МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

Бочкарьов О. Ю.

Паралельне програмування в ОС Linux

Навчальний посібник

Львів
Видавництво Львівської політехніки
2022

УДК 004.4
Б

Рецензенти:

Рак Т. Є., доктор технічних наук, доцент, професор кафедри інформаційних технологій, проректор з науково-педагогічної роботи ПЗВО “ІТ СТЕП Університет”;

Ткачук Р. Л., доктор технічних наук, професор, начальник кафедри управління інформаційною безпекою, Львівський державний університет безпеки життєдіяльності;

Глухов В. С., доктор технічних наук, професор, професор кафедри електронних обчислювальних машин, Національний університет “Львівська політехніка”

Рекомендувала Науково-методична рада
Національного університету “Львівська політехніка”
як навчальний посібник для студентів спеціальності 123 “Комп’ютерна інженерія”
(протокол № 65 від 20 жовтня 2022 р.)

Бочкарьов О. Ю.

Б Паралельне програмування в ОС Linux : навч. посібник / О. Ю. Бочкарьов. – Львів: Видавництво Львівської політехніки, 2022. – 161 с. Режим доступу: http://eom.lp.edu.ua/textbooks/np-pposlinux.pdf

ISBN xxx-xxx-xxx-xxx-x

В навчальному посібнику розглянуто основи паралельного програмування в ОС Linux на рівні обчислювальних процесів та на рівні програмних потоків. Основна увага приділена способам організації взаємодії паралельних процесів в ОС Linux та технологіям паралельного програмування.

Навчальний посібник призначений для студентів спеціальності 123 “Комп’ютерна інженерія” галузі знань 12 “Інформаційні технології”.

УДК 004.4

© Бочкарьов О. Ю., 2022
© Національний університет
“Львівська політехніка”, 2022

ЗМІСТ

Вступ

Використання паралельних комп'ютерних систем та паралельних програм дозволяє суттєво збільшити швидкість та ефективність розв'язування складних обчислювальних задач, в тому числі задач оптимізації, задач комп'ютерного моделювання в рамках наукових досліджень та промислового проектування, задач штучного інтелекту та машинного навчання, задач обробки великих наборів даних, задач біоінформатики та багатьох інших важливих задач. Відтак розробка і використання паралельних комп'ютерних систем та технологій паралельного програмування є надзвичайно актуальним питанням. Безкоштовна та відкрита операційна система Linux (ОС Linux) є основною ОС сучасних суперкомп'ютерів та високопродуктивних комп'ютерних систем. Крім цього ядро ОС Linux широко використовується для управління мобільними обчислювальними пристроями (смартфонами, планшетами, тощо) та вбудованими обчислювальними пристроями різного призначення, в тому числі в рамках концепції Інтернету речей (Internet of Things, IoT). Розробка та застосування технологій паралельного програмування, як на рівні системного, так і на рівні прикладного програмного забезпечення, є одним з головних напрямків розвитку ОС Linux. Переважна більшість прикладних програмних проектів в ОС Linux в тому, чи іншому вигляді використовує технології паралельного програмування задля максимального використання обчислювального ресурсу сучасних комп'ютерних систем та забезпечення високої продуктивності роботи.

В навчальному посібнику розглянуто основи паралельного програмування в ОС Linux на рівні обчислювальних процесів та на рівні програмних потоків. Оглянуто підходи до організації паралельних обчислень в сучасних паралельних та розподілених комп'ютерних системах. Розглянуто основні питання організації паралельних обчислень в ОС Linux, зокрема організацію обчислювальних процесів, способи запуску та породження нових паралельних процесів, запит та встановлення атрибутів процесів, а також завершення виконання процесів. Основна увага приділена способам організації взаємодії паралельних процесів в ОС Linux та технологіям паралельного програмування. Розглянуто всі основні способи взаємодії та синхронізації паралельних процесів в ОС Linux, в тому числі неіменовані канали (anonymous pipes), іменовані канали (named pipes), черги повідомлень (message queues), спільна пам'ять (shared memory), семафори (semaphores), сигнали (signals) та сокети Берклі (Berkeley sockets). Оглянуто програмний інтерфейс System V IPC для організації взаємодії паралельних процесів. Розглянуто питання організації паралельних обчислень в ОС Linux на рівні програмних потоків з використанням інтерфейсу прикладного програмування POSIX Threads (Pthreads), зокрема створення та ідентифікація програмних потоків, завершення їх виконання, очікування на завершення програмних потоків, управління атрибутами програмних потоків, а також синхронізація програмних потоків за допомогою м'ютексів, умовних змінних, блокування читання-запису (readers–writer lock), спін-блокування (spinlock) та бар'єрної синхронізації. Розглянуто технології паралельного програмування OpenMP та oneTBB. Наведено приклади їх практичного застосування. Розглянуто проблему паралельного програмування в розподілених системах з використанням програмного інтерфейсу MPI.

Метою навчального посібнику є виробити у студентів чітке та систематизоване уявлення про основні принципи паралельного програмування, надати студентам базові знання про технології паралельного програмування в ОС Linux, а також надати навички практичного застосування цих технологій для створення паралельних програм на рівні системного та прикладного програмного забезпечення. Використання навчального посібника допоможе студенту досягнути наступних результатів навчання: знати загальні принципи організації обчислювальних процесів в паралельних комп'ютерних системах різних класів; розуміти концептуальні основи функціонування системного програмного забезпечення багатозадачних, багатоядерних та багатопроцесорних паралельних комп'ютерних систем; знати основні способи організації обчислювальних процесів в ОС Linux та відповідні структури даних системного рівня; знати основні способи взаємодії паралельних процесів в ОС Linux та вміти використовувати їх для створення паралельних програм; знати та вміти використовувати

технології паралельного програмування на рівні програмних потоків, зокрема інтерфейс прикладного програмування POSIX Threads і технології паралельного програмування OpenMP та OneTBB; мати практичні навички роботи з технологіями паралельного програмування та бібліотеками паралельного програмування в ОС Linux; вміти створювати, тестувати та відлагоджувати паралельні програми в ОС Linux з розпаралелюванням на рівні обчислювальних процесів та на рівні програмних потоків.

Навчальний посібник призначений для студентів спеціальності 123 «Комп'ютерна інженерія» галузі знань 12 «Інформаційні технології», зокрема для студентів другого (магістерського) рівня вищої освіти спеціалізацій 123.01 «Комп'ютерні системи та мережі», 123.02 «Системне програмування» та 123.04 «Кіберфізичні системи». В основу посібника покладено навчальні матеріали, розроблені та підготовлені автором для викладання навчальних дисциплін «Організація обчислювальних процесів у паралельних системах», «Технології паралельного програмування», «Високопродуктивні обчислення» та «Паралельне програмування високопродуктивних комп'ютерних систем» на кафедрі електронних обчислювальних машин Національного університету «Львівська політехніка».

1. Паралельні обчислення в ОС Linux

1.1. Організація паралельних обчислень

1.1.1. Проблема організації паралельних обчислень

Проблема організації паралельних обчислень є одним з найбільш перспективних напрямків розвитку сучасних комп'ютерних та інформаційних технологій. Використання паралельних комп'ютерних систем та паралельних програм дозволяє суттєво збільшити швидкість та ефективність розв'язування складних обчислювальних задач, в тому числі задач оптимізації, задач комп'ютерного моделювання в рамках наукових досліджень та промислового проектування, задач штучного інтелекту та машинного навчання, задач обробки великих наборів даних, задач біоінформатики та багатьох інших важливих задач.

З точки зору організації паралельних обчислень можна виділити два основних способи розпаралелювання:

1) Апаратне розпаралелювання, коли обчислення розподіляються між кількома обчислювальними вузлами: комп'ютерами, процесорами, ядрами процесора, тощо.

2) Розпаралелювання в режимі розділення у часі (time sharing), коли час роботи окремого обчислювального взула розподіляється між кількома обчислювальними процесами або програмними потоками.

В переважній більшості сучасних паралельних комп'ютерних систем одночасно реалізовані обидва способи розпаралелювання обчислень.

З точки зору загальної проблематики паралельних обчислень можна виділити три основні підходи (рис.1.1):

1) підхід на основі апаратного паралелізму – обчислення розпаралелюються внаслідок наявності паралельних апаратних засобів комп'ютерної системи;

2) підхід на основі алгоритмічного паралелізму – обчислення розпаралелюються внаслідок паралельності алгоритму, який вони реалізують;

3) підхід на основі утилітарного паралелізму – обчислення розпаралелюються внаслідок відповідної організації взаємодії користувачів з комп'ютерною системою з міркувань зручності роботи, наприклад, в рамках забезпечення багатокористувацького режиму роботи комп'ютерної системи.

Мета організації паралельних обчислень полягає в отриманні виграшу у продуктивності за рахунок прискорення обчислень, зменшення обчислювального навантаження на комп'ютерну систему та розподілу обчислювального навантаження між вузлами паралельної комп'ютерної системи [1-7].

Розглянемо послідовність кроків по проектуванню паралельної програми та організації паралельних обчислень в процесі переходу від поставленої задачі до відповідних обчислень.

1. Постановка обчислювальної задачі в рамках деякої предметної області та відповідних математичних моделей.

2. Розв'язок задачі, в тому числі шляхом розбиття задачі на підзадачі, як перший крок розпаралелювання відповідних обчислень.

3. Розроблення алгоритму розв'язку задачі з використанням принципів та методик проектування паралельних алгоритмів.

4. Проектування та розроблення паралельної програми, яка реалізує алгоритм розв'язку задачі, з використанням технологій паралельного програмування.

5. Виконання програми у комп'ютерній системі з розпаралелюванням на рівні обчислювальних процесів та/або на рівні програмних потоків.

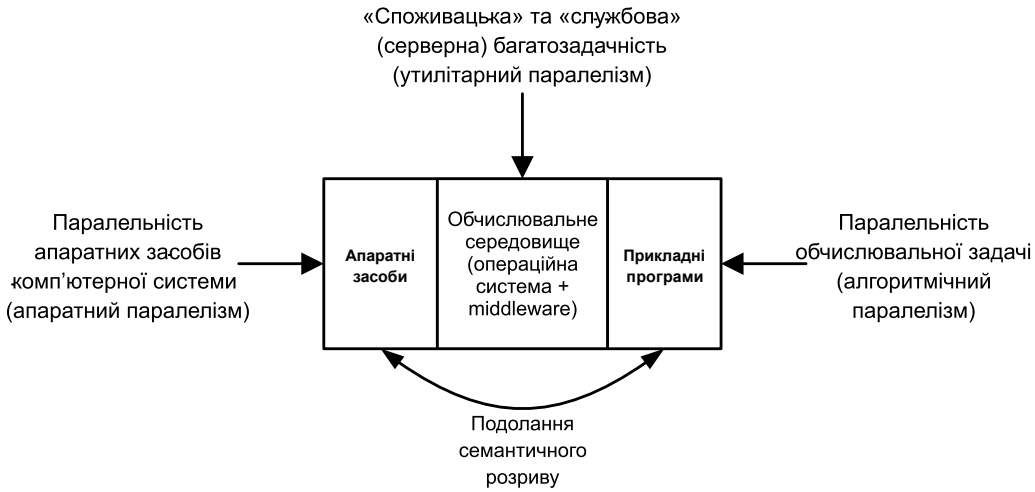

Рис.1.1. Проблема організації паралельних обчислень з точки зору різних підходів до розпаралелювання.

У відповідності до розглянутих кроків по проектуванню паралельної програми та організації паралельних обчислень, розрізняють наступні основні проблеми:

1) розробка паралельних алгоритмів, які, як правило, суттєво відрізняються за принципами роботи від своїх послідовних аналогів;

2) розробка та оптимізація паралельних програм, в тому числі складних програмних систем та комплексів;

3) розробка паралельних комп'ютерних систем, в тому числі нових паралельних архітектур та нових принципів розпаралелювання на апаратному рівні;

4) розробка програмних засобів (інструментів) та технологій для організації паралельних обчислень, зокрема

4.1) засобів організації паралельних обчислень на рівні операційної системи та системного програмного забезпечення (диспетчеризація, синхронізація, взаємодія процесів, оптимізуючі компілятори);

4.2) засобів організації паралельних обчислень на рівні проміжного програмного забезпечення (middleware);

4.3) засобів автоматизованого проектування паралельних програм та програмних систем (аналіз залежностей, візуалізація потоків команд та даних, виявлення ситуацій взаємного блокування, тощо);

5) розробка засобів автоматичного розпаралелювання, зокрема

5.1) на рівні переходу від алгоритму до паралельної програми;

5.2) на рівні переходу від програми до паралельних обчислювальних процесів та програмних потоків.

