови. Максимальний час перебування в системі може бути обмежений із застосуванням контексту з тайм-аутом. Після вичерпання

часового ліміту буде подано сигнал, який буде доступний в горутині, що обслуговує заявку, і буде надавати право переривання обробки. Кількість каналів обслуговування для системи буде визначатись кількістю горутин, які задіяні в пулі обробки. Самі по собі канали СМО можуть бути реалізовані за допомогою патерну паралельного програмування «worker» [71], реалізованого за допомогою легких програмних потоків – горутин.

Такий підхід чудово гармонує з іншим дуетом підходів fan-in та fan-out [71, 72]. Вони описують взаємодію між паралельними процесами та організацію асинхронної комунікації за умов різної кількості споживачів та генераторів повідомлень. Задля відокремленої та гнучкої пост-обробки ми додаємо кілька додаткових елементів, які будуть тимчасово розділяти успішні та невиконані заявки. І після здійснення необхідних кроків передавати у канал відповіді заявнику. Отже, дана система може бути реалізована мовою програмування Go і в повній мірі задовільнити вимоги, сформовані у постановці задачі.

Гарною практикою при проєктуванні API пакетів у мові програмування Go – є зменшення кількості способів взаємодії, а також максимальне їх спрощення [71]. Метою такого підходу є унеможливлення хибного використання функціоналу та скорочення часу необхідного для засвоєння можливостей пакету. У нашому випадку кількість функцій, що доступні до використання клієнтам, може бути доведено до всього лиш однієї. Це буде операція по ініціалізації СМО із заданими параметрами й передачею каналу, що буде використовуватись для подачі заявок. У відповідь дана функція (конструктор) поверне змінну типу канал для читання результатів, і вся подальша взаємодія буде відбуватись за допомогою двох каналів. Як результат, замість кооперації через надання спільного доступу до певних ділянок пам’яті кільком паралельним процесам, ми перебудовуємо механізм на взаємодію через спілкування повідомленнями. Такий підхід було введено і детально описано в роботі C. A. R. Hoare “Communicating Sequential Processes” [58], або скорочено CSP. Для завершення роботи зі СМО буде достатньо закриття каналу для надходження заявок, що призведе до лагідного згортання всіх процесів. Як і

було зазначено у вимогах, всі характеристики системи мають задаватись через структуру конфігурації перед початком роботи та бути використані при ініціалізації процесу.

4.4 Імплементація програмного забезпечення

Одним з ключових підходів у архітектурі програмного забезпечення є правило залишатись на рівні абстракцій на стільки довго, на скільки це можливо. Головна ідея полягає у тому, що вносити зміни на рівні узагальнень та концепцій значно дешевше, ніж коли вже готова імплементація. Також розповсюдженою проблемою є прийняття передчасних рішень. Справа в тому, що чим далі розвивається проєкт, тим більше стає розуміння про його фактичний спосіб використання. Відповідно, кожну ітерацію життєвого циклу програмного забезпечення відбувається переосмислення та уточнення потреб. Задля зменшення кількості змін внесених на півшляху, існує рекомендація відтермінувати низькорівневі рішення на стільки, на скільки це можливо. В даному розділі настав час перейти від загальної архітектури до конкретної реалізації.

4.4.1 Пакети та їх API

Як вже було зазначено у попередньому розділі, коли ми вибудовували архітектуру програмного забезпечення, основою рішення будуть саме пакети. На рис. 4.12 наведено структуру пакетів для реалізації програмного забезпечення.

На найвищому рівні пакети нашого проєкту поділяються на дві категорії:

– internal: пакет, що мітить бізнес-логіку клієнта, тобто тут буде зберігатись ієрархія пакетів, що складають експеримент;

– pkg: це «пакет з пакетами», призначений для зберігання інструментарію, що буде застосовано для моделювання СМО та реалізації взаємодіючого співвиконання, далі наша увага буде сфокусована саме на цьому

пакеті, оскільки реалізація конкретної СМО не є об’єктом даного дослідження.

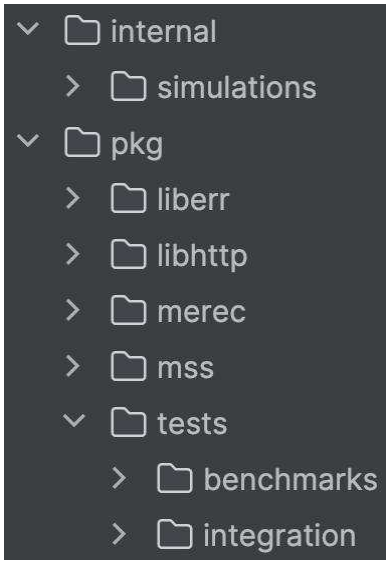

Рисунок 4.12 – Структура пакетів програмного рішення

Пакет pkg, у свою чергу, складається з наступних пакетів:

– liberr: призначений для спрощення та уніфікації обробки помилок;

– libhttp: має логіку призначену для спрощення HTTP маршрутизації;

– merec: пакет для автоматизації взаємодіючого співвиконання;

– mss: пакет для програмної симуляції систем масового обслуговування;

– tests: зберігає інтеграційні тести та бенчмарки, для замірів ефективності.

Оскільки liberr, libhttp та tests є поміжними пакетами, ми не будемо глибоко занурюватись у їхню реалізацію. Нас, передусім цікавлять пакети merec та mss. Самі по собі назви цих пакетів є частиною їх публічного API, оскільки вони будуть приймати участь у виклику експортованих сутностей. Саме тому вони мають задовольняти певні вимоги:

– назва має бути лаконічною;

– має бути префікс, якщо вона відноситься до категорії поміжних;

– назва не має перетинатися з пакетами стандартної бібліотеки;

– призначення пакету має бути зрозумілим з його назви.

Вище наведена структура пакетів задовольняє вимоги найменування.

Пакет mss, призначений для програмного моделювання систем масового

обслуговування має пласку структуру. Задля більшої гнучкості, в пакеті об’явлені наступні інтерфейси: sourcer, preprocessor, postprocessor (рис. 4.13). Звернемо увагу на те, що всі назви написані з малої літери, що у мові Go означає, що доступ до даних сутностей буде наданий тільки в рамках пакету mss.

Sourcer: мають імплементувати типи, що будуть слугувати джерелом даних для обслуговування. Реалізуючи єдиний метод feed, вони будуть надсилати до каналу дані до тих пір, поки вони є.

```
type sourcer[In any] interface {
    feed() <-chan In
}

type preprocessor[In, Out any] interface {
    inQueue(
        inCh <-chan In,
        postProcessCh chan<- merec.Result[Out],
    ) <-chan In
}

type postprocessor[Out any] interface {
    ok(Out)
    err(error)
}
```

Рисунок 4.13 – Перелік абстракцій

Preprocessor: це реалізація модуля сортування. Структура, що його реалізує, визначає політику потрапляння в загальну чергу імплементуючи метод inQueue.

Postprocessor: пропонує контракт для реалізації структурами, що визначатимуть як саме додатково мають бути опрацьовані результати обробки. Також пакет включає в себе кілька сутностей:

– System: займається безпосередньо моделюванням СМО, серед експортованого API, окрім самого типу System, має конструктор для нього та єдиний метод Run;

– ChanSourcer: призначений для надання вхідних даних з каналу, найбільш ймовірно, саме він буде застосовуватись у реальних симуляція;

– SliceSourcer: надає системі заявки які зберігаються у вигляді слайса;

– LimitedQueueRejectProcessor: симулює поведінку системи з обмеженою чергою та відмовами у випадку переповнення;

– LimitedSizedQueueRejectProcessor: симулює поведінку системи з обмеженою чергою та відмовами у випадку переповнення, із додатковим розбиттям на окремі класи пріоритетів за розміром заявки;

– LoggerPostprocessor – виконує логування результатів обслуговування заявок перед тим як повернути їх клієнту, може використовуватись для покращення прозорості системи, а також аналізу проблем.

Важливо зауважити, що даний перелік не є кінцевим, ми можемо додавати довільну кількість імплементацій для раніше описаних інтерфейсів, тим самим моделюючи різноманітні системи масового обслуговування. Фактично тип та характеристики системи, що моделюється визначаються під час створення System через відповідний конструктор. Саме там вказується який має бути sourcer, preprocessor та postproseccor.

Як бачимо, публічне API даного пакета навмисно обмежено для того, щоб задовільнити чергову вимогу до якості коду в Go. Ми надаємо доступ виключно до System, конструкторів реалізацій інтерфейсів та методу Run. Вся залишкова реалізація буде прихована від користувача, реалізуючи тим самим принцип інкапсуляції.

Згідно з запроєктованою архітектурою, для розподілення обов’язків, вся логіка, необхідна для оркестрації взаємодіючого співвиконання, була винесена в окремий пакет. Клієнту доступні для використання декілька різних функцій (ранерів), але на даний момент у моделюванні систем масового обслуговування приймає участь тільки одна: RunWorkerPool. Є ще декілька допоміжних сутностей та функцій.

4.4.2 Особливості взаємодіючого співвиконання у Go

Більшість систем у комп’ютерній інженерії або є вкрай складними із самого початку, або стають такими з часом. Їхнє спрощення є складною

задачею, що споживає величезну кількість зусиль, чим пізніше – тим більше. Одним з ефективних способів запобігання ускладненню систем – є декомпозиція складних задач на низки простих, маленьких операцій, що взаємодіють між собою. У той же час таке розбиття дозволяє більш ефективно будувати математичні моделі систем, оскільки замість однієї надскладної задачі, ми будемо моделювати маленькі значно простіші процеси.

Одним із ключових етапів у формуванні мови Go є реалізація ідеї взаємодіючого співвиконання [67]. За цим терміном ховається низка особливостей потоку виконання програми, а саме:

– представлення процесу у вигляді великої кількості незалежних маленьких програмних потоків (goroutine);

– взаємодія між процесами відбувається через обмін повідомленнями;

– заблоковані задачі стають на очікування й виконання переходить до інших процесів у черзі;

– здатність легко збільшити кількість обробників для простих задач.

Кожна з вищезгаданих властивостей має своє відображення у теорії масового обслуговування. Тож розглянемо всі відмінності підходу та їхній вплив на моделювання систем масового обслуговування.

В компіляторі та потоці виконання програми мови Go реалізована величезна кількість механізмів, що стимулюють та спрямовують інженерів до розбиття великих складних процесів на послідовності маленьких незалежних задач. Такими чинниками є, наприклад, малий початковий розмір goroutine, дешевизна переключення контексту, зручні та безпечні механізми керування goroutine.

Ініціалізація програми виконується в одній goroutine, але ця goroutine може створювати інші goroutine, які працюють паралельно. Оператор go, який запускає нову goroutine, синхронізується перед початком виконання цієї goroutine. Завершення goroutine не гарантується як синхронізоване перед будь-якою подією в програмі. Якщо впливи goroutine повинні бути спостережені іншою goroutine, необхідно використовувати механізм синхронізації, такий як

блокування або комунікація через канал, щоб встановити відносний порядок виконання.

Комунікація через канали є основним методом синхронізації між goroutine. Кожна операція надсилання в певний канал співвідноситься з відповідною операцією отримання з цього каналу, зазвичай у різній goroutine. Надсилання в канал синхронізується перед завершенням відповідної операції отримання з цього каналу. Закриття каналу синхронізується перед операцією отримання, яка повертає нульове значення через те, що канал закритий. Операція отримання з небуферизованого каналу синхронізується перед завершенням відповідної операції надсилання в цей канал. Цю особливість будемо застосовувати в процесі згортання роботи програми. Закриття каналу постачання заявок буде означати, що весь механізм симуляції СМО має бути поетапно завершено.

Це правило узагальнює попереднє правило для буферизованих каналів. Воно дозволяє моделювати семафор з підрахунком за допомогою буферизованого каналу: кількість елементів у каналі відповідає кількості активних використань, ємність каналу відповідає максимальній кількості одночасних використань, надсилання елемента захоплює семафор, а отримання елемента звільняє семафор. Це поширений підхід для обмеження паралелізму.

Завдяки особливостям потоку виконання програми в Go ми можемо збільшувати кількість каналів без необхідності наново реалізовувати код програми. Єдине, що обмежує нас від лінійного збільшення продуктивності роботи програми – це фізична кількість потоків операційної системи. Нарощування кількості goroutine, що займаються обслуговуванням заявок, даватиме приріст часу до певного моменту. Після двократного перевищення кількістю серверів числа потоків ОС буде спостерігатися плато продуктивності. При застосуванні надзвичайно великої кількості goroutine програма почне працювати повільніше, ніж раніше. Це пояснюється накладними розходами на переключення контексту виконання. В мові Go вартість однієї goroutine є відносно малою, але не безкоштовною. Цю різницю можна прослідкувати при

використанні інструменту profiler. Даний функціонал дозволяє збирати подробиці про роботу потоку виконання програми, для подальшої візуалізації та аналізу. Profiler дає вичерпну інформацію щодо того, яка горутина на якому потоці ОС виконувалась, як довго, наскільки рівномірно розподіллено навантаження між потоками, як часто відбувається збірка сміття та як багато ресурсів відіймає.

На рис. 4.14 зображено як поводить система за незначної кількості горутин.

На рис. 4.15 зображена поведінка потоку виконання програми, коли горутин стає надзвичайно багато.

Рисунок 4.14 – Розподілення роботи для невеликої кількості goroutine

Для даного дослідження це означає, що наближення поведінки симуляції до роботи реальної системи буде відбуватись орієнтовно до моменту співпадіння кількості каналів обслуговування з кількістю потоків операційної системи. Далі певний час комп’ютерна модель буде продовжувати забезпечувати необхідний рівень паралельності.

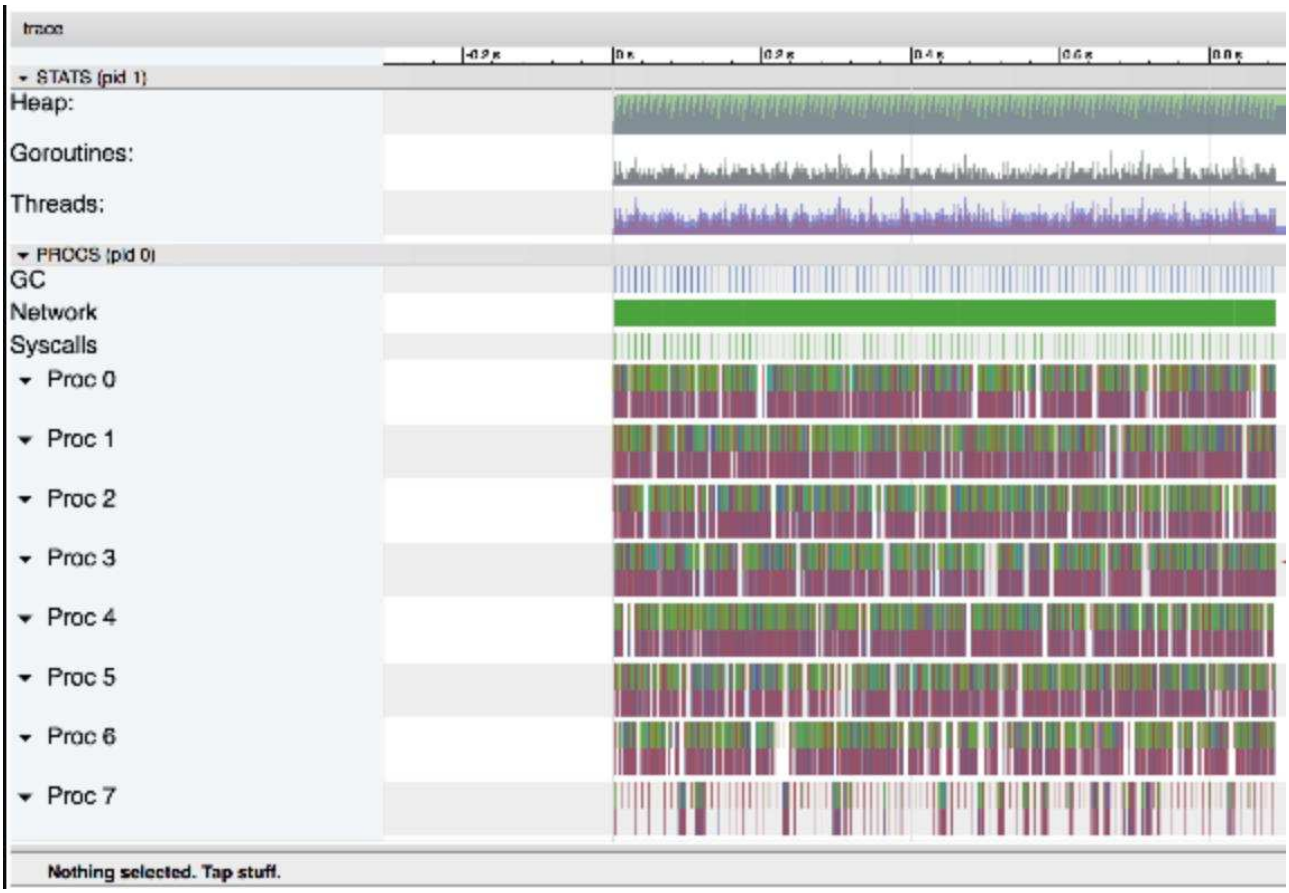

Рисунок 4.15 – Розподілення роботи за великої кількості goroutine

Однак орієнтовно з моменту перевищення кількості потоків у два рази буде спостерігатись відрив від реального процесу. Дана особливість в певний момент буде присутня в програмних реалізація будь-якими мовами програмування, оскільки вона бере свій початок від обмеженості комп’ютерного устаткування та не може бути розв’язана програмно. Задля визначення оптимальної кількості програмних потоків можна застосувати навчену модель штучного інтелекту, що буде враховувати поточне та прогнозоване навантаження, та на основі досвіду надавати рекомендації.