1.1.2. Паралельні комп'ютерні системи

Паралельна комп'ютерна система – це комп'ютерна система, яка складається з деякої кількості обчислювальних вузлів (ядер процесора, процесорів, комп'ютерів, тощо), об'єднаних каналами передачі даних з високою пропускною здатністю та/або використовуючих спільну апаратну пам'ять. Структура паралельної комп'ютерної системи дозволяє виконувати обчислення з явним (апаратним) розпаралелюванням. Окремим різновидом паралельних комп'ютерних систем є, так звані, розподілені комп'ютерні системи. Розподілена система (distributed system) – це комп'ютерна система, в якій розв'язуються складні обчислювальні завдання з використанням двох і більше комп'ютерів, об'єднаних в мережу. Розподілену обчислювальну систему можна розглядати як слабо-зв'язний набір

окремих обчислювачів (вузлів), об'єднаних в рамках єдиної комп'ютерної мережі. В ході роботи розподіленої обчислювальної системи забезпечується доступ процесів даного вузла до віддалених ресурсів інших вузлів та надання локальних ресурсів для використання процесами віддалених вузлів.

Зауважимо, що машина фон Неймана, яка є архітектурною основою переважної більшості сучасних комп'ютерних систем, по своїй будові є послідовною обчислювальною машиною, в якій обчислення виконуються покроково. Ця обставина є одною з основних причин тих проблем, які виникають при організації паралельних обчислень в сучасних паралельних комп'ютерних системах, оскільки, не дивлячись на реалізований в них апаратний паралелізм, в їх основі все одно лежать архітектурні принципи послідовної машини фон Неймана, в тому числі архітектурний принцип RAM (random access memory). При цьому основний підхід до побудови паралельних архітектур сучасних комп'ютерних систем полягає у "клонуванні" обчислювальних вузлів з RAM, використовуючи архітектурний принцип parallel RAM (PRAM). В результаті паралельна комп'ютерна система складається з кількох обчислювальних вузлів, які мають локальну та спільну пам'ять (рис.1.2). В якості альтернативного архітектурного принципу, який є більш вигідним з точки зору розпаралелювання обчислень, можна навести машину потоків даних (data flow machine). З точки зору узагальнення та класифікації різних варіантів забезпечення апаратного паралелізму використовується класифікація Фліна, яка розрізняє чотири основних архітектури: SISD, SIMD, MISD, MIMD.

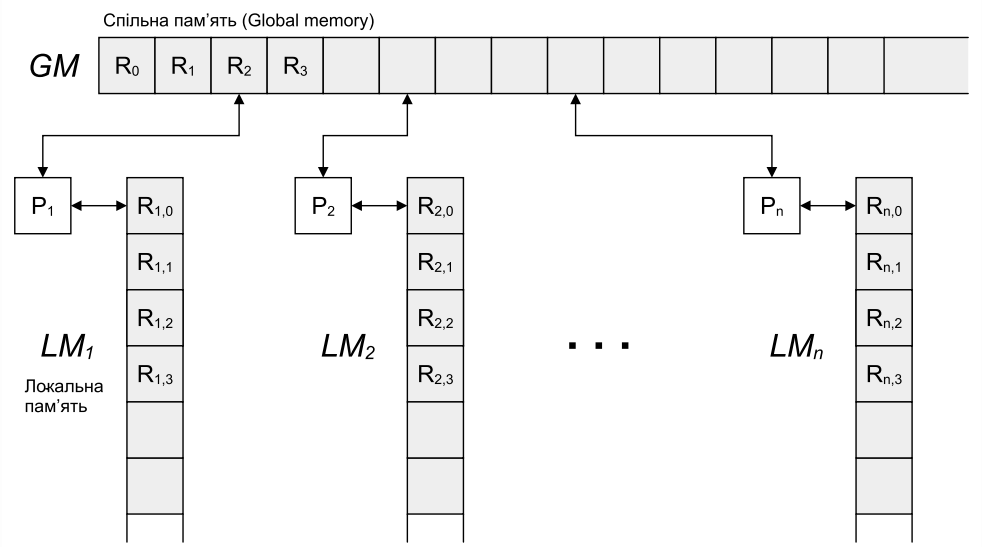

Рис. 1.2. Застосування архітектурного принципу parallel RAM (PRAM): апаратна пам'ять комп'ютерної системи поділяється на локальну (LM) та спільну (GM).

Розглянемо класифікацію паралельних комп'ютерних систем з точки зору основної проблеми розподілених систем, яка полягає в тому, що змінення станів окремих елементів системи відбувається швидше (в масштабі часу t_1), ніж інформація про це надходить іншим елементам системи (в масштабі часу t_2), тобто $t_1 < t_2$. З точки зору співвідношення t_1 і t_2 можна виділити наступні основні типи паралельних комп'ютерних систем (в порядку зростання різниці між t_1 і t_2):

1) сильно-зв'язані паралельні комп'ютерні системи, в тому числі системи з архітектурами Massively Parallel Processors (MPP), Shared Memory Multi-Processor (SMMP) та Symmetric MultiProcessor (SMP);

2) слабо-зв'язані паралельні комп'ютерні системи, в тому числі обчислювальні кластери різного призначення, наприклад Beowulf cluster;

3) мережні системи, в яких обчислювальні вузли об'єднані в рамках локальної комп'ютерної мережі (LAN);

4) розподілені комп’ютерні системи, побудовані за схемою клієнт-сервер (client-server) чи за схемою одноранговоі взаємодії (peer-to-peer, p2p), в тому числі розподілені системи на основі мережі Інтернет, наприклад, системи грід-обчислень (grid computing).

Зауважимо, що одною з найбільш розповсюджених архітектур для побудови сильно-зв’язаних паралельних комп’ютерних систем є SMP (Symmetric MultiProcessing) – архітектура багатопроцесорних комп'ютерів, в яких два або більше однакових процесора підключаються до спільної пам'яті, зокрема більшість сучасних багатопроцесорних систем використовують архітектуру SMP.

Серед прикладів паралельних комп’ютерних систем відмітимо:

- системи високо-продуктивних обчислень (high-performance computing), зокрема суперкомп'ютери (див. список www.top500.org), багатоядерні процесори (multi-core processors) та ін.;

- графічні прискорювачі (general-purpose computing on graphics processing units, GPGPU), які додатково використовуються для виконання паралельних обчислень, зокрема графічні прискорювачі з архітектурою Nvidia CUDA (Compute Unified Device Architecture) та відповідні технології паралельних обчислень: OpenCL (Open Computing Language), OpenACC (open accelerators), C++ AMP (C++ Accelerated Massive Parallelism) та ін.;

- прискорювачі обчислень (hardware acceleration), зокрема прискорювачі побудовані на основі принципу обчислювальної еквівалентності апаратного та програмного забезпечення (computational equivalence of hardware and software), в тому числі з використанням мов опису апаратури (Verilog, VHDL, тощо); реконфігуровані прискорювачі (reconfigurable computing) на основі ПЛІС (FPGA), які здатні адаптувати структуру обчислювача під час виконання обчислень шляхом завантаження її у FPGA; прискорювачі алгоритмів штучного інтелекту (AI accelerators) та нейроморфні обчислювачі (neuromorphic computing).

1.1.3. Паралельне програмування

Паралельне програмування – це реалізація паралельних алгоритмів з використанням можливостей паралельних комп’ютерних систем, в тому числі забезпечення паралельного виконання системних та користувацьких обчислювальних процесів. Для цього застосовуються технології паралельного програмування – набори програмних засобів та інструментів для реалізації паралельних програм.

Головним принципом, покладеним в основу розроблення паралельних програм є декомпозиція, зокрема

1) функціональна декомпозиція, коли деяка загальна функція розбивається на окремі під-функції, що можуть виконуватись паралельно,

2) декомпозиція даних, коли вхідні дані розбиваються на окремі частини, що можуть оброблятись паралельно.

На системному рівні паралельні обчислення представлені у вигляді паралельних обчислювальних процесів. Схема породження та запуску паралельних процесів, як правило, реалізується у вигляді графу-дерева: в одного обчислювального процесу може бути декілько синівських процесів, і в кожного процесу є один батьківський процес. Оскільки в сучасних операційних системах обчислювальні процеси “ізольовані” один від одного з міркувань надійності та безпеки, то додатково застосовуються спеціальні системні програмні засоби, які забезпечують обмін інформації та взаємодію між паралельними обчислювальними процесами (Inter-Process Communication, IPC). Крім цього застосовуються системні програмні засоби для синхронізації виконання паралельних обчислювальних процесів, зокрема для організації їх доступу до спільних даних та ресурсів.

Для формального представлення та опису паралельних обчислень використовують, так звані, моделі паралельних обчислень (models of concurrent computing). Моделі паралельних обчислень використовують з метою:

1) отримання універсальних рішень (які не залежать від апаратної платформи комп’ютерної системи та її конфігурації);

2) дослідження загальних принципів та законів розпаралелювання обчислень.

Різні типи моделей паралельних обчислень можна розділити на дві основні групи:

1) графо-компонентні моделі:

1.1) Parallel RAM,

1.2) Petri nets,

1.3) Actor model,

2) алгебри процесів (числення процесів, process calculus):

2.1) CSP - communicating sequential processes,

2.2) CCS - Calculus of communicating systems,

2.3) ACP - Algebra of Communicating Processes,

2.4) π-calculus,

2.5) Ambient calculus,

2.6) Trace theory та ін.

Окрім моделі паралельних обчислень використовується також схоже поняття «модель паралельного програмування» (parallel programming model). Модель паралельного програмування – це узагальнене представлення архітектури паралельного комп'ютера на тому чи іншому рівні абстракції, за допомогою якої зручно проектувати паралельні алгоритми та розробляти їх програмну реалізацію. Зручність моделі паралельного програмування обумовлюється її універсальністю з точки зору можливостей опису різних обчислювальних задач в термінах різних паралельних архітектур. Моделі паралельного програмування реалізуються або у вигляді програмних бібліотек, або у вигляді мов паралельного програмування.

В паралельному програмуванні розрізняють наступні рівні програмного паралелізму або «гранулярності» паралельних обчислень (у порядку збільшення масштабу одиниць розпаралелювання):

1) рівень машинних інструкцій (паралельні апаратні засоби, оптимізуючі компілятори);

2) рівень підпрограм (функцій, методів);

3) рівень об’єктів (структур даних з відповідним набором методів);

4) рівень програм/задач, наприклад розпаралелювання інтерфейсу користувача і підсистеми обміну повідомленнями через у програмному сервісі миттєвих повідомлень.

1.1.4. Способи організації паралельних обчислень

Способи організації паралельних обчислень визначають рівень розпаралелювання програми та принципи взаємодії і синхронізації її паралельних компонент. Різні способи організації паралельних обчислень можна класифікувати

1) за типом паралельної комп’ютерної системи, для якої створюється паралельна програма;

2) за типом моделі паралельних обчислень, яка використовується у проектуванні та реалізації паралельної програми.

З точки зору взаємодії паралельних компонент програми способи організації паралельних обчислень поділяються на два основних класи:

1) організація паралельних обчислень на основі спільної пам’яті (shared memory), коли паралельні компоненти програми мають одночасний доступ до одних і тих самих структур даних, навколо яких вибудовується взаємодія паралельних компонент [8];

2) організація паралельних обчислень на основі обміну повідомленнями (message passing), коли паралельні компоненти програми взаємодіють шляхом пересилання та отримання повідомлень.

З позицій оптимізації паралельних обчислень можна висунути гіпотезу про наявність деякого оптимального сполучення та співвідношення організації паралельних обчислень на

основі спільної пам’яті та обміну повідомленнями при реалізації паралельної програми, з урахуванням особливостей

1) відповідної обчислювальної задачі,

2) паралельної комп’ютерної системи, у якій виконується програма,

3) доступних програмі обчислювальних ресурсів на момент її виконання.

Один з підходів до оптимізації паралельних обчислень на основі цієї гіпотези полягає у використанні, так званих, координаційних просторів (coordination spaces). Ідея полягає в тому, щоб приховати різницю між спільною пам’яттю та обміном повідомленнями за допомогою програмного інтерфейсу організації паралельних обчислень більш високого рівня абстракції. Прикладом реалізації цієї ідеї може бути мова програмування Linda, заснована на концепції розподіленої спільної пам’яті (distributed shared memory).

Важливим питанням при організації паралельних обчислень є оцінка виграшу, отриманого в результаті розпаралелювання. Для цього, зокрема, виконується вимірювання часу виконання паралельної програми за різних умов (стан операційного середовища, поточна завантаженість комп’ютерної системи, тощо) та визначення обсягів обчислювальних ресурсів (процесорного часу, оперативної пам’яті, тощо), які споживає паралельна програма. Для розрахунку характеристик продуктивності паралельної програми використовують

- залежність продуктивності від кількості обчислювальних вузлів (ядер, процесорів, комп’ютерів);
- залежність продуктивності від пропускної здатності каналів зв’язку між обчислювальними вузлами (наприклад, процесорами чи комп’ютерами),
- оцінку співвідношення обчислювальних та комунікаційних витрат (computation-to-communication).