4.4.3 Патерни взаємодіючого співвиконання

Патерни проєктування – це узагальнені рішення для типових проблем, які виникають під час розробки програмного забезпечення. Вони є перевіреними методами побудови архітектури та організації коду, які дозволяють зробити

програму більш гнучкою, масштабованою та легшою у підтримці. Кожен патерн є описом певної архітектурної або структурної проблеми та пропонує спосіб її вирішення, що сприяє ефективному використанню ресурсів і полегшує процес розробки. Патерни проєктування в цілому є універсальними для різних мов програмування, але інколи вимагають адаптації під особливості мови.

Патерни взаємодіючого співвиконання – це специфічні патерни, які спрямовані на вирішення проблем, що виникають при розробці багатозадачних та паралельних систем [71]. Вони допомагають організувати взаємодію між асинхронними потоками та процесами, забезпечуючи безпеку даних і спільний доступ до ресурсів. Патерни взаємодіючого співвиконання зосереджуються на управлінні станом і черговістю виконання задач, а також на мінімізації блокувань, що можуть призводити до помилок або зниження продуктивності. Приклади таких патернів включають семафори, монітори, черги завдань та блокування з обмеженим доступом.

Роль та важливість патернів взаємодіючого співвиконання полягає у забезпеченні можливості ефективно використовувати обчислювальні ресурси та зменшені складності управління потоками, що особливо важливо для високонавантажених та критичних додатків таких як: бази даних, сервери, хмарні сервіси та ін [71]. Патерни взаємодіючого співвиконання не лише підвищують продуктивність, але й зменшують ризик виникнення помилок у багатопотокових середовищах, що дозволяє розробникам створювати більш надійне та стійке до збоїв програмне забезпечення.

Далі розглянемо патерни, реалізація яких має найбільші особливості в мові програмування Go. Вони відіграють важливу роль в побудові ефективної взаємодіючого співвиконання в застосунку.

Розглянемо патерн Generator (генератор). Головна ідея полягає в тому що функція перед запуском goroutine створює канал та повертає його назовні. Даний канал буде використовуватись для синхронізації та обміну повідомленнями. Цей патерн має два підтипи: Generator-sender (відправник) (рис. 4.16) та Generator-receiver (отримувач) (рис. 4.17). Даний підхід дозволяє

реалізувати практику, за якої goroutine, що надсилає дані, має закривати канал. В іншому випадку є ймовірність отримати стан паніки через спробу надіслати дані до закритого каналу.

```
func fabricsOut() <-chan int {
    ch := make(chan int)

    go func() {
       for i := 0; i < 3; i++ {
          ch <- i
       }

       close(ch)
    }()

    return ch
}
```

Рисунок 4.16 – Реалізація патерна Generator-sender

При цьому, цікавим є особливість Generator-receiver, який приймає примітив синхронізації. Такий крок зумовлений необхідністю змусити клієнта чекати, до поки отримувач не завершить обробку даних.

```
func fabricsIn(wg *sync.WaitGroup) chan<- int {
    ch := make(chan int)

    wg.Add(1)
    go func() {
       for v := range ch {
          fmt.Printf("Recieved value %d\n", v)
       }

       wg.Done()
    }()

    return ch
}
```

Рисунок 4.17 – Реалізація патерна Generator-receiver

Розглянемо далі патерн Selector (селектор). Часто при синхронізації кількох процесів виникає потреба дочекатись першого. Так може бути, наприклад, якщо ми надсилаємо одні дані на обробку трьом різним сервісам. Після отримання першого результату інші нас більше не цікавлять. Розрізняють

дві політики: з очікуванням Synchronous (синхронний) (рис. 4.18) та з результатом за замовчанням Asynchronous (асинхронний) (рис. 4.19).

```
func syncSelector(ch1, ch2, ch3 <-chan int) int {
    select {
    case v := <-ch1:
       return v
    case v := <-ch2:
       return v
    case v := <-ch3:
       return v
    }
}
```

Рисунок 4.18 – Synchronous Selector

У випадку Synchronous Selector програма чекатиме, коли результат надійде хоча б до одного з каналів. Якщо одночасно повідомлення надійде до декількох каналів, результат буде обраний випадковим чином. Asynchronous Selector перевіряє чи є хоча б один такий канал, що має повідомлення, і у випадку якщо немає – візьме значення за замовчення, не чекаючи результатів.

```
func asyncSelector(ch1, ch2, ch3 <-chan int) int {
    select {
    case v := <-ch1:
       return v
    case v := <-ch2:
       return v
    case v := <-ch3:
       return v
    default:
       return 0
    }
}
```

Рисунок 4.19 – Asynchronous Selector

Розглянемо патерн Notificator (нотифікатор). Синхронізації незалежних процесів приділено велику увагу в інших розділах дослідження. Однак в життєвому циклі програми є особливий період – згортання. Коли система складна та широко застосовує підходи взаємодіючого співвиконання для коректного завершення програми необхідно подати сигнал всім goroutine.

Такий сигнал може бути реалізований щонайменше двома способами. Перший підхід застосовує особливість поведінки закритого каналу. При його закритті всі goroutine, які намагаються читати з каналу, неперервно отримують значення за замовченням для типу повідомлення (рис. 4.20).

```
func chanNotificator() {
    ch := make(chan struct{})
    wg := sync.WaitGroup{}

    wg.Add(3)
    for i := 0; i < 3; i++ {
       go func(i int) {
          <-ch
          fmt.Printf("Goroutine %d unlocked\n", i)
          wg.Done()
       }(i)
    }

    close(ch)
    wg.Wait()
}
```

Рисунок 4.20 – Notificator з каналом

З цікавих особливостей можна виділити використання пустої структури в якості типу повідомлень. Завдяки цьому вдається уникнути виділення пам’яті для роботи Notificator.

Другий підхід є певною мірою надбудовою навколо попереднього. Його суть полягає у використанні контексту з відміною. Оскільки контекст з відмовою працює за допомогою каналів, то і реалізація буде дуже схожою (рис. 4.21).

Є кілька відмінностей на практиці між двома підходами. Перевагою використання каналу є простота застосування базового типу. Однак при спробі повторно закрити канал виникне паніка (позаштатна помилка). Перевагою контексту є можливість багаторазово викликати функцію переривання. Повторні виклики ніяк не вплинуть на поведінку програми, але і не викличуть паніки.

```
func contextNotificator() {
    ctx, cancel := context.WithCancel(context.Background())
    wg := sync.WaitGroup{}

    wg.Add(3)
    for i := 0; i < 3; i++ {
       go func(i int) {
          <-ctx.Done()
          fmt.Printf("Goroutine %d unlocked\n", i)
          wg.Done()
       }(i)
    }

    cancel()
    wg.Wait()
}
```

Рисунок 4.21 – Notificator з контекстом

Наступним розглянемо патерн Pipeline (пайплайн). Чисто виникає потреба збудувати алгоритм з чіткою послідовністю дій, але слабкою залежністю між етапами. Яскравим прикладом слугує конвеєр. При реалізації даного патерну в Go скористаємось можливістю передавати канали до функції в якості параметрів. Тоді ми можемо утворити ланцюг з каналів, поєднавши тим самим всі етапи обробки повідомлення (рис. 4.22).

```
func pipeline() {
    leftmost := make(chan int)
    right, left := leftmost, leftmost

    for i := 0; i < 3; i++ {
       right = make(chan int)

       go func(left, right chan int) {
          val := <-right
          fmt.Printf("Link %d processed\n", val)
          left <- val + 1
       }(left, right)

       left = right
    }

    go func(c chan int) { c <- 0 }(right)

    <-leftmost
}
```

Рисунок 4.22 – Pipeline

Це може бути корисно, коли ми хочемо динамічно додавати логіку обробки так, щоб не треба було вносити зміни в код.