Іншим важливим питанням є перевірка коректності роботи паралельної програми, в тому числі з точки зору відсутності ситуацій взаємного блокування (deadlocks) її паралельних компонент. В деяких випадках замовник вимагає від розробника паралельної програми гарантій коректності її роботи, в тому числі на основі аналізу програми з використанням моделей паралельних обчислень.

Використання різних способів організації паралельних обчислень передбачає розв’язок наступних типів оптимізаційних задач:

- оптимізаційні задачі в рамкам концепції само-адаптивного програмного забезпечення (self-adaptive software), зокрема концепції адаптивного паралелізму (adaptive parallelism), тобто само-налаштування паралельної програми під особливості паралельної комп’ютерної системи та наявні обчислювальні ресурси в момент її виконання;
- оптимізаційні задачі в рамкам концепції адаптивного апаратного забезпечення (adaptive hardware), зокрема само-налаштування паралельної комп’ютерної системи під особливості обчислювальної задачі та умови, в яких вона вирішується;
- оптимальний розподіл обчислювального навантаження між компонентами паралельної комп’ютерної системи (load balancing).

Розмаїття способів організації паралельних обчислень відображається у технологіях паралельного програмування, серед яких зазначимо

- розпаралелювання на рівні обчислювальних процесів: POSIX parallel processes (fork, exec, wait, exit), UNIX/Linux IPC (pipes, named pipes, message queues, shared memory, semaphores), multiprocessing module (Python);
- розпаралелювання на рівні програмних потоків: POSIX Threads (pthreads), Windows Threads, Threads APIs (C++, Java, .NET), threading module (Python), QThread (Qt);
- засоби автоматичного розпаралелювання: OpenMP (Open Multi-Processing), oneTBB (oneAPI Threading Building Blocks, Intel TBB), Chapel (Cascade High Productivity Language, Cray), X10 (IBM), OpenCL, PyOpenCL, Erlang, Elixir.
- розпаралелювання в розподілених комп’ютерних системах: MPI (Open MPI, MPICH, pyMPI, mpi4py), Charm++ (Adaptive MPI, Charm4py), Slurm Workload Manager (Simple Linux Utility for Resource Management), BOINC (Berkeley Open Infrastructure for Network Computing), HTCondor.

1.2. Паралельні обчислювальні процеси в ОС Linux

1.2.1. Органіазація обчислювальних процесів в ОС Linux

Обчислювальний процес - це примірник деякої програми під час її виконання. Окремий обчислювальний процес складається з послідовності машинних інструкцій, які виконує процесор. В ОС Linux, яка відноситься до багатозадачних ОС, є можливість одночасно виконувати декілька обчислювальних процесів та організовувати взаємодію між ними за допомогою спільної пам'яті, обміну повідомленнями та інших механізмів [9-14]. Процеси виконуються паралельно як в режимі розділення у часі (time-sharing), так і "фізично" паралельно, якщо апаратні засоби надають таку можливість (мульти-процесорна система, багатоядерний процесор).

З точки зору паралельних обчислень всі програми можна поділити на послідовні та паралельні. Для виконання послідовної програми породжуються один обчислювальний процес. Для виконання паралельної програми породжуються або декілька процесів, або декілька програмних потоків (threads) у складі одного процесу.

Блок управління процесом (Process Control Block), який містить всю необхідну інформацію про окремий процес, в ОС Linux називається дескриптором процесу (process descriptor), і є примірником структури task_struct (оголошена у <linux/sched.h>) (рис.1.3). Один примірник цієї структури займає в пам'яті приблизно 1.7kb. Дескриптори процесів об'єднані в кільцевий двонаправлений (circular doubly linked) список процесів (task list). На множині всіх процесів визначено бінарне відношення «батько-син» (parent-child), на основі якого будується дерево процесів (tree of processes).

```
struct task_struct {
      volatile long state;
      void *stack;
      unsigned int flags;

      int prio, static_prio;
      struct sched_entity se;
      struct list_head tasks;
      struct mm_struct *mm, *active_mm;

      pid_t pid;
      pid_t tgid;
      struct task_struct *real_parent;
      struct list_head children;

      char comm[TASK_COMM_LEN];

      struct thread_struct thread;

      struct files_struct *files;

      ...
};
```

Рис.1.3. Структура task_struct.

До основних параметрів та атрибутів процесу відносяться:
- стан процесу (state),
- пріоритет (prio, static_prio),
- ідентифікатор процесу (pid),
- ідентифікатор групи процесів, до якої належить процес (tgid),
- ідентифікатор батьківського процесу (ppid),

- командний рядок, яким процес був запущений на виконання (comm) та ін.

Контекст процесу (process context) - це стан регістрів процесора та стан операційного середовища поточного процесу. Зокрема в UNIX-подібних ОС розрізняють: користувацький контекст, регістровий контекст та контекст системного рівня. Переключення контексту (context switch) - це процес збереження стану процесу або потоку, щоб забезпечити можливість відновити його виконання пізніше, та завантаження стану іншого процесу або потоку для продовження його виконання.

Основні стани процесу в ОС Linux (табл.1.1.):

- R: виконується або готовий до виконання і чекає доступу до процесора (running or runnable),
- S: очікує на завершення деякої події, наприклад, завершення вводу з терміналу користувача (interruptible sleep),
- D: очікує на якусь подію, очікування не може бути перервано сигналом,
- Z: завершений процес, інформація про статус якого зберігається для коректного спрацювання системних викликів waitpid та wait,
- T: зупинений (виконання процесу призупинено, після того як він отримав сигнал SIGSTOP; виконання процесу продовжиться після того, як йому буде надіслано сигнал SIGCONT).

Таблиця 1.1. Основні стани процесу в ОС Linux та їх позначення.

Value	State	Abbreviation	Meaning
0	TASK_RUNNING	R	Running or runnable
1	TASK_INTERRUPTIBLE	S	Interruptible sleep
2	TASK_UNINTERRUPTIBLE	D	Non interruptible sleep
4	__TASK_STOPPED	T	Stop status, when the process receives sigstop and other signal information
8	__TASK_TRACED	t	Trace the status, and the process is suspended by the debugger program, such as debugging with ptrace()
16	EXIT_ZOMBIE	Z	Zombie state, call do_exit() at the end of the process to enter zombie first
32	EXIT_DEAD	X	Death status. After the parent process uses waitpid() or wait4() to recycle the dead child process, the status changes from exit - zombie to exit - dead

Процеси в ОС Linux поділяються на 1) користувацькі процеси (user processes), 2) системні процеси (daemon processes) та 3) процеси ядра (kernel processes). Окремий процес може виконуватись в інтерактивному (foreground) або у фоновому (background) режимі. Також з точки зору прав доступу до системних та апаратних ресурсів процес може виконуватись в режимі користувача (user mode) або в режимі ядра (kernel mode). Для запуску процесу у фоновому режимі використовується символ “&”. Для запуску процесу у режимі ігнорування сигналу припинення роботи терміналу використовується команда nohup. Основні команди для роботи с процесами в ОС Linux: ps, pstree, top (htop), kill, nice, nohup та ін.

Найпростіший засіб організації взаємодії паралельних процесів (inter-process communication), який використовується у командному рядку, – це конвеєр команд (pipeline). Приклад використання конвеєру команд:

```
$ ls -l | grep key | less
```

1.2.2. Запуск та породження нових процесів в ОС Linux

Для породження нового процесу в ОС Linux використовується системний виклик **fork()** (рис.1.4). В разі успішного виконання системного виклику fork() ядро створює копію (clone) батьківського процесу (parent process), який зробив цей виклик, і запускає цю копію - синівський процес (child process) на виконання (рис.1.5). При цьому в батьківському процесі fork() повертає ідентифікатор породженого синівського процесу, в синівському процесі fork() повертає 0.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Рис.1.4. Опис системного виклику fork().

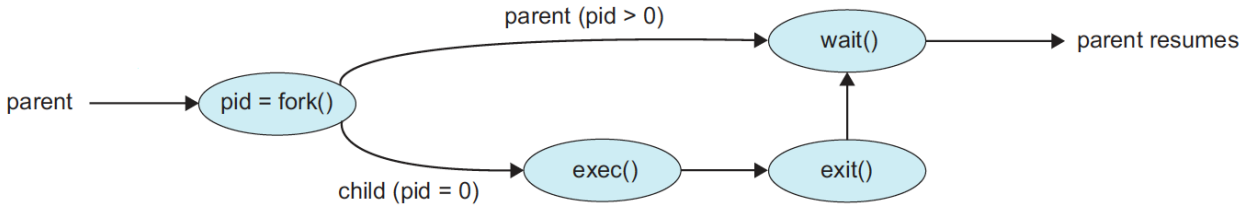

Рис.1.5. Схема використання системних викликів fork, exec, wait, exit.

В момент створення синівський процес наслідує від батьківського процесу:

1. Оточення (в тому числі значення змінних оточення).
2. Способи обробки сигналів (тобто налаштування SIG_DFL, SIG_IGN, SIG_HOLD, і адреси функцій обробки сигналів).
3. Дозвіл перевстановлювати діючий ідентифікатор користувача.
4. Дозвіл перевстановлювати діючий ідентифікатор групи.
5. Значення поправки до пріоритету.
6. Всі приєднанні сегменти спільної пам’яті (shared memory).
7. Ідентифікатор групи процесів.
8. Поточний робочий каталог.
9. Кореневий каталог.
10. Маску режиму створення файлів.
11. Обмеження на розмір файлу.

Породжений синівський процес відрізняється від батьківського наступним:

1. Породжений процес має свій унікальний ідентифікатор процесу.
2. Породжений процес має інший ідентифікатор батьківського процесу, рівний ідентифікатору процесу, який його породив.
3. Породжений процес має свої власні копії батьківських дескрипторів файлів. Кожний дескриптор файлу породженого процесу розділяє з відповідним батьківським дескриптором файлу спільний вказівник поточної позиції у файлі.

Синівський процес не може бути породжений у двох основних випадках (в цих випадках системний виклик fork() повертає відповідну помилку):

1) створити процес забороняє системне обмеження на загальну кількість процесів,
2) створити процес забороняє системне обмеження на кількість процесів у одного користувача.

За допомогою системного виклику fork() може бути реалізована будь-яка схема розпаралелювання обчислень на рівні процесів (рис.1.6).

```
int childPID;

if((childPID = fork()) == -1)
  {
   fprintf(stderr,"Can't fork for a child.\n");
   exit(1);
  }
if( childPID == 0)
  {
   // child process
  }
  else
    {
      // parent process
    }
```

Рис.1.6. Приклад використання системного виклику fork().

Системний виклик **exec()** (рис.1.7) використовується для заміни образу процесу (process image), який викликав exec() на образ іншого процесу. В разі успішного виконання системного виклику exec() ядро замінює користувацький контекст викликаючого процесу на користувацький контекст процесу, який реалізує вказану у аргументах exec() виконавчу програму. Тобто замість коду і даних викликаючого процесу завантажується на виконання код і дані іншого процесу.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ... /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
                    /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Рис.1.7. Опис системного виклику exec().

Процес, який заміщає собою попередній процес, що викликав exec, успадковує у нього:
1. Значення поправки до пріоритету.
2. Ідентифікатор процесу.
3. Ідентифікатор батьківського процесу.
4. Ідентифікатор групи процесів.
5. Час, що залишився до спрацьовування будильника.
6. Поточний робочий каталог.
7. Кореневий каталог.
8. Маску режиму створення файлів.
9. Обмеження на розмір файлу.

Всього є шість варіантів системного виклику **exec()** (рис.1.7), які відрізняються способом вказання яку виконавчу програму буде реалізувати новий процес. Зокрема суфікс <**l**> вказує на те, що параметрами exec() “моделюється” командний рядок (command **l**ine), тобто параметри системного виклику exec(), починаючи з другого, еквівалентні відповідним аргументам командного рядку для запуску програми на виконання. Суфікс <**v**> вказує на те, що аргументи командного рядку подаються у вигляді масиву (argv[] – argument **v**ector). Суфікс <**p**> вказує на те, що якщо у першому параметрі exec() (const char *file) не зустрічається

символ ’/’ (slash), то виконавчий файл із вказаною назвою буде шукатися у списку шляхів до виконавчих файлів, який знаходиться у змінній оточення **P**ATH. Суфікс <**e**> вказує на те, що серед параметрів exec() додатково вказується список змінних оточення зі значеннями (параметр envp[] - **e**nvironment of the executed program).

Першим параметром системного виклику exec() завжди є шлях до виконавчого файлу (const char *path), або його назва (const char *file). Оскільки системний виклик exec() у варіантах execl(), execlp() та execle() може мати змінну кількість параметрів, то в цих випадках останнім параметром обов’язково має бути нульовий вказівник (null pointer), який сигналізує про кінець командного рядка, що формується параметрами exec(). Так само останнім елементом масивів argv[] та envp[] у відповідних варіантах exec() має бути нульовий вказівник ((char *) NULL).

Повернення з exec() у викликаючий процес – це помилка. В цьому разі exec() повертає -1. Це означає, що з якихось причин не вдалося виконати заміщення поточного процесу іншим процесом (наприклад, невірно вказано шлях до виконавчого файлу у першому параметрі exec()). Приклад використання системного виклику exec() показано на рис.1.8.

```
int childPID;

if((childPID = fork()) == -1)
  {
   fprintf(stderr,"Can't fork for a child.\n");
   exit(1);
  }
if( childPID == 0)
  {
     // child process
     if(execl("./child","child",0)==-1)
     {
      fprintf(stderr,"Can't execute the external child.\n");
      exit(1);
     }
  }
  else
    {
      // parent process
    }
```

Рис.1.8. Приклад використання системного виклику exec().

1.2.3. Запит та встановлення атрибутів процесу в ОС Linux

Для отримання значень атрибутів процесу (запиту атрибутів) в ОС Linux використовується набір функцій з префіксом <get>.

Приклади функцій запиту атрибутів процесу:

getpid() - ідентифікатор процесу, який зробив виклик getpid(),

getppid() - ідентифікатор батьківського процесу,

getpgrp() - ідентифікатор групи процесів,

getuid() - реальний ідентифікатор користувача, якому належить процес,

geteuid() - діючий ідентифікатор користувача, якому належить процес,

getgid() - реальний ідентифікатор групи користувачів, до якої належить процес,

getegid() - діючий ідентифікатор групи користувачів, до якої належить процес,

getpriority() - пріоритет процесу.

Для встановлення атрибутів процесу в ОС Linux використовується набір функцій з префіксом <set>.

Приклади функцій встановлення атрибутів процесу:

setpgrp() - встановлює ідентифікатор групи процесів рівним ідентифікатору даного процесу і повертає значення tpgrp,

setuid() - встановлює реальний ідентифікатор користувача,

setgid() - встановлює реальний ідентифікатор групи користувачів,

setpriority() - встановити пріоритет процесу,

nice() - встановити пріоритет процесу (до поточного значення пріоритету додається аргумент функції nice()).

Системний виклик **nice()** (рис.1.9) додає значення inc до користувацького пріоритету (nice value) процесу або програмного потоку. Чим більше значення користувацького пріоритету (nice value), тим більш низький пріоритет має задача. Діапазон користувацького пріоритету (nice value) - від +19 (низький пріоритет) до -20 (високий пріоритет). Спроби встановити користувацький пріоритет (nice value) поза діапазоном обмежуються границями діапазону без повернення помилки. Системний виклик nice() повертає нове значення користувацького пріоритету (nice value).

```
#include <unistd.h>

int nice(int inc);
```

Рис.1.9. Опис системного виклику nice().

1.2.4. Завершення виконання процесів в ОС Linux

Системні виклики **wait()** та **waitpid()** (рис.1.10) призначенні для організації коректного завершення батьківського процесу, для чого він перед своїм завершенням повинен дочекатися завершення своїх синівських процесів.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Рис.1.10. Опис системних викликів wait() та waitpid().

Системний виклик waitpid() блокує виконання поточного процесу доти, доки не завершиться породжений ним синівський процес з ідентифікатором pid, або поточний процес не одержить сигнал, для якого встановлена реакція за замовчуванням "завершити процес" (або реакція обробки функцією користувача). Якщо породжений процес, заданий параметром pid, до моменту системного виклику закінчив виконання, то системний виклик повертається негайно без блокування поточного процесу.

Параметр pid визначає породжений процес, завершення якого чекає процес-батько, у такий спосіб:

- якщо pid > 0 очікуємо завершення процесу з ідентифікатором pid,
- якщо pid = 0, то очікуємо завершення будь-якого породженого процесу в групі, до якої належить процес-батько,
- якщо pid = -1, то очікуємо завершення будь-якого породженого процесу,
- якщо pid < -1, то очікуємо завершення будь-якого породженого процесу з групи, ідентифікатор якої дорівнює абсолютному значенню параметра pid.

За допомогою статусу (параметр wstatus) можна дізнатися спосіб завершення синівського процесу (чи його було зупинено, чи він завершився сам). Якщо синівський процес

завершився сам, то статус wstatus вказує причину завершення. Статус (wstatus) трактується в такий спосіб:

- якщо породжений процес завершився «самостійно» за допомогою системного виклику exit(), то молодші 8 біт статусу будуть рівні 0, а старші 8 біт будуть містити молодші 8 біт аргументу, який породжений процес передав системному виклику exit();

- якщо породжений процес завершився через одержання сигналу, то старші 8 біт статусу будуть рівні 0, а молодші 8 біт будуть містити номер сигналу, що викликав завершення процесу.

Параметр options задає деякі опції виконання системного виклику. Якщо значення options дорівнює WNOHANG (=1) повернення з виклику відбувається негайно без блокування поточного процесу в будь-якому випадку.

При виявленні процесу, що завершився, системний виклик повертає його ідентифікатор. Якщо виклик був зроблений із встановленою опцією WNOHANG, і породжений процес, специфікований параметром pid, існує, але ще не завершився, системний виклик поверне значення 0. У всіх інших випадках він повертає негативне значення. Якщо виконання системного виклику waitpid завершилося внаслідок одержання сигналу, то результат буде дорівнювати -1, а змінній errno буде присвоєно значення EINTR (переривання системного виклику).

Системний виклик wait() є синонімом системного виклику waitpid() зі значеннями параметрів pid = -1, options = 0. Приклад використання системного виклику wait() показано на рис.1.11.

```
int childPID;

if((childPID = fork()) == -1)
  {
   fprintf(stderr,"Can't fork for a child.\n");
   exit(1);
  }
if( childPID == 0)
  {
     // child process
     if(execl("./child","child",0)==-1)
     {
      fprintf(stderr,"Can't execute the external child.\n");
      exit(1);
     }
  }
  else
    {
     // parent process
     printf("Parent: My PID=%d, child PID=%d \n", getpid(), childPID);
     for(i=0; i<10; i++)
        {
         printf("PARENT now is working.\n");
         for(j=0;j<1000000;j++);
        }
      wait(&status);
      printf("Parent: Child exit code is %d.\n",(status&0xff00)>>8);
      exit(0);
    }
```

Рис.1.11. Приклад використання системного виклику wait().

Системний виклик **exit()** (рис.1.12) призначений для завершення виконання процесу. Системний виклик exit() завершує процес, що звернувся до нього, при цьому послідовно виконуються наступні дії:

1) у процесі, що викликав exit(), закриваються всі дескриптори відкритих файлів;

2) якщо батьківський процес знаходиться в стані виклику wait(), то цей виклик wait() завершується, повертаючи батьківському процесу як результат ідентифікатор завершеного процесу і молодші 8 біт коду його завершення (аргумент status системного виклику exit()).

```
#include <stdlib.h>

void exit(int status);
```

Рис.1.12. Опис системного виклику `exit()`.

Якщо батьківський процес не знаходиться в стані виклику wait(), то процес, що викликав exit(), переходить у стан зомбі (zombie). Це такий стан, коли процес лише займає рядок у таблиці процесів і не займає пам'яті ні в адресному просторі користувача, ні в адресному просторі ядра. Це потрібно для того, щоб пізніше, коли батьківський процес викличе wait(), цей виклик буд оброблений коректно.

Функція **sleep()** (рис.1.13) призначена для призупинки роботи викликаючого процесу на задану кількість секунд. Тобто виконання процесу припиняється на задане параметром seconds число секунд.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Рис.1.13. Опис функції sleep().

Час фактичного призупинення може виявитися менше заданого з двох причин:

- плановані пробудження процесів відбуваються у фіксовані секундні інтервали часу, відповідно до внутрішнього годинника;
- будь-який перехоплений сигнал перериває "сплячку", після чого спрацьовує реакція на сигнал.

З іншого боку, фактичний час призупинення може виявитися більше заданого через те, що система зайнята іншою, більш пріоритетною діяльністю. Функція sleep() повертає або 0, якщо минув увесь заданий параметром seconds час, або кількість секунд, що залишились до кінця “сну”, якщо виклик був перерваний обробником сигналу (тобто заданий час мінус фактичний).

1.2.5. Організація взаємодії обчислювальних процесів в ОС Linux

Взаємодія обчислювальних процесів (Inter-Process Communication, IPC) – це набір механізмів, які надає операційна система для забезпечення обміну даними між процесами, роботи процесів зі спільними даними та синхронізації процесів. Потреба у механізмах IPC обумовлена ізоляцією процесів один від одного в сучасних ОС задля надійності та безпеки обчислень на системному рівні (в тому числі з використанням механізму віртуальної пам’яті). Відтак концепція IPC доповнює концепцію незалежного адресного простору для кожного обчислювального процесу в ОС UNIX, ОС Linux та інших UNIX-подібних ОС.

Механізми взаємодії обчислювальних процесів в ОС Linux призначені для обміну даними між процесами можуть працювати у двох режимах: синхронному (по замовчуванню) та асинхронному. Відповідно ці механізми обміну даними можуть одночасно використовуватись, як засіб синхронізації процесів.

В ОС Linux та інших UNIX-подібних ОС використовуються два основних стандарти програмних інтерфейсів IPC системного рівня:

1) System V Release 4 (SVR4): System V interprocess communication mechanisms (svipc),

2) POSIX (Portable Operating System Interface).

Коротку довідкову інформацію про svipc можна отримати за допомогою команди:

```
$ man 5 ipc
```

До механізмів взаємодії обчислювальних процесів (IPC) в ОС Linux та інших UNIX-подібних ОС відносяться:

1) обмін даними (локальний):
- неіменовані канали (anonymous pipes),
- іменовані канали (named pipes),
- черги повідомлень (message queues),
- D-Bus (Desktop Bus),
- сокети (UNIX domain sockets),

2) робота зі спільними даними:
- спільна пам’ять (shared memory),
- спільне використання файлів (files),
- спільне використання файлів, відображених в пам’ять (memory-mapped files),

3) інші:
- семафори (semaphores),
- сигнали (signals),

4) обмін даними (мережний):
- сокети Берклі (sockets),
- Transparent Inter Process Communication (TIPC),
- Remote procedure call (RPC) interfaces.

До механізмів взаємодії обчислювальних процесів (IPC) реалізованих в ОС Linux за стандартом System V Release 4 (SVR4) зокрема відносяться (див. man 5 ipc):

1) черги повідомлень (message queues),

2) сегменти спільної пам’яті (shared memory segments),

3) множини семафорів (semaphore sets).

Окремі примірники черг повідомлень, сегментів спільної пам’яті та множин семафорів називають об’єктами System V IPC.

Контрольні питання

1. В чому полягає мета організації паралельних обчислень?
2. В який спосіб відбувається розпаралелювання обчислень в комп'ютерних системах?
3. Які основні проблеми організації паралельних обчислень потрібно вирішувати задля створення та виконання паралельних програм?
4. Чим відрізняються між собою основні типи паралельних комп'ютерних систем?
5. Який головний принцип покладено в основу розроблення паралельних програм?
6. Для чого використовують моделі паралельних обчислень (models of concurrent computing)?
7. Чим відрізняються різні рівні програмного паралелізму?
8. В чому полягають основні способи організації паралельних обчислень і чим вони відрізняються?
9. Як організовано паралельне виконання обчислювальних процесів в ОС Linux?
10. Для чого використовується структура task_struct в ОС Linux?
11. В яких полях структури task_struct містяться дані, що використовуються для планування паралельного виконання обчислювальних процесів?
12. В яких основних станах може знаходитись обчислювальний процес в ОС Linux?
13. В який спосіб відбувається запуск та породження нових процесів в ОС Linux?
14. Для чого призначений системний виклик fork() і які значення він повертає?
15. В якій ситуації системний виклик fork() завершиться з помилкою?
16. Які параметри використовуються у різних варіантах системного виклику exec()?
17. Що вказується першим параметром у системного виклику execl()?
18. Які системні виклики використовуються для запиту та встановлення атрибутів процесу в ОС Linux?
19. Чим відрізняються системні виклики getpid() та getppid()?
20. Який системний виклик ОС Linux призначений для зміни пріоритету виконання обчислювального процесу?
21. Для чого призначений системний виклик waitpid()?
22. Що потрібно вказати у першому параметрі системного виклику waitpid() для очікування завершення будь-якого породженого процесу в групі, до якої належить батьківський процес?
23. Чим відрізняються системні виклики waitpid() та wait()?
24. Яку послідовність дій виконує обробник системного виклику exit()?
25. Після чого процес в ОС Linux переходить в стан zombie?