Далі розглянемо патерн Fan Out. Розповсюдженою проблемою при проєктуванні програмного забезпечення є потреба розподілити загальний потік повідомлень на категорії за якимись ознаками. В прикладі наведеному на рис. 4.23 зображено розбиття вхідного потоку повідомлень. Кожне значення, надіслане до каналу, є цілим числом. Реалізація патерну Fan Out створює колекцію вихідних каналів. Після чого розподіляє всі числа, що кратні трьом до другого каналу, кратні двом – до першого, та всі інші будуть надсилатись до нульового каналу (за замовченням). В даному прикладі застосовано ще одну особливість каналів у мові Go, а саме – можливість створити буфер. Завдяки цьому горутина, що намагається надіслати дані до каналу не буде блокуватися до моменту переповнення буфера, що у свою чергу робить алгоритм більш гнучким.

```
func fanOut(in <-chan int) []chan int {
    var out []chan int

    for i := 0; i < 3; i++ {
       out = append(out, make(chan int, 1))
    }

    go func() {
       for v := range in {
          switch {
          case v%3 == 0:
             out[2] <- v
          case v%2 == 0:
             out[1] <- v
          default:
             out[0] <- v
          }
       }

       for _, ch := range out {
          close(ch)
       }
    }()

    return out
}
```

Рисунок 4.23 – Застосування Fan Out для розбиття вхідних повідомлень

Тепер розглянемо патерн Fan In. Логічним наступним кроком після розбиття потоку повідомлень буде поєднання даних з кількох джерел в один канал. Відповідно на вхід ми отримуємо колекцію каналів, а на вихід повертаємо один канал (рис. 4.24).

Важливо, що такий патерн можна застосувати тільки в тому випадку, коли всі вхідні канали оперують з повідомленнями однакового типу даних. Додаткову складність додає необхідність відслідковувати, коли всі вхідні канали будуть закриті для безпечного закриття вихідного каналу. Це необхідно для запобігання паніки при спробі писати в закритий канал. Важливо пам’ятати, що навантаження на результуючий канал буде дорівнювати сумарному навантаженню на всі вхідні канали. Відповідно задля безперебійної роботи системи бажано передбачити буфер, що дозволить працювати незалежним процесам без зайвих блокувань.

```
func fanIn(ins []chan int) chan int {
    var wg sync.WaitGroup
    out := make(chan int, len(ins))

    wg.Add(len(ins))
    for i, in := range ins {
       go func(i int, in <-chan int) {
          for val := range in {
             out <- val
             fmt.Printf("Value from the chan #%d = %d was
merged\n", i, val)
          }
          wg.Done()
       }(i, in)
    }

    go func() {
       wg.Wait()
       close(out)
    }()

    return out
}
```

Рисунок 4.24 – Застосування Fan In для поєднання вхідних повідомлень

Розглянемо патерн Workers (воркери). Одним з найбільш популярних патернів взаємодіючого співвиконання є принцип паралельної обробки. Як

показують дослідження, створення необмеженої кількості goroutine, що займаються опрацюванням даних, може призвести до збільшення накладних витрат потоку виконання програми на переключення контексту. Фактично це означає, що goroutine будуть сперечатись за ресурси операційної системи. Для більш оптимального керування потоками при розподілені паралельних обчислень застосовується патерн Workers (рис. 4.25). Його ідея полягає в додаванні абстракції серверу, який буде займатись контролем кількості goroutine, що займаються обслуговуванням вхідного потоку повідомлень та розподіленням роботи між ними. Певне ускладнення реалізації викликано необхідністю відслідкувати момент, коли всі воркери завершили обробку даних для закриття каналу вихідних повідомлень, що у свою чергу запустить ланцюгову реакцію та проінформує всіх читачів з каналу результатів про те, що обробка завершена та більше повідомлень очікувати не варто.

```
func workers(in <-chan int) chan int {
    const n = 3
    outCh := make(chan int, n)
    var wg sync.WaitGroup

    wg.Add(n)

    for i := 0; i < n; i++ {
       go func(i int) {
          for val := range in {
             outCh <- val
             fmt.Printf("worker %d processed value %d\n", i, val)
          }
          wg.Done()
       }(i)
    }

    go func() {
       wg.Wait()
       close(outCh)
    }()

    return outCh
}
```

Рисунок 4.25 – Паралельна обробка із застосуванням патерна Workers

4.4.4 Взаємодіючого співвиконання та комунікація

Однією з ключових ідей дослідження є впровадження методології CSP до програмного моделювання систем масового обслуговування. Найважливішими складовими компонентами CSP є взаємодіюче співвиконання та комунікація. Тобто задача полягає у тому, щоб відобразити симуляцію СМО в якості набору атомарних, незалежних процесів та поєднати їх мережею взаємодії. На рис. 4.26 наведено схему реалізації для симулювання системи масового обслуговування з обмеженою чергою та відмовами із застосуванням пріоритезації за розміром задачі, а також перенесено на вигляд, що відповідає структурі CSP. На схемі прямокутні блоки відповідають незалежним процесам, що виконуються в окремих goroutine. А стрілки – це канали, через які goroutine обмінюються даними. Напрямок стрілки вказує від процесу, який надсилає дані, до процесу, який читає дані з каналу. Таким чином, можна прослідкувати повний цикл обслуговування заявки заданою системою.

При реалізації алгоритму оптимізації багатоканальних систем масового обслуговування із застосуванням розбиття вхідного потоку було застосовано всі попередньо описані патерни проєктування взаємодіючого співвиконання. Завдяки цьому вдалося досягнути високого рівня організованості та потенціалу до горизонтально масштабування.

Розберемо покроково з поясненням всі етапи:

– обробка заявки розпочинається з того, що клієнт (Client) надсилає дані заявки у канал, що лежить в основі імплементації Sourcer, а Sourcer в даній схемі є конкретною реалізацією інтерфейсу, суть якого полягає у трансформації вхідного потоку даних в такий, що є зручним для обробки в рамках симуляції;

– другим етапом є надсилання заявки на сортувальник (Sorter), на цьому етапі буде оцінено розмір вимоги та розподілено її до відповідної черги за класом. З цього моменту послідовність надходження буде зберігатись тільки в рамках одного класу розмірів, дана компонента реалізує патерн паралельного проєктування, також відомий, як Fan Out;