2. Взаємодія обчислювальних процесів в ОС Linux

2.1. Неіменовані та іменовані канали

2.1.1. Неіменовані канали (anonymous pipes)

Неіменований канал (anonymous pipe) – це механізм IPC призначений для обміну даними між процесами за принципом FIFO (рис.2.1). Неіменований канал може бути використаний для обміну даними лише між процесами, які пов'язані один з одним в рамках відношення «батько-син». Наприклад, між процесом-батьком та породженим ним синівським процесом, або між двома синівськими процесами одного батьківського процесу.

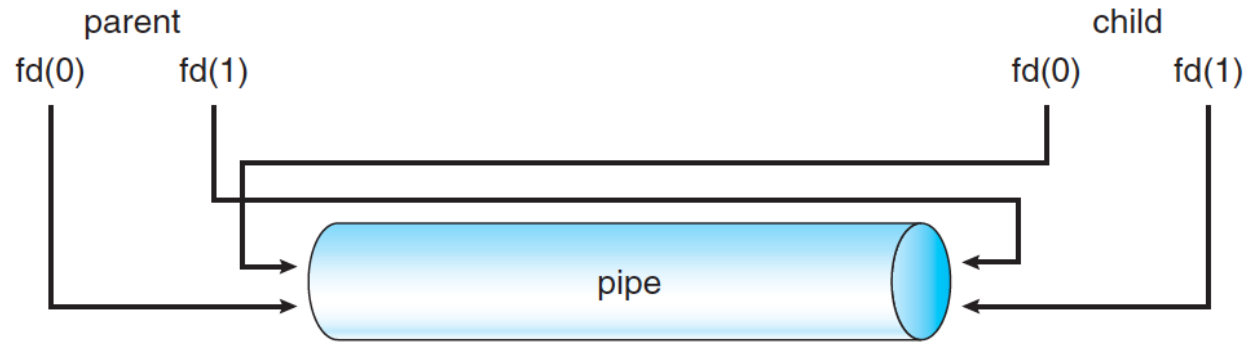

Рис.2.1. Схема організації обміну даними між двома процесами (parent, child) за допомогою неіменованого каналу (pipe).

Канал називається неіменованим (anonymous) тому що доступ до нього забезпечується у вигляді пари дескрипторів файлу (для читання - fd[0] та запису - fd[1]), в той час як "файлу" як такого (і відповідно його імені) не існує. Натомість неіменований канал є структурою даних ядра (як правило, у вигляді кільцевого буферу), через яку дані передаються у вигляді потоку байтів за принципом FIFO (рис.2.2). Дані читаються в тій послідовності, в якій пишуться в канал.

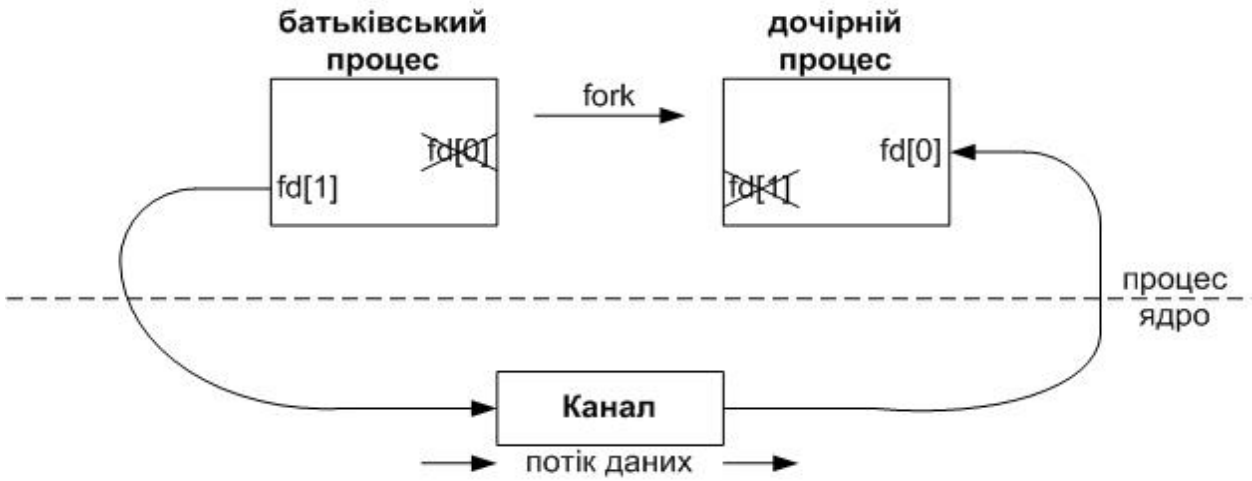

Рис.2.2. Схема типового використання неіменованого каналу.

Неіменований канал використовується в наступний спосіб. Батьківський процес створює неіменований канал у вигляді пари дескрипторів на запис та читання (fd[1], fd[0]). Після цього він породжує один або декілька синівських процесів, які успадковують від нього таблицю дескрипторів файлів, в тому числі дескриптори неіменованого каналу (fd[0], fd[1]). Оскільки батьківському процесу та його синівським процесам "відомі" значення дескрипторів fd[0],

fd[1], вони можуть обмінюватись даними через відповідний канал (тобто записувати і зчитувати з нього дані).

Схему обміну даними між процесами через канал визначає програміст згідно логіки роботи відповідної паралельної програми. Найбільш часто використовуються або схема однонаправленого потоку даних (рис.2.2), або схема двонаправленого потоку даних, яка передбачає використання двох каналів (рис.2.3).

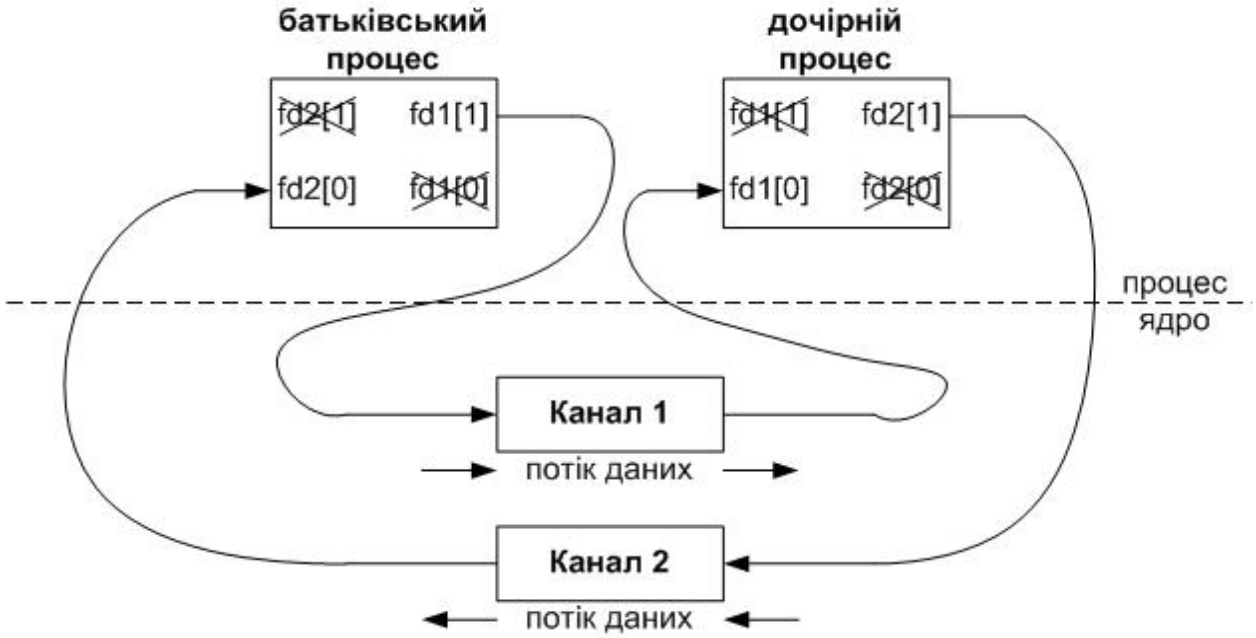

Рис.2.3. Схема використання двох каналів для організації двостороннього зв’язку між процесами.

Пропускна здатність або ємність каналу (pipe capacity) обумовлена тим, що канал буферизує деяку обмежену кількість байт даних (ця величина залежить від типу системи). Відтак канал має обмежену пропускну здатність. Якщо канал заповнений, виклик write() заблокує викликаючий процес або завершиться помилкою, залежно від того, чи встановлено прапорець O_NONBLOCK. Різні реалізації системи Linux мають різні обмеження на пропускну здатність каналу. Програма не повинна покладатися на конкретну ємність, вона має бути спроектована таким чином, щоб процес, що читає з каналу, отримував дані, як тільки вони стають доступними, щоб процес, що пише в канал, не блокувався.

У версіях Linux до 2.6.11 ємність каналу дорівнювала розміру системної сторінки (наприклад, 4096 байт для i386). Починаючи з Linux 2.6.11, ємність каналу становить 16 сторінок (тобто 65536 байт у системі з розміром сторінки 4096 байт). Починаючи з Linux 2.6.35, ємність каналу за замовчуванням становить 16 сторінок, але ємність можна запросити і встановити за допомогою операцій F_GETPIPE_SZ і F_SETPIPE_SZ системного виклику fcntl().

Канал (pipe) в режимі з блокуванням (O_NONBLOCK не встановлено, цей режим встановлено за замовчуванням) одночасно є засобом синхронізації. Спроба записати більшу ніж ємність каналу кількість даних (без зчитування даних іншим процесом) призведе до блокування процесу, що здійснює запис. Спроба прочитати дані з пустого каналу призведе до блокування процесу, що здійснює читання.

2.1.2. Створення неіменованого каналу

Неіменований канал створюється за допомогою системного виклику pipe() (рис.2.4), який повертає у параметр pipefd[] два дескриптори:

1) на читання з каналу - pipefd[0], та

2) на запис у канал - pipefd[1].

Після цього читання та запис відбувається за допомогою системних викликів read() та write().

```
#include <unistd.h>

int pipe(int pipefd[2]);

int pipe2(int pipefd[2], int flags);
```

Рис.2.4. Опис системних викликів pipe() та pipe2().

Системний виклик pipe() може не створити неіменований канал у двох випадках:

1) перевищена максимальна дозволена кількість файлів, відкритих одночасно одним процесом;

2) переповнено системну таблицю відкритих файлів.

При успішному завершенні виклик pipe() повертає 0. У випадку помилки виклик pipe() повертає -1, а змінній errno присвоюється код помилки.

В ОС Linux також реалізовано другий варіант цього системного виклику - pipe2() (рис.2.4), в якому другим параметром (flags) можна вказати різні режими використання створеного каналу, в тому числі забезпечити використання каналу в асинхронному режимі (O_NONBLOCK).

Ще один варіант створення неіменованого каналу – використання функції popen() (рис.2.5). Ця функція запустить (fork+exec) новий синівський процес (відповідний виконавчий файл задається параметром command), створить неіменований канал (pipe) для зв'язку з цим процесом та поверне дескриптор на запис чи читання відповідного каналу (тип дескриптора задається параметром type). Після завершення роботи канал, створений за допомогою popen(), потрібно закрити за допомогою функції pclose(fd) (close+wait) (рис.2.5).

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

Рис.2.5. Опис функцій popen() та pclose().

2.1.3. Робота з неіменованим каналом

Запис даних у канал відбувається за допомогою системного виклику write() (рис.2.6). Виклик write() намагається записати count байт з буфера, на який вказує аргумент buf, у файл, асоційований з дескриптором fd. Якщо флаг O_NONBLOCK встановлено (рис.2.7), то запис у переповнений канал призводить до негайного повернення значення 0 (асинхронний режим). Якщо O_NONBLOCK не встановлено, то запис у переповнений канал блокує викликаючий процес доти, доки не звільниться простір для запису (синхронний режим). За замовчуванням O_NONBLOCK не встановлено. При успішному завершенні виклик write() повертає кількості реально записаних у канал байт. У випадку помилки виклик write() повертає -1, а змінній errno присвоюється код помилки.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, const void *buf, size_t count);
```

Рис.2.6. Опис системних викликів read() та write().