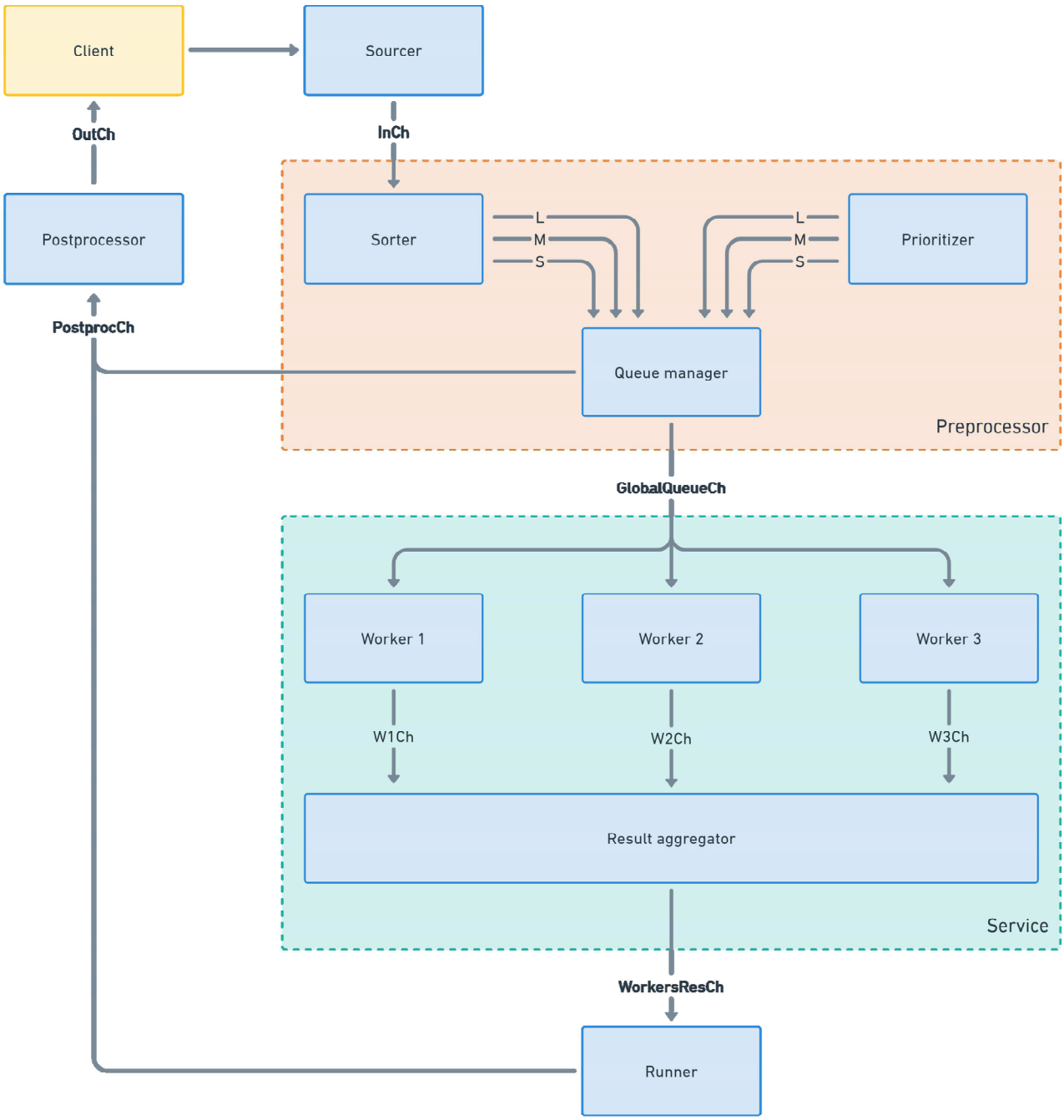

Рисунок 4.26 – Інформаційна модель організації процесів та комунікації

– Prioritizer – це goroutine, відповідальність якої полягає у наданні дозволів на постановку в головну чергу для відповідних розмірів, ця частина програми займається балансуванням пріоритетів і вона має надавати шанс виконатись всім розмірам задач, але у пропорції, що буде збільшувати продуктивність системи, окрім того, модуль пріоритезації має слідкувати за наповненістю черг розмірів і надавати шанс тим класам, черги яких переповнюються;

– з каналів за класами заявки зчитує менеджер черги (Queue manager), але зчитування буде заблоковано допоки від іншої компоненти (модуля пріоритезації) не надійде дозвіл і таким чином в наступний канал, глобальну чергу (GlobalQueueCh) будуть надсилатись задачі згідно з пріоритетами, а у випадку, якщо загальна черга переповнена – заявці буде відмовлено в обробці, про що буде надіслано результат до каналу постобробки (PostprocCh), при цьому всю комбінацію Sorter, Prioritizer та Queue manager можна замінити на альтернативну імплементацію інтерфейсу Preprocessor, чим досягається модульність та гнучкість рішення;

– на наступному кроці в гру вступає пакет merec, вся обробка блоку Service прихована інкапсуляцією, варіативність реалізації досягається за рахунок вибору функції, яка буде займатись паралельною обробкою, параметром кількості каналів обслуговування (Worker) та додатковими обгортками, при чому Обгортки (Option) можна додавати за необхідністю, наприклад, для того щоб обмежити максимальний час обробки заявки;

– з каналу глобальної черги (GlobalQueueCh) читає одночасно декілька серверів обслуговування (Workers), кількість таких воркерів визначається на початку і відповідає кількості каналів обслуговування СМО, що моделюється, має бути щонайменше один сервер обслуговування, також неможливо зробити нескінченну кількість воркерів, кожен з каналів обслуговування виконує однакову логіку, а саме функцію, яка була надана при ініціалізації системи масового обслуговування, по завершенню обробки буде сформовано результат, який у свою чергу може бути успіхом або невиконанням, і даний результат надсилається у канал результатів привласнений конкретним воркером;

– оскільки кожен сервер обслуговування пише результати у власний канал, нам необхідно мати компоненту, яка буде займатись збором всіх результатів до загального потоку (result aggregator), вона читає з усіх каналів воркерів та пише поєднані результати до загального каналу результатів (WorkerResCh), дана компонента реалізує паттерн паралельного проектування, також відомий як Fan In, також важливо врахувати кількість працюючих

воркерів про розрахунках розміру буфера для каналу WorkerResCh, щоб уникнути блокування через його переповнення;

– з загального каналу результатів читає основний оркестратор системи (Runner), він пересилає отримані повідомлення до каналу додаткової обробки (PostprocCh), також в цій частині програми відбувається ініціалізація та загальне керування усім процесом обробки заявок;

– нарешті, останнім кроком з відповідного каналу вичитує результати горутина додаткової обробки (Postprocessor), дана компонента є модульною, і може бути замінена на довільну імплементацію інтерфейса, за замовченням ми проводимо логування до консолі усіх успішних та невиконаних заявок, після чого готові результати надсилаються до клієнтського каналу результатів (OutCh), клієнт пакету mss зможе отримати готовий результат, вичитавши його з каналу.

4.4.5 Обробка помилок

Помилки при обробці заявок є очікуваним сценарієм. В Go підхід до обробки помилок базується на тому, що помилки є значеннями. Це означає, що замість використання механізмів виключень, як у деяких інших мовах програмування, Go обробляє помилки як звичайні значення, які повертаються з функцій. Такий підхід дає розробникам контроль над тим, як і де відбувається обробка помилок, забезпечуючи чітку і передбачувану поведінку програм.

У Go функції можуть повертати кілька значень, включаючи одне значення помилки (error). Коли функція виявляє помилку, вона повертає значення помилки разом із результатом виконання (або nil, якщо помилки немає). Це дозволяє виклику функції одразу перевіряти, чи виникла помилка, і відповідно реагувати.

Починаючи з версії Go 1.13, додана підтримка обгортання помилок за допомогою fmt.Errorf з %w для створення ланцюга помилок. Це дозволяє додавати контекст до помилки та виводити пов’язані помилки за допомогою

функцій errors.Is та errors.As для порівняння та визначення типу помилки. Такий підхід спрощує діагностику та забезпечує більш гнучке керування помилками в коді.

Пропонується підкреслити декілька розповсюджених практик:

– Go заохочує явну перевірку помилок там, де вони виникають, що забезпечує більшу передбачуваність, що відрізняється від механізмів виключень, які можуть бути прихованими, що ускладнює розуміння потоку виконання програми;

– використовуючи обгортання помилок, важливо розуміти, коли варто надавати додатковий контекст, а коли краще уникати цього, щоб не розкривати деталей реалізації, наприклад, якщо помилка пов’язана з базою даних, важливо, чи слід відкривати помилку типу sql.ErrNoRows зовнішньому виклику, адже це може порушити абстракцію;

– іноді корисно визначати спеціалізовані типи помилок з додатковими методами або полями, які дозволяють розробникам гнучкіше обробляти певні ситуації (наприклад, помилки тимчасового характеру для мережевих операцій).

В рамках реалізації пакетів для моделювання СМО було розроблено рішення, що задовольняє наші потреби з точки зору бізнес-логіки, а також, яке відповідає вимогам в мові Go. Помилку було вбудовано в сам результат обслуговування (рис. 2.27). При цьому доступу до поля з помилкою у клієнта не буде. Натомість, користувач має використовувати відповідний метод Err для перевірки на наявність помилки обробки заявки. І тільки у випадку, якщо помилки немає, запитувати у змінної типу Result значення. Для цього передбачений метод Value. Якщо спробувати отримати значення для випадку, коли результат містить помилку, користувач отримає значення за замовченням.

При цьому в користувача залишається можливість співставляти помилку з шаблонами наданими пакетами merec та mss. Це дозволить зрозуміти, в чому саме полягала проблема та класифікувати помилку для подальшої комунікації з користувачем.

```go
// Result is the Call execution result. If the error happened, the
value should be empty.
type Result[Out any] struct {
    value Out
    err   error
}

// String implements io.Stringer interface.
func (r Result[Out]) String() string {
    return fmt.Sprintf("value: %v, err: %v", r.value, r.err)
}

// Value returns the execution result value.
func (r Result[Out]) Value() Out {
    return r.value
}

// Err returns the execution error.
func (r Result[Out]) Err() error {
    return r.err
}

// ValueResult creates a new success result with a set value.
func ValueResult[Out any](value Out) Result[Out] {
    return Result[Out]{value: value}
}

// ErrorResult creates a new failure result with a set error.
func ErrorResult[Out any](err error) Result[Out] {
    return Result[Out]{err: err}
}
```

Рисунок 4.27 – Реалізація динамічних результатів

4.4.6 Налаштування та варіативність

Всі налаштування майбутньої системи визначаються безпосередньо у процесі її створення:

– кількість каналів обслуговування;

– розмір черги;

– спосіб надання заявок;

– політика постанови до черги;

– постобробка результатів.

Кожен з цих параметрів має варіативність та можливість розширювати

кількість різновидів систем за рахунок імплементації відповідних інтерфейсів. Такий підхід надає високий рівень гнучкості та можливість проводити експерименти з використанням нових відходів до організації черг. За необхідності, можна замінити модуль пріоритезації та повністю делегувати навченій нейронній мережі процес прийняття рішення про обрання задачі для подальшої постанови в загальну чергу. Водночас, інженерам, які додають нові процесори немає необхідності з нуля реалізовувати складний механізм комунікації та синхронізації. Відкритий код та прості абстракції спонукають до подальшого розвитку програмного продукту призначеного для моделювання СМО.

Важливим напрямком у гнучкості є використання дженерік типів. Справа в тому, що Go, як жорстко типізована мова, вимагає описати всі типи перед тим як скомпілювати бінарний файл. Але на момент проєктування гнучкої платформи для моделювання СМО ми не можемо знати, які саме типи будуть використовуватись в якості вхідних та вихідних даних. Все стає ще складнішим, коли мова заходить про визначення логіки обслуговування, адже єдиний тип, що описує поведінку, а не представлення даних у пам’яті, є функція. Рішення було знайдено у нещодавно доданому функціоналі динамічних типів. Завдяки ній логіка обслуговування може бути представлена в якості змінної наступного типу (рис. 4.28).

```
// Call is the function to be executed.
type Call[In, Out any] func(context.Context, In) (Out, error)
```

Рисунок 4.28 – Функція що описую логіку обслуговування

Дана форма дозволяє визначати клієнтом під час написання фактичної СМО типи даних для входу та виводу з системи, а також, описати логіку перетворення. При цьому динамічні типи у Go підтримують складні структури.

Код розробленої програмного забезпечення знаходиться у додатку В.

Висновки за розділом 4

1. Обрано мову програмування Go як таку, що найкращим чином передає

ідею взаємодіючого співвиконання в інструментах доступних розробникам. Розібрано особливості технології, а саме: кроскомпіляція, модель пам’яті, організація потоку виконання програми та синхронізації.

2. Розглянуто проблеми, що вирішуються архітектурою програмного забезпечення, її роль у подальшому житті проєкту, а також основні задачі. Було сформульовано вимоги до архітектури програмного забезпечення.

3. Було проведено аналіз існуючих типів архітектури програмного забезпечення та на підставі вимог, обрано такий, що найбільш задовольняє потреби при написанні застосунку мовою Go.

4. Розроблено дизайн архітектури, складено структурну, компонентну діаграми, а також схеми послідовної абстракції C4.

5. Розглянуто реалізацію програмного забезпечення мовою програмування Go із детальним розбором прийнятих технічних рішень. Додаткова увага приділена гнучкості рішення, необхідній для моделювання різноманітних систем. Оскільки реалізація автоматизації взаємодіючого співвиконання побудована на абстракціях та винесена в окремий пакет, її можна застосувати у будь-якій програмі написаній мовою Go. В такий спосіб програмне забезпечення, розроблене в рамках роботи, набуває практичної цінності для різних галузей науки, де є потреба в застосуванні паралельних або асинхронних обчислень.

5. Було розібрано механізми синхронізації у мові програмування Go, внутрішня будова, а також підходи до їх застосування. Особлива увага приділена каналам, оскільки вони грають одну з ключових ролей у реалізації підходів CSP в мові Go, роблячи її унікальною. Надано структуру незалежних процесів та послідовність комунікації, що виникає при моделюванні систем масового обслуговування.

Основні результати четвертого розділу опубліковані у роботах [79 – 81].

Список джерел, які використано у даному розділі, наведено у повному списку використаних джерел [43, 54, 58, 62, 63, 66 – 68, 70 – 78].

ВИСНОВКИ

Дисертаційна робота присвячена вирішенню науково-практичної задачі, що полягає в розробці комп'ютерної та інформаційної моделей аналізу процесів масового обслуговування й удосконалення методу та моделей оптимізації систем масового обслуговування з використанням технології пріоритезації вхідного потоку заявок. Найбільше важливі наукові та практичні результати роботи полягають у наступному.

1. Проведено аналіз сучасного стану задачі моделювання систем масового обслуговування. Виділено клас систем, що застосовують багатоканальність та обмежену чергу з відмовами, для подальшого вивчення через їхню практичну цінність. Зроблено висновок про необхідність удосконалення методів і моделей, спрямованих на зменшення ймовірності відмови в обслуговуванні за рахунок пріоритезації вхідного потоку заявок за їх розміром. Також було виявлено необхідність удосконалити метод пріоритезації в частині балансування пріоритетів для уникнення заборони виконання великих задач за великого навантаження на систему.

2. Розглянуто багатоканальну систему масового обслуговування з обмеженою чергою та відмовами, для якої вперше запропоновано інформаційну та комп'ютерну моделі аналізу із застосуванням взаємодіючого співвиконання (concurrency). Це дозволило подати досліджувану систему у вигляді набору простих незалежних процесів, які взаємодіють між собою шляхом обміну подіями, що в свою чергу збільшило потенціал до масштабування та ефективності застосування паралельних обчислень.

3. В процесі дослідження моделей оцінки складності заявки та балансування класів пріоритетів було виявлено недолік, пов'язаний з тим, що при застосуванні класичних підходів всі заявки попадають до однієї черги і це обмежує можливості окремого управління квотами на знаходження в черзі. Для подолання цього була розвинута модель розбиття загального вхідного потоку

заявок по категоріях за розміром із застосуванням поділу черги очікування за квотами для категорій заявок.

4. Метод оптимізації систем масового обслуговування з наданням пріоритету меншим задачам SJF (Shortest Job First) є ефективним для зменшення загальної ймовірності відмови, але призводить до втрати великих задач при значному навантаженні. Було удосконалено цей метод, завдяки поєднанню комбінованої системи пріоритетів, моделі оцінки складності задачі та моделі розбиття загальної черги на окремі під кожен з класів заявок.

5. На основі запропонованих інформаційної та комп'ютерної моделей розроблено програмне забезпечення мовою Go, що в повній мірі задовольняє моделям Communicating Sequential Processes (CSP) та взаємодіючого співвиконання (concurrency). Імплементація автоматизації взаємодіючого співвиконання побудована на абстракціях із застосування патернів проєктування concurrency та винесена в окремий пакет, що надає можливість застосовувати її у будь-якій програмі, що написана мовою Go. Отже, розроблене програмне забезпечення набуває практичної цінності для різних галузей науки та техніки, де є потреба в застосуванні паралельних або асинхронних обчислень.

6. Результати досліджень дисертаційної роботи впроваджені в освітній процес Харківського національного університету радіоелектроніки.

7. Отримані результати розширюють теоретичну та практичну основи для розв'язання математичними методами системного аналізу з застосуванням сучасних інформаційних технологій прикладних задач, що зводяться до навантажених систем масового обслуговування з рівнозначними каналами обслуговування.

8. Напрям подальших досліджень може бути пов'язаний з вдосконаленням модуля сортування задля забезпечення ефективнішого потрапляння в зони балансу для складних практичних задач з нерівномірним вхідним потоком заявок. Система балансування пріоритетів також може набути додаткового розвитку задля розробки балансних стратегій, наприклад,

досягнення режиму роботи, коли для задач більшого розміру ймовірність відмов залишається тою самою, що і без оптимізації, одночасно зі зменшенням загальної ймовірності відмови. Програмне забезпечення може отримувати подальшу підтримку у вигляді нових імплементацій для модуля пріоритетів, завдяки чому стати більш універсальним інструментом моделювання систем масового обслуговування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kleinrock L. Queueing Systems, Volume 1: Theory. New York: John Wiley & Sons, 1975. 448 p.

2. Kleinrock L., Gail R. Queueing Systems: Problems and Solutions. New York: John Wiley & Sons, 1996. 240 p.

3. Cooper R. B. Introduction to Queueing Theory. New York: Macmillan, 1972. 277 p.

4. Allen A. O. Probability, Statistics, and Queueing Theory: With Computer Science Applications. 2nd ed. Boston: Academic Press, 1990. 768 p.

5. Gross D., Shortle J. F., Thompson J. M., Harris C. M. Fundamentals of Queueing Theory, Solutions Manual. 4th ed. Hoboken: Wiley-Interscience, 2008. 88 p.