```
#include <unistd.h>
.  .  .
fcntl(fd, F_SETFL, O_NONBLOCK);
.  .  .
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) & ~O_NONBLOCK);
```

Рис.2.7. Приклад встановлення та відміни режиму “без блокування” O_NONBLOCK за допомогою функції fcntl().

Читання даних з каналу відбувається за допомогою системного виклику read() (рис.2.6). Системний виклик read() намагається прочитати count байт з файлу, асоційованого з дескриптором fd, в буфер, на який вказує аргумент buf. Якщо O_NONBLOCK встановлено, то при спробі читання з порожнього каналу системний виклик read() негайно повертає значення 0 (асинхронний режим). Якщо O_NONBLOCK не встановлено, то читаючий процес блокується доти, доки дані не будуть записані у канал або не буде закритий дескриптор на запис у канал. При успішному завершенні виклик read() повертає кількість реально прочитаних байт. У випадку помилки виклик read() повертає -1, а змінній errno присвоюється код помилки.

Для того, щоб дізнатись скільки лишилося непрочитаних байт в каналі, можна скористатись функцією ioctl(), зокрема викликати її з такими параметрами: ioctl(fd, FIONREAD, &nbytes); де fd - дескриптор на читання чи запис каналу, FIONREAD - команда визначити кількість непрочитаних байт в каналі (ця команда не описана в жодному стандарті, але реалізована в багатьох системах), nbytes - змінна типу int, в яку виклик ioctl() поверне кількість непрочитаних байт в каналі.

Розглянемо порядок запису даних у канал з точки зору блокування операцій запису та одночасного запису даних у канал кількома процесами. Стандартом POSIX.1 визначено, що запис у канал функцією write() менше ніж PIPE_BUF байтів має бути атомарним. При цьому дані записуються у канал у вигляді безперервної послідовності. Запис більше PIPE_BUF кількості байтів може бути неатомарним: ядро може чергувати дані, які записують різні процеси. Стандарт POSIX.1 вимагає, щоб PIPE_BUF дорівнював не менше 512 байт. У Linux PIPE_BUF складає 4096 байт. Логіка одночасного запису канал залежить від того, чи встановлений для відповідного файлового дескриптора прапор блокування O_NONBLOCK, чи пишуть в канал кілька процесів і скільки байт n записується в канал.

1) O_NONBLOCK відключений, n <= PIPE_BUF: всі n байтів записуються атомарно; write() може заблокуватись, якщо в каналі немає місця для запису чергових n байтів.

2) O_NONBLOCK включений, n <= PIPE_BUF: якщо є місце для запису n байтів у канал, write(2) відразу завершується успішно, записуючи всі n байтів; в іншому випадку write() завершиться з помилкою і errno буде встановлено в EAGAIN.

3) O_NONBLOCK відключений, n > PIPE_BUF: запис в канал неатомарний; дані, які write() записує в канал, можуть чергуватись з даними, які записує в цей же канал write() іншого процесу; write() блокується доти, доки не буде записано n байтів. 4) O_NONBLOCK включений, n > PIPE_BUF: якщо канал заповнений, write() завершується з помилкою, а errno встановлюється в EAGAIN. В іншому випадку може бути записано від 1 до n байтів, тобто може відбутися «частковий запис». Тому в програмах доцільно перевіряти, яке значення повертає write(), щоб визначити, скільки байтів було фактично записано в канал, з

урахуванням того, що записані дані можуть чергуватись з даними, записаними в канал іншими процесами.

Після завершення роботи з неіменованим каналом потрібно закрити відповідні дескриптори на читання та запис за допомогою системного виклику close() (рис.2.8). Виклик close() закриває дескриптор fd (в такий спосіб процес повідомляє системі, що він завершив роботу з каналом). Виклик close() для останнього дескриптора неіменованого каналу приводить до ліквідації цього каналу (звільнення ресурсу). При успішному завершенні виклик close() повертає 0. У випадку помилки виклик close() повертає -1, а змінній errno присвоюється код помилки.

```
#include <unistd.h>

int close(int fd);
```

Рис.2.8. Опис системного виклику close().

При використанні неіменованих каналів існує загроза виникнення тупикової ситуації (deadlock). Наприклад, батьківський процес (parent) чекає на завершення синівського процесу (child), тобто батьківський процес заблокований на виклику wait(). Одночасно з цим синівський процес пише в канал, з якого може зчитати дані лише батьківський процес. В кінці кінців синівський процес заблокується на виклику write(), коли канал буде переповнений (оскільки заблокований на виклику wait() батьківський процес ніколи не зчитає з нього дані).

2.1.4. Іменовані канали (named pipes)

Іменований канал (named pipe, FIFO) – це механізм IPC призначений для обміну даними між процесами за принципом FIFO. Іменований канал відрізняється від неіменованого тим, що має ім’я. Іменування каналу забезпечується тим, що він відображається у файловій системі у вигляді спеціального файлу (як правило, в директорії tmp). Відтак ім’я каналу еквівалентне імені файлу і обирається програмістом (наприклад, ‘fifopipe’).

Наявність імені дає можливість організувати обмін даними через канал між процесами, що не пов’язані між собою в рамках відношення «батько-син» (тобто не успадковують один в одного таблицю дескрипторів файлів, як це відбувається у випадку використання неіменованих каналів). Підключення процесів до спільного іменованого каналу організується на основі попередньої «домовленості» про його назву.

Для створення іменованого каналу використовується системний виклик mknod (створити вузол (node) файлової системи) або функція mkfifo (створити іменований канал).

Системний виклик mknod() (рис.2.9) створює вузол файлової системи (звичайний файл, спеціальний файл пристрою чи іменований канал) згідно значень заданих параметрів: pathname – назва файлу, mode – тип файлу та права доступу до нього, dev – номер/ідентифікатор пристрою (якщо створюється спеціальний файл пристрою, в інших випадках значення dev ігнорується). В разі успішного виконання mkfifo() повертає 0, а в разі помилки -1.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Рис.2.9. Опис системного виклику mknod().

Для створення іменованого каналу за допомогою mknod() (рис.2.10) першим аргументом потрібно вказати його ім’я (FIFOPIPE), а другим аргументом комбінацію типу файлу (S_IFIFO - іменований канал) та прав доступу до нього (PERM).

```
#define FIFOPIPE "fifopipe"
#define PERM 0666
   .  .  .
   int errFlag;

   errFlag = mknod(FIFOPIPE, S_IFIFO|PERM, 0);
   if(errFlag == -1)
   {
      fprintf(stderr, "Can't create the named fifo pipe.\n");
      exit(1);
   }
```

Рис.2.10. Приклад використання системного виклику mknod()
для створення іменованого каналу.

Функція mkfifo() (рис.2.11) створює новий іменований канал із вказаним іменем (pathname) та правами доступу до нього (mode). В разі успішного виконання mkfifo() повертає 0, а в разі помилки -1.

```
       #include <sys/types.h>
       #include <sys/stat.h>

       int mkfifo(const char *pathname, mode_t mode);
```

Рис.2.11. Опис системного виклику mkfifo().

Після створення іменований канал потрібно відкрити для запису чи читання за допомогою системного виклику open(). Іменований канал, як правило, відкривається або лише на читання, або лише на запис (рис.3.12). Відповідно системний виклик open() повертає або дескриптор на читання, або дескриптор на запис у змінну fifoPipe (рис.2.12). Після цього іменований канал готовий для використання.

```
   fifoPipe = open(FIFOPIPE, O_RDONLY);
   fifoPipe = open(FIFOPIPE, O_WRONLY);
```

Рис.2.12. Приклад використання open() для відкриття іменованого каналу.

Для організації обміну даними через іменований канал використовуються системні виклики write() - запис даних у канал, та read() - читання даних з каналу (рис.2.6) за допомогою відповідних дескрипторів на запис та читання (fifoPipe) (рис.2.13).

```
   write(fifoPipe, buf, BUF_SIZE);
   read(fifoPipe, buf, BUF_SIZE);
```

Рис.2.13. Приклад використання write() та read() для обміну даними через іменований канал.

Після завершення роботи з іменованим каналом потрібно спочатку закрити відповідний дескриптор за допомогою системного виклику close() (рис.2.14).

```
close(fifoPipe);
```

Рис.2.14. Приклад використання close().

Для знищення іменованого каналу (звільнення ресурсу) після завершення роботи з ним використовується системний виклик unlink() (знищення імені (link) файлу, а також самого файлу, якщо в нього немає іншого імені) (рис.2.15).

```
errFlag = unlink(FIFOPIPE);
```

Рис.2.15. Приклад використання unlink().

2.2. Об’єкти System V IPC

2.2.1. Права доступу до об’єктів System V IPC

До механізмів взаємодії обчислювальних процесів (IPC) реалізованих в ОС Linux за стандартом System V Release 4 (SVR4) відносять (див. $man 5 ipc):

1) черги повідомлень (message queues),

2) сегменти спільної пам’яті (shared memory segments),

3) множини семафорів (semaphore sets).

Окремі примірники черг повідомлень, сегментів спільної пам’яті та множин семафорів називають об’єктами System V IPC (інші назви: примірники механізмів IPC, ресурси IPC).

Для кожного створеного об’єкту System V IPC система створює свій примірник структури ipc_perm (рис.2.16, рис.2.17), в полях якої міститься інформація про власника об’єкту і права доступу до нього.

```
            struct ipc_perm {
                uid_t          cuid;   /* creator user ID */
                gid_t          cgid;   /* creator group ID */
                uid_t          uid;    /* owner user ID */
                gid_t          gid;    /* owner group ID */
                unsigned short mode;   /* r/w permissions */
            };
```

Рис.2.16. Основні поля структури ipc_perm.

```
struct kern_ipc_perm {
      spinlock_t        lock;
      bool              deleted;
      int               id;
      key_t             key;
      kuid_t            uid;
      kgid_t            gid;
      kuid_t            cuid;
      kgid_t            cgid;
      umode_t           mode;
      unsigned long     seq;
      void              *security;

      struct rhash_head khtnode;