6. Kingman J. F. C. Poisson Processes. Vol. 3. Oxford Studies in Probability: Clarendon Press, 1993. 104 p.

7. Sommereder M. Modelling of Queueing Systems with Markov Chains: An Introduction to Basic and Advanced Modelling Techniques. Norderstedt: Books on Demand, 2011. 288 p.

8. Bolch G., Greiner S., de Meer H., Trivedi K. S. Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. 2nd ed. Hoboken: Wiley-Interscience, 2006. 896 p.

9. Rece L., Vlase S., Ciuiu D., Neculoiu G., Mocanu S., Modrea A. Queueing Theory-Based Mathematical Models Applied to Enterprise Organization and Industrial Production Optimization. *Mathematics*. 2022. Vol. 10, No. 14. Article 2520.

10. Matzka J. M. Discrete Time Analysis of Multi-Server Queueing Systems in Material Handling and Service. Karlsruhe: KIT Scientific Publishing, 2014. 104 p.

11. Chen Y., Dong J. Scheduling with Service-Time Information: The Power of Two Priority Classes. *Operations Research.* 2021. Vol. 69, No. 6. Pp. 1811–1827.

12. Tikhonenko O., Ziółkowski M. Queueing Systems Models and Their Applications. *Mathematics.* Special Issue: Computational and Applied Mathematics. 2024.

13. Lakatos L., Szeidl L., Telek M. Introduction to Queueing Systems with Telecommunication Applications. 2nd ed. Cham: Springer, 2019. 576 p.

14. Dshalalow J. H. Advances in Queueing Theory, Methods, and Open Problems. Boca Raton: CRC Press, 2023. 528 p.

15. Schrage L. E., Miller L. W. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*. 1966. Vol. 14, No. 4. Pp. 670–684.

16. Harchol-Balter M., Bansal N., Schroeder B., Agrawal M. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems.* 2003. Vol. 21, No. 2. Pp. 207–233.

17. Scully Z., Harchol-Balter M., Scheller-Wolf A. SOAP: One Clean Analysis of All Age-Based Scheduling Policies. *Proceedings of the ACM on Measurement and Modeling of Computer Systems*. 2018. Vol. 2, No. 1. Pp. 16–30.

18. Dell'Amico M., Carra D., Pastorelli M., Michiardi P. Revisiting Size-Based Scheduling with Estimated Job Sizes. *Proceedings of the 22nd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS).* 2014. Pp. 249–258.

19. Down D. G. Open Problem—Size-Based Scheduling with Estimation Errors. *Stochastic Systems*. 2019. Vol. 9, No. 3. Pp. 295–296.

20. Nuyens M., Wierman A. The Foreground–Background Queue: A Survey. *Performance Evaluation*. 2008. Vol. 65, No. 3–4. Pp. 286–307.

21. Aalto S., Ayesta U., Righter R. On the Gittins Index in the M/G/1 Queue. *Queueing Systems*. 2009. Vol. 63, No. 1–4. Pp. 437–458.

22. Scully Z., Grosof I., Harchol-Balter M. The Gittins Policy is Nearly Optimal in the M/G/k under Extremely General Conditions. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. 2020. Vol. 4, No. 3. Article 43.

23. Emadi S., Ibrahim R., Kesavan S. Can "Very Noisy" Information Go a

Long Way? An Exploratory Analysis of Personalized Scheduling in Service Systems. Working Paper. 2019.

24. Gromoll H. C. Diffusion Approximation for a Processor Sharing Queue in Heavy Traffic. *Annals of Applied Probability*. 2004. Vol. 14, No. 2. Pp. 555–611.

25. Gromoll H. C., Kruk Ł., Puha A. L. Diffusion Limits for Shortest Remaining Processing Time Queues. *Stochastic Systems*. 2011. Vol. 1, No. 1. Pp. 1–16.

26. Puha A. L. Diffusion Limits for Shortest Remaining Processing Time Queues under Nonstandard Spatial Scaling. *The Annals of Applied Probability*. 2015. Vol. 25, No. 6. Pp. 3381–3404.

27. Banerjee S., Budhiraja A., Puha A. L. Heavy Traffic Scaling Limits for Shortest Remaining Processing Time Queues with Heavy Tailed Processing Time Distributions. *The Annals of Applied Probability*. 2022. Vol. 32, No. 4. Pp. 2587–2651.

28. Lin M., Wierman A., Zwart B. Heavy-Traffic Analysis of Mean Response Time under Shortest Remaining Processing Time. *Performance Evaluation*. 2011. Vol. 68, No. 10. Pp. 955–966.

29. Cox D. R., Smith W. L. Queues. London: Methuen, 1961. 180 p.

30. Van Mieghem J. A. Dynamic Scheduling with Convex Delay Costs: The Generalized cμ-Rule. *Annals of Applied Probability*. 1995. Vol. 5, No. 3. Pp. 809–833.

31. Mandelbaum A., Stolyar A. L. Scheduling Flexible Servers with Convex Delay Costs: Heavy-Traffic Optimality of the Generalized cμ-Rule. *Operations Research*. 2004. Vol. 52, No. 6. Pp. 836–855.

32. Argon N. T., Ziya S. Priority Assignment Under Imperfect Information on Customer Type Identities. *Manufacturing & Service Operations Management*. 2009. Vol. 11, No. 4. Pp. 674–693.

33. Sun Z., Argon N. T., Ziya S. Patient Triage and Prioritization Under Austere Conditions. *Management Science*. 2018. Vol. 64, No. 10. Pp. 4471–4489.

34. Grosof I., Scully Z., Harchol-Balter M. SRPT for Multiserver Systems.

Performance Evaluation. 2018. Vol. 127–128. Pp. 154–175.

35. Harrison J. M., Zeevi A. Dynamic Scheduling of a Multiclass Queue in the Halfin–Whitt Heavy Traffic Regime. *Operations Research*. 2004. Vol. 52, No. 2. Pp. 243–257.

36. Kim J., Randhawa R. S., Ward A. R. Dynamic Scheduling in a Many-Server, Multiclass System: The Role of Customer Impatience in Large Systems. *Manufacturing & Service Operations Management*. 2018. Vol. 20, No. 2. Pp. 285–301.

37. Хинчин А. Я. Работы по математической теории массового обслуживания. Москва: ГИФМЛ, 1963. 235 с.

38. Джейсуол Н. Очереди с приоритетами. Москва: Мир, 1973. 279 с.

39. Kendall D. G. Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*. 1953. Vol. 24, No. 3. Pp. 338–354.

40. Литвинов А. Л. Вероятностный анализ буферной памяти вычислительного комплекса с групповым адаптивным выбором информации. *Автоматизированные системы управления:* тематический сборник. Харьков: ХАИ, 1984. Вып. 5. С. 82–87.

41. Новиков О. А., Петухов С. И. Прикладные вопросы теории массового обслуживания. Москва: Советское радио, 1969. 400 с.

42. Камке Э. Справочник по обыкновенным дифференциальным уравнениям. Санкт-Петербург: Лань, 2003. 576 с.

43. Литвинов А. Теорія систем масового обслуговування. Харків: ХНУМГ ім. О. М. Бекетова, 2018. 141 с.

44. Ложковський А. Теорія масового обслуговування в телекомунікаціях. Одеса: ОНАЗ ім. О. С. Попова, 2010. 112 с.

45. Alotaibi F. M., Ullah I., Ahmad S. Modeling and Performance Evaluation of Multi-Class Queuing System with QoS and Priority Constraints. *Electronics*. 2021. Vol. 10, No. 4. P. 500.

46. Beshley M., Kryvinska N., Beshley H., Yaremko O., Pyrih J. Virtual

Router Design and Modeling for Future Networks with QoS Guarantees. *Electronics*. 2021. Vol. 10, No. 10. P. 1139.

47. Malik S., Gupta K., Gupta D., Singh A., Ibrahim M., Ortega-Mansilla A., Goyal N., Hamam H. Intelligent Load-Balancing Framework for Fog-Enabled Communication in Healthcare. *Electronics*. 2022. Vol. 11, No. 4. P. 566.

48. Malik N., Sardaraz M., Tahir M., Shah B., Ali G., Moreira F. Energy-Efficient Load Balancing Algorithm for Workflow Scheduling in Cloud Data Centers Using Queuing and Thresholds. *Applied Sciences*. 2021. Vol. 11, No. 13. P. 5849.

49. Vercellino C., Scionti A., Varavallo G., Viviani P., Vitali G., Terzo O. A Machine Learning Approach for an HPC Use Case: The Jobs Queuing Time Prediction. *Future Generation Computer Systems*. 2023. Vol. 143. Pp. 215–230.