      struct rcu_head   rcu;
      refcount_t refcount;
} ____cacheline_aligned_in_smp __randomize_layout;
```

Рис.2.17. Структура kern_ipc_perm (include/linux/ipc.h)
в програмному коді ядра Linux (v.5.12-rc2).

Права доступу до об’єкту IPC призначаються тим, хто його створює (cuid), для користувача-власника об’єкту (uid) та усіх інших користувачів. В молодших 9-ти бітах поля структури ipc_perm.mode у форматі rwx-rwx-rwx визначаються права по здійсненню операцій (доступу) над відповідним об’єктом IPC.

Вміст поля структури ipc_perm.mode інтерпретується системою в наступний спосіб (як будь яка комбінація з наведених варіантів):

0400 - дозвіл на читання користувачем-власником (uid),

0200 - дозвіл на запис користувачем-власником (uid),

0040 - дозвіл на читання користувачами з групи власника (gid),

0020 - дозвіл на запис користувачами з групи власника (gid),

0004 - дозвіл на читання усіма іншими користувачами,

0002 - дозвіл на запис усіма іншими користувачами.

Поля структури ipc_perm.cuid та ipc_perm.cgid приймають значення в момент створення відповідного об’єкту IPC системним викликом ***get (msgget, shmget, semget) і дорівнюють uid та gid користувача - власника процесу, який звертається до системи з викликом ***get. Значення полів структури ipc_perm.uid та ipc_perm.gid від початку мають такі самі значення, як поля ipc_perm.cuid та ipc_perm.cgid, але можуть бути змінені за допомогою системного виклику ***ctl (msgctl, shmctl, semctl).

Перевірка прав доступу процесу до об’єкту IPC виконується в два етапи:

1) Під час спроби процесу отримати доступ до існуючого об'єкту IPC за допомогою системного виклику ***get() здійснюється первинна перевірка відповідності прав доступу, які запитує процес (параметр msgflg), та прав доступу, визначених у полі mode відповідної структури ipc_perm. У разі невідповідності виклик ***get повертає помилку і доступ до об’єкту IPC не надається.

2) Під час будь-якої операції з об'єктом IPC здійснюється перевірка прав доступу процесу, що намагається здійснити цю операцію. Наприклад, операція дозволяється, якщо діючий ідентифікатор користувача - власника процесу збігається зі значенням uid або cuid об'єкту IPC і встановлено відповідний біт дозволу доступу в полі mode структури ipc_perm. Суперкористувачу (root) доступ надається завжди.

2.2.2. Режими іменування об’єктів System V IPC

Існують два режими іменування об’єктів System V IPC (черг повідомлень, сегментів спільної пам’яті та множин семафорів):

1) З використанням ключа IPC_PRIVATE, коли об’єкт IPC використовується лише батьківським процесом та породженими ним синівськими процесами. В цьому режимі об’єкт IPC спочатку створюється деяким процесом (системним викликом ***get з ключем IPC_PRIVATE), потім процес розпаралелюється (fork-exec), після чого об’єкт IPC є доступним для організації взаємодії між даним процесом та його синівськими процесами, яким відомий ідентифікатор цього об’єкту IPC (оскільки вони є клонами батьківського процесу). Можна провести певну аналогію між цим режимом іменування об’єктів IPC та схемою використанням неіменованого каналу (anonymous pipe).

2) З використанням ключа, згенерованого функцією ftok(), коли об’єкт IPC може бути використаний для організації взаємодії будь яких процесів, в тому числі тих що не пов’язані між собою в рамках відношення “батько-син” (тобто породжені різними батьківськими процесами). В цьому режимі об’єкт IPC створюється першим по часу викликом ***get (з вказаним ключем), а наступними по часу викликами ***get в інших процесах відбувається підключення (отримання доступу) до цього об’єкту IPC (з тим самим ключем). Можна провести певну аналогію між цим режимом іменування об’єктів IPC та схемою використанням іменованого каналу (named pipe).

Послідовність дій по створенню об’єкту IPC (рис.2.18) в першому режимі іменування передбачає використання лише системного виклику ***get(), а в другому режимі іменування - функції ftok() та системного виклику ***get(). В рамках другого режиму іменування у виклику ***get() можуть бути використані два варіанти прапорців (рис.2.19):

1) лише IPC_CREAT, тоді якщо це перший по часу виклик ***get(), то об’єкт IPC буде створено; якщо це наступний у часі виклик ***get(), то відбудеться підключення до існуючого об’єкту IPC (створеного першим викликом); в обох випадках повертається ідентифікатор об’єкту IPC для подальшої роботи з ним;

2) IPC_CREAT та IPC_EXCL, тоді якщо це перший по часу виклик ***get(), то об’єкт IPC буде створено (повертається його ідентифікатор); якщо це наступний у часі виклик ***get(), то він поверне помилку (без підключення до існуючого об’єкту IPC); цей варіант

використовується тоді, коли потрібно створити гарантовано новий об’єкт IPC, без підключення до вже існуючого об’єкту.

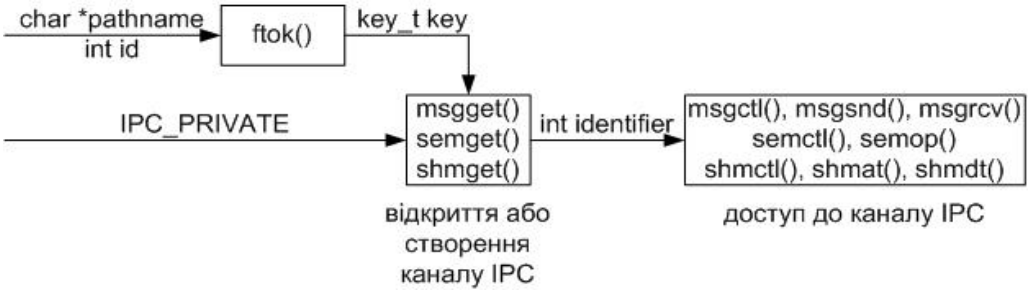

Рис.2.18. Схема створення об’єкту IPC.

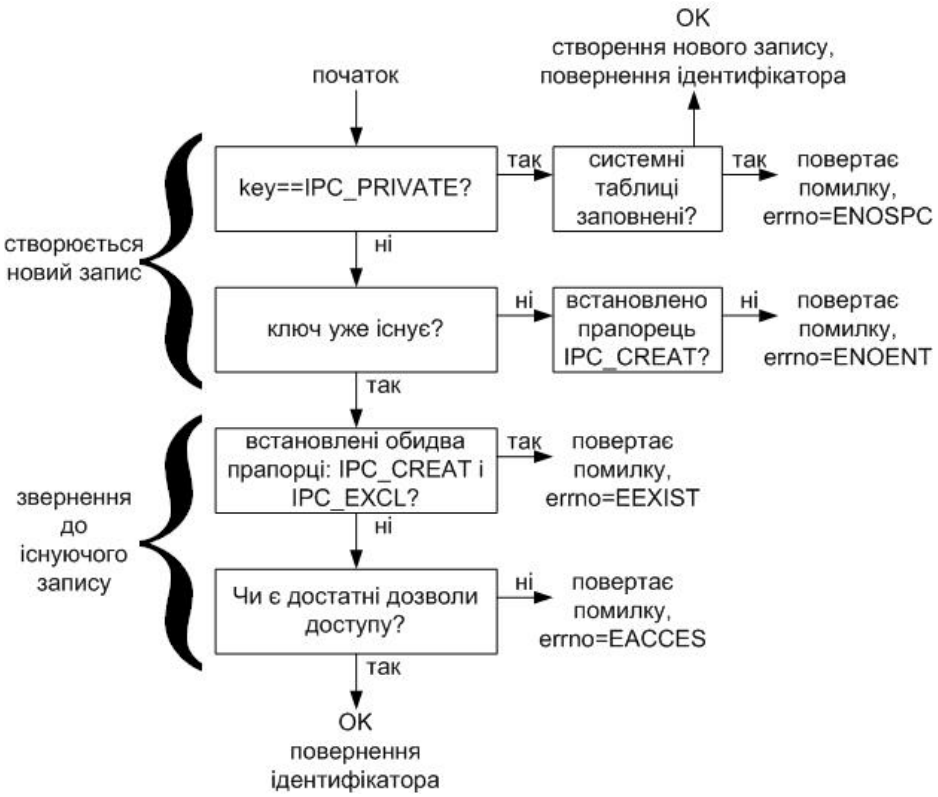

Рис.2.19. Послідовність дій по створенню об’єкта IPC та підключення до нього в різних режимах іменування.

Функція ftok() (рис.2.20) використовується для генерування ключа System V IPC. На вхід цій функції передаються два параметри: pathname - повний шлях до деякого існуючого файлу, proj_id - цілочисельний ідентифікатор. Для двох однакових значень pathname і proj_id виклик функції ftok() в різних процесах поверне однаковий ключ System V IPC. Якщо хоча б один з параметрів (pathname або proj_id) буде відрізнятись, то значення ключів будуть різними. Відтак параметр pathname грає роль “точки прив’язки” до одного і того самого об’єкту IPC. В якості значення цього параметру потрібно обирати шлях до такого файлу, який гарантовано буде існувати в системі (наприклад, для цього не підходять будь-які тимчасові файли, в тому числі ті, які створює сама програма). В якості параметру proj_id, як правило, вказують порядковий номер об’єкту IPC в межах програми (якщо їх використовується декілька), або просто 1, якщо такий об’єкт в програмі лише один.

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

Рис.2.20. Опис функції ftok().

В разі успішного спрацювання функція ftok() повертає ключ об’єкту IPC, інакше в разі помилки, функція повертає -1. Ключ об’єкту IPC, який повертає ftok() - це ціле число, утворене об’єднанням інформації про вказаний файл та молодших 8-ми бітів параметру proj_id. В багатьох реалізаціях функції ftok() інформація про файл складається з 1) інформації про файлову систему, в якій міститься файл з повним ім'ям pathname (поле st_dev структури stat); 2) номеру вузла (i-node) у файловій системі (поле st_ino структури stat). Отриманий від функції ftok() ключ передається на вхід системного виклику ***get().

2.2.3. Робота з об’єктами System V IPC з використанням команд ipcs та ipcrm

Для роботи з об’єктами System V IPC в командному рядку використовуються команди: ipcs, ipcrm та ipcmk.

Команда ipcs виводить інформацію про об’єкти IPC, до яких користувач (процес користувача) має доступ хоча б на читання. За замовчуванням команда ipcs виводить інформацію у вигляді списку об’єктів IPC усіх трьох типів (рис.2.21): черг повідомлень, сегментів спільної пам'яті та масивів семафорів.

```
[lcx@localhost MsgQueue]$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 262144     lcx        600        524288     2          dest
0x00000000 360449     lcx        600        4194304    2          dest
0x00000000 393218     lcx        600        524288     2          dest
0x00000000 491523     lcx        600        524288     2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Рис.2.21. Приклад виконання команди ipcs.

За допомогою команди ipcs можна отримати інформацію окремо про кожний тип об’єктів IPC (ключі -q, -m, -s) або докладну інформацію про окремий об’єкт із вказаним ідентифікатором. Наприклад, для отримання інформації про сегмент спільної пам’яті з id=3768321 потрібно виконати команду:

```
$ipcs -mi 3768321
Shared memory Segment shmid=3768321
uid=1000 gid=1000       cuid=1000      cgid=1000
mode=01600      access_perms=0600
bytes=36864     lpid=6028      cpid=10351     nattch=2
att_time=Mon Mar  8 20:17:07 2021
det_time=Mon Mar  8 20:17:07 2021
change_time=Mon Mar  8 18:17:16 2021
```

За допомогою команди ipcs також можна дізнатись: інформацію про те, хто створив та є власником об’єктів IPC (ключ -c); ідентифікатор процесу, який створив об’єктів IPC та останню операцію, яку він над ним виконав (ключ -p); час останньої операції над об’єктом IPC

чи операції по управлінню об’єктом (ключ -t); підсумкову інформацію про використання об’єктів IPC (ключ -u).

Команда ipcs також дозволяє дізнатись системні обмеження, яка накладені на механізми IPC в системі (ключ -l). Наприклад:

```
$ ipcs -l
------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

Команда ipcrm видаляє із системи об'єкти IPC System V та пов'язані з ними структури даних. Для того, щоб видалити такі об'єкти, користувач має бути або суперкористувачем (root), або творцем чи власником об'єкту IPC. Перед тим як видалити об’єкт IPC потрібно дізнатись його ключ або ідентифікатор (наприклад, за допомогою команди ipcs). Якщо в системі є процес, заблокований спробою виконання операції над знищеним об’єктом IPC, то він буде розблокований. Приклад видалення сегменту спільної пам’яті з id=3768321:

```
$ ipcrm -m 3768321
```

Створити новий об'єкт IPC у командному рядку можна за допомогою команди ipcmk.

2.3. Черги повідомлень (message queues)

2.3.1. Взаємодія процесів з використанням черги повідомлень

Черга повідомлень (message queue) - це механізм IPC, який дозволяє двом або більше процесам обмінюватись повідомленнями, кожному з яких призначається деяке ціле число (“тип”, “ідентифікатор”, “пріоритет” повідомлення). В процесі, який читає дані (отримує повідомлення) з черги, є можливість вказати “тип” (“ідентифікатор”) повідомлення, яке буде отримано. Повідомлення одного “типу” (з однаковим “ідентифікатором”) передаються за принципом FIFO. На відміну від каналів (неіменованих та іменованих) черга повідомлень дає можливість отримувати повідомлення різного “типу” в довільний послідовності.

Черга повідомлень створюється і використовується в одному з двох режимів іменування об’єктів IPC (з ключем IPC_PRIVATE або з ключем, згенерованим функцією ftok). Після створення в програму повертається ідентифікатор черги, який вказується в системних викликах по відправленню-отриманню повідомлень з черги та управління чергою. Після завершення роботи з чергою повідомлень її потрібно знищити (звільнити відповідний ресурс системи).

На відміну від неіменованих та іменованих каналів, дані через чергу передаються не у вигляді потоку байтів, а “порціями”, які називаються “повідомлення”. Окреме повідомлення складається з двох частин: 1) “тип” (“ідентифікатор”, “пріоритет”) у вигляді цілого числа; 2) дані (вміст повідомлення) у довільному вигляді. Загальна структура повідомлення задається структурою msgbuf (рис.2.22). Для використання черг повідомлень програміст має створити в програмі подібну структуру даних, перше поле якої обов’язково має бути long mtype (“тип” повідомлення) (рис.2.23).

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1];    /* message data */
};
```

Рис.2.22. Загальний варіант структури msgbuf.

```
#define MsgPLen       4
#define MsgDataCount  6

typedef struct {
  long type;
  char payload[MsgLen + 1];
  int data[MsgDataCount];
} queuedMessage;
```

Рис.2.23. Приклад опису структури msgbuf в програмі.

Для кожного примірника черги повідомлень (як об’єкта IPC) в ядрі зберігається структура msqid_ds (рис.2.24), яка містить всю необхідну інформацію про цей об’єкт IPC, в тому числі msg_perm - інформація про власника та права доступу до черги (struct ipc_perm), msg_stime - час здійснення останнього запису даних в чергу (відправлення повідомлення), msg_rtime - час здійснення останнього читання з черги (отримання повідомлення), msg_qnum - поточна кількість повідомлень в черзі, msg_qbytes - максимальна кількість байт, які можуть міститися у черзі та ін. Отримати інформацію, яка міститься у полях структури msqid_ds, можна за допомогою команди $ipcs -qi id=<ID> або за допомогою системного виклику msgctl().

```
struct msqid_ds {
    struct ipc_perm msg_perm;     /* Ownership and permissions */
```

```
            time_t          msg_stime;    /* Time of last msgsnd(2) */
            time_t          msg_rtime;    /* Time of last msgrcv(2) */
            time_t          msg_ctime;    /* Time of last change */
            unsigned long   __msg_cbytes; /* Current number of bytes in
                                             queue (nonstandard) */
            msgqnum_t       msg_qnum;     /* Current number of messages
                                             in queue */
            msglen_t        msg_qbytes;   /* Maximum number of bytes
                                             allowed in queue */
            pid_t           msg_lspid;    /* PID of last msgsnd(2) */
            pid_t           msg_lrpid;    /* PID of last msgrcv(2) */
        };
```

Рис.2.24. Структура msqid_ds (sys/msg.h).

2.3.2. Створення черги повідомлень

Для створення черги повідомлень використовується системний виклик msgget() (рис.2.25). Першим параметром key визначається режим іменування черги повідомлень (ключ IPC_PRIVATE або ключ, згенерований функцією ftok). Другим параметром msgflg вказуються потрібні прапорці (IPC_CREAT, IPC_EXCL) та права доступу до черги. В разі успішного спрацювання системний виклик msgget() повертає ідентифікатор черги повідомлень, асоційований із ключем key.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

Рис.2.25. Опис системного виклику msgget().

При виконанні системного виклику msgget() ядро ОС Linux або створює нову чергу повідомлень, розміщуючи її заголовок в таблиці черг повідомлень і повертаючи користувачеві дескриптор новоствореної черги, або знаходить елемент таблиці черг повідомлень, що містить зазначений ключ, і повертає відповідний дескриптор вже існуючої черги.

В режимі іменування з ключем IPC_PRIVATE (рис.2.26) чергою повідомлень можуть скористатись для обміну даними лише батьківський процес та його синівські процеси.

```
int mqueue;

mqueue = msgget(IPC_PRIVATE, 0666 | IPC_CREAT);

if(mqueue == -1){
   fprintf(stderr,"Can't associate the messages queue.\n");
   exit(1);
}
```

Рис.2.26. Приклад створення черги повідомлень з ключем IPC_PRIVATE.

В режимі іменування з ключем, згенерованим ftok() (рис.2.27), до черги повідомлень можуть підключитись процеси, породжені різними батьківськими процесами, з використанням “точки прив’язки” у вигляді шляху до деякого файлу та заданого цілого числа.

```
#define ProjectId 29
#define PathName  "queue.h"

  key_t key = ftok(PathName, ProjectId);
  if (key < 0) report_and_exit("couldn't get key...");

  int qid = msgget(key, 0666 | IPC_CREAT);
  if (qid < 0) report_and_exit("couldn't get queue id...");
```

Рис.2.27. Приклад створення (чи підключення до) черги повідомлень з ключем, згенерованим ftok().

2.3.3. Відправлення та отримання повідомлень з черги

Для відправлення повідомлень в чергу використовується системний виклик msgsnd() (рис.2.28). Перед тим, як відправляти повідомлення в чергу, потрібно заповнити поля відповідної структури (msgbuf), яка описує повідомлення (другий параметр - msgp). Першим параметром (msqid) вказується ідентифікатор черги, в яку відправляється повідомлення. Другим параметром (msgp) вказується адреса структури з повідомленням, що відправляється. В третьому параметрі (msgsz) вказується розмір повідомлення в байтах. В четвертому параметрі (msgflg) вказуються потрібні прапорці або 0, якщо msgsnd() викликається в звичайному режимі.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Рис.2.28. Опис системного виклику msgsnd().

В разі успішного виконання системного виклику msgsnd() повертається 0. У випадку помилки повертається -1, а змінній errno присвоюється код помилки. Приклад використання системного виклику msgsnd() наведено на рис.2.29.

За замовчуванням msgsnd() виконується в блокуючому (синхронному) режимі. Блокування процесу, що викликав msgsnd(), відбувається в ситуаціях, коли

1) в черзі недостатньо місця для запису msgsz байт (тобто додавання msgsz байт до вмісту черги перевищує максимально допустимий розмір черги (max size of queue), який зберігається у полі msg_qbytes структури msqid_ds);

2) додавання чергового повідомлення до черги призводить до того, що кількість повідомлень в черзі перевищує число, яке дорівнює максимально допустимому розміру черги (max size of queue - значення msqid_ds.msg_qbytes); це обмеження використовується для запобігання запису у чергу великої кількості повідомлень нульової довжини (оскільки на них все одно витрачається ресурс пам’яті ядра).

Для виконання msgsnd() в режимі без блокування (асинхронному режимі) потрібно в четвертому параметрі msgflg встановити прапорець IPC_NOWAIT. Тоді в разі успішного виконання виклик msgsnd() так само поверне 0, а в разі неможливості записати дані у чергу, виклик поверне -1 (помилка EAGAIN) без блокування викликаючого процесу.

```
  char* payloads[] = {"msg1", "msg2", "msg3", "msg4", "msg5", "msg6"};
  int types[] = {1, 1, 2, 2, 3, 3}; /* each must be > 0 */

  for (i = 0; i < MsgCount; i++) {
    /* build the message */
```

```
    queuedMessage msg;
    msg.type = types[i];
    strcpy(msg.payload, payloads[i]);

    /* send the message */
    msgsnd(qid, &msg, sizeof(msg), IPC_NOWAIT); /* don't block */
    printf("%s sent as type %i\n", msg.payload, (int) msg.type);
  }
```

Рис.2.29. Приклад використання системного виклику msgsnd().

Для отримання повідомлень з черги використовується системний виклик msgrcv() (рис.2.30). Першим параметром (msqid) вказується ідентифікатор черги, з якої отримується повідомлення. Другим параметром (msgp) вказується адреса структури, в якій потрібно розмістити отримане повідомлення. В третьому параметрі (msgsz) вказується розмір повідомлення в байтах. Четвертим параметром (msgtyp) вказується “тип” (“ідентифікатор”) повідомлення, яке потрібно отримати з черги. П’ятим параметром (msgflg) вказуються потрібні прапорці або 0, якщо msgrcv() викликається в звичайному режимі.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Рис.2.30. Опис системного виклику msgrcv().

Значення параметру msgtyp визначає яке повідомлення буде отримано з черги:

1) якщо msgtyp = 0, то буде отримано перше повідомлення в черзі (повідомлення, яке було відправлено в чергу раніше за всі інші повідомлення без врахування їх “типів”);

2) якщо msgtyp > 0, то буде отримано перше повідомлення “типу” msgtyp (якщо в параметрі msgflg встановлено прапорець MSG_EXCEPT, то буде отримано перше повідомлення, “тип” якого не дорівнює msgtyp);

3) якщо msgtyp < 0, то буде отримано перше повідомлення найменшого з “типів”, що не перевершують абсолютної величини msgtyp (наприклад, якщо msgtyp = -2, то буде отримано перше повідомлення, “тип” якого дорівнює 1, або, якщо таких повідомлень в черзі немає, то перше повідомлення, “тип” якого дорівнює 2).

Режим роботи msgrcv() по замовченню синхронний (з блокуванням). Процес, що викликає msgrcv(), блокується тоді, коли в черзі немає повідомлення вказаного “типу” msgtyp, і очікує на появу такого повідомлення в черзі. Щоб виконати msgrcv() в асинхронному режимі (без блокування) потрібно встановити прапорець IPC_NOWAIT в параметрі msgflg. Тоді, якщо повідомлення “типу” msgtyp є в черзі, то воно буде отримано, а якщо його нема, то msgrcv() поверне -1 (помилка ENOMSG) без блокування викликаючого процесу.

По замовченню, якщо розмір отриманого повідомлення більший ніж задано параметром msgsz, то виклик msgrcv() завершується, повертаючи -1 (помилка E2BIG), а повідомлення лишається в черзі. Якщо ж в параметрі msgflg встановити прапорець MSG_NOERROR, то повідомлення буде обрізане (truncated) до довжини msgsz, а повідомлення видалено з черги. При цьому виклик msgrcv() завершується успішно, а відрізаний кінець повідомлення втрачається.

При успішному завершенні системного виклику msgrcv() повертається кількість байт, які були зчитані з черги у поле mtext структури msgbuf (або у всі поля даних, якщо ця структура має інший вигляд). У випадку помилки msgrcv() повертає -1. Приклад використання системного виклику msgrcv() наведено на рис.2.31.

```
int types[] = {3, 1, 2, 1, 3, 2};

for (i = 0; i < MsgCount; i++) {
  queuedMessage msg; /* defined in queue.h */
  if (msgrcv(qid, &msg, sizeof(msg), types[i], MSG_NOERROR | IPC_NOWAIT) < 0)
    puts("msgrcv trouble...");
  printf("%s received as type %i\n", msg.payload, (int) msg.type);
}
```

Рис.2.31. Приклад використання системного виклику msgrcv().

2.3.4. Управління чергою повідомлень

Для виконання команд по управлінню чергою повідомлень використовується системний виклик msgctl() (рис.2.32). Першим параметром (msqid) вказується ідентифікатор черги, над якою виконується команда. Другим параметром (cmd) вказується команда по управлінню чергою (IPC_STAT, IPC_SET, IPC_RMID, IPC_INFO, MSG_INFO, MSG_STAT). Третім параметром вказується адреса структури типу msqid_ds або структури типу msginfo, якщо це потрібно для виконання команди (як свого роду аргумент команди).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Рис.2.32. Опис системного виклику msgctl().

Над чергою повідомлень можна виконати наступні команди:

1) IPC_STAT - скопіювати поточне значення кожного поля структури даних (msqid_ds), асоційованої з ідентифікатором черги повідомлень msqid, у структуру, на яку вказує buf (в даному випадку buf має бути структурою типу msqid_ds);

2) IPC_SET - у структурі даних (msqid_ds), асоційованій з ідентифікатором msqid, змінити значення діючих ідентифікаторів користувача (msg_perm.uid) і групи (msg_perm.gid), прав доступу (msg_perm.mode) та максимально припустимої кількості байт у черзі (msg_qbytes) на ті значення, що вказані у структурі buf (в даному випадку buf має бути структурою типу msqid_ds);

3) IPC_RMID - видалити з системи чергу повідомлень з ідентифікатором msqid та асоційовану з нею структуру даних (msqid_ds); виконання всіх процесів, заблокованих на операціях msgsnd/msgrcv з цією чергою, розблоковується;

4) IPC_INFO - повертає системні обмеження на використання черг та загальні параметри черг у структуру, на яку вказує buf (в даному випадку buf має бути структурою типу msginfo) (рис.2.33).

```
struct msginfo {
    int msgpool; /* Size in kibibytes of buffer pool
                    used to hold message data;
                    unused within kernel */
    int msgmap;  /* Maximum number of entries in message
                    map; unused within kernel */
    int msgmax;  /* Maximum number of bytes that can be
                    written in a single message */
    int msgmnb;  /* Maximum number of bytes that can be
                    written to queue; used to initialize
                    msg_qbytes during queue creation
                    (msgget(2)) */
```

```
                int msgmni;  /* Maximum number of message queues */
                int msgssz;  /* Message segment size;
                                unused within kernel */
                int msgtql;  /* Maximum number of messages on all queues
                                in system; unused within kernel */
                unsigned short int msgseg;
                             /* Maximum number of segments;
                                unused within kernel */
            };
```

Рис.2.33. Структура msginfo (sys/msg.h).

Щоб виконати команди IPC_SET чи IPC_RMID, процес повинен мати ідентифікатор користувача, рівний ідентифікатору творця чи власника черги (або ідентифікатору суперкористувача). Щоб виконати команду IPC_STAT, потрібне хоча б право на читання. При успішному виконанні (команди IPC_STAT, IPC_SET, IPC_RMID) системний виклик msgctl() повертає 0. У випадку помилки msgctl() повертає -1. Приклад використання системного виклику msgctl() наведено на рис.2.34.

```
/** remove the queue **/
if (msgctl(qid, IPC_RMID, NULL) < 0)  /* NULL = 'no flags' */
  report_and_exit("trouble removing queue...");
```

Рис.2.34. Приклад використання системного виклику msgctl().

2.4. Спільна пам’ять (shared memory)

2.4.1. Взаємодія процесів з використанням спільної пам’яті

Спільна пам’ять (shared memory) – це механізм IPC (об’єкт System V IPC), який дозволяє двом або більше процесам мати спільні області віртуальної пам'яті і, як наслідок, обмінюватись даними, що містяться в них. Одиницями спільної пам'яті є сегменти, властивості яких залежать від особливостей апаратних засобів керування пам'яттю. Використання сегменту спільної пам’яті забезпечує найбільш швидкий обмін даними між процесами у порівнянні з усіма