50. Apachidi X. N., Katsman Yu. Ya. Development of a Queuing System with Dynamic Priorities. *Key Engineering Materials*. 2016. Vol. 685. Pp. 934–938.

51. Reiman M. I. Some Diffusion Approximations with State Space Collapse. *Modelling and Performance Evaluation Methodology*. 1984. Vol. 60. Pp. 207–240.

52. Whitt W. Stochastic-Process Limits: An Introduction to Stochastic-Process Limits and Their Application to Queues. New York: Springer, 2002. 602 p.

53. Chikriy A., Gubarev V., Kondratenko Y., Turovyerova N. Multi-channel Queuing Systems with the Dynamic Priority. *Journal of Automation and Information Sciences*. 2009. Vol. 41, No. 8. Pp. 49–54.

54. Гнеденко Б. В., Коваленко И. Н. Введение в теорию массового обслуживания. Москва: Наука, 1987. 336 с.

55. Nielsen M., Winskel G. Models for Concurrency. Handbook of Logic in Computer Science. Vol. 4: Semantic Modelling. Oxford: Oxford University Press, 1995. Pp. 1–148.

56. Segala R. Probability and Nondeterminism in Operational Models of Concurrency. *CONCUR 2006 – Concurrency Theory*: Proceeding of 17th International Conference, Bonn, Germany, August 27-30, 2006. Berlin, Heidelberg: Springer, 2006. Pp. 64–78.

57. Reppy J. H. Concurrent Programming in ML. Cambridge: Cambridge

University Press, 1999. 328 p.

58. Hoare C. A. R. Communicating Sequential Processes. New Jersey: Prentice Hall, 1985. 235 p.

59. Sutter H., Larus J. Software and the Concurrency Revolution. *ACM Queue*. 2005. Vol. 3, No. 7. Pp. 54–62.

60. Chabbi M., Ramanathan M. K. A study of real-world data races in Golang. *PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022),* San Diego, CA, USA, June 13–17, 2022. New York: ACM, 2022. Pp. 474–489.

61. Liu Y., Gao R., Li Y., Fang D. EMsFEM based concurrent topology optimization method for hierarchical structure with multiple substructures. *Computer Methods in Applied Mechanics and Engineering*. 2024. Vol. 418, Part A. Article 116549.

62. Hoare C. A. R. Communicating Sequential Processes. *Communications of the ACM*. 1978. Vol. 21, No. 8. Pp. 666–677.

63. Donovan A. A. A., Kernighan B. W. The Go Programming Language. New York: Addison-Wesley Professional, 2015. 400 p.

64. Pontelli E., Gupta G. On the Duality Between Or-Parallelism and And-Parallelism in Logic Programming. *EURO-PAR '95 Parallel Processing*. Lecture Notes in Computer Science. Vol. 966. Berlin: Springer, 1995. Pp. 43–54.

65. Komendantskaya E., Schmidt M., Heras J. Exploiting Parallelism in Coalgebraic Logic Programming. *Electronic Notes in Theoretical Computer Science*. 2014. Vol. 303. Pp. 121–148.

66. Whitney J., Gifford C., Pantoja M. Distributed execution of communicating sequential process-style concurrency: Golang case study. *The Journal of Supercomputing*. 2019. Vol. 75, No. 3. Pp. 1396–1409.

67. Sufyan bin Uzayr. Mastering GoLang: A Beginner's Guide. Boca Raton: CRC Press, 2022. 298 p.

68. Fava D.S., Steffen M. Ready, set, Go!: Data-race detection and the Go language. *Science of Computer Programming*. 2020. Vol. 195. Article 102473. Pp. 1–

23.

69. Sottile M. J., Mattson T. G., Rasmussen C. E. Introduction to Concurrency in Programming Languages. New York: Chapman and Hall/CRC, 2009. 344 p.

70. Sufyan bin Uzayr. GoLang: The Ultimate Guide. Boca Raton: CRC Press, 2022. 366 p.

71. Castro Contreras M. Go Design Patterns. Birmingham: Packt, 2017. 402 p.

72. Cox-Buday K. Concurrency in Go: Tools and Techniques for Developers 1st Edition. Sebastopol: O'reilly Media, 2017. 236 p.

73. Conery R., Hanselman S., Haack P., Guthrie S. Microsoft Application Architecture Guide, 2nd Edition: Designing Applications on the .NET Platform. Microsoft Press, 2009. 560 p.

74. Кірхар Н. В. Застосування технології архітектурного проектування програмного забезпечення. *Проблеми інформатизації та управління*. 2019. Т. 1, № 61. С. 50–56.

75. Форкун Ю., Мартинюк В., Яшина О. Метод розробки та проектування архітектурної складової програмного застосунку. *Вимірювальна та обчислювальна техніка в технологічних процесах*. 2023. № 4. С.87–93.

76. Морозов А., Вакалюк Т., Кубрак Ю., Зосімович Д. Аналіз факторів впливу на архітектури програмних систем. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2022. № 1, С. 44–52.

77. Monolith vs SOA vs Microservices vs Serverless Architecture. 2019. [Електронний ресурс]. URL: https://rubygarage.org/blog/monolith-soa-microservices-serverless (дата звернення: 03.03.2024).

78. Лісняк А. О., Мильцев О. М., Мухін В. В., Чопорова О. В. Архітектура та проектування програмного забезпечення: метод. рек-ції до лаб. зан. для здобувачів ступ. вищ. осв. бакалавра спец. 121 «Інженерія програмного забезпечення» осв.-проф. прогр. «Програмна інженерія». Запоріжжя: ЗНУ, 2022. 53 с. URL: https://dspace.znu.edu.ua/xmlui/handle/12345/11642 (дата звернення: 01.04.2024).

79. Goldiner D., Tevyashev A. System Analysis of the Parallel Execution

Problem. *Інформаційні системи та технології – ICT-2019*: матеріали 8-ї Міжнародної науково-технічної конференції, Коблеве-Харків, 9-14 вересня 2019 р. Харків: ХНУРЕ, 2019. С. 210–213.

80. Гольдінер Д. І. Застосування мови програмування GO для моделювання процесів масового обслуговування. *Сучасний стан наукових досліджень та технологій в промисловості*. 2024. № 2(28). С. 65–75.

81. Гольдінер Д. І. Розробка архітектури програмного забезпечення для моделювання систем масового обслуговування під імплементацію мовою програмування GO. *Вісник Національного технічного університету «ХПІ»*. Серія: Системний аналіз, управління та інформаційні технології. 2024. № 1 (11). С. 85–90.

82. Goldiner D. Rejection probability reduction in queueing systems with limited queue using size-based prioritization. *Perspectives of Contemporary Science: Theory and Practice:* Proceedings of VII International Scientific and Practical Conference, Lviv, Ukraine, 19-21 August 2024. Lviv, Ukraine, 2024. Pp. 256–262.

83. Гольдінер Д. І., Матвієнко О. І. Зменшення ймовірності відмови в системах масового обслуговування з обмеженою чергою із застосуванням пріоритезації за розміром та штучного інтелекту. *Біоніка інтелекту*. 2024. № 1 (100). С. 36–42.

84. Безкоровайний В. В., Чоломбитько Д. В. Моделювання процесу багатокритеріального розподілу та виконання пакетів робіт під час проєктування технологічних систем. *Вісник Харківського національного автомобільно-дорожнього університету*. 2024. Т. 1. № 104. С. 7–14.

85. Bezkorovainyi V., Bezuhla H., Cholombytko D. Mathematical models of the cyclic work package distribution task. *Innovative integrated computer systems in strategic project management: Collective monograph*. European University Press. Riga: ISMA, 2022. P. 7–15. URL: https://mmp-conf.org/documents/archive/monography2022.pdf (дата звернення: 10.08.2024)

86. Beskorovainyi V., Kolesnyk L., Russkin V. Decision making support under conditions of incomplete consistency of expert advantages. *Innovative integrated*

computer systems in strategic project management: Collective monograph. European University Press. Riga: ISMA, 2022. P. 16–26. URL: https://mmp-conf.org/documents/archive/monography2022.pdf (дата звернення: 24.08.2024).

87. Bezkorovainyi V., Bezuhla H. Simulation modelling of the process of distribution and execution of work packages. *Information systems in project and program management: Collective monograph. ed. by I. Linde.* Riga: ISMA, European University Press, 2023. P. 16–28. URL: https://mmp-conf.org/documents/archive/monography2023.pdf (дата звернення: 30.09.2024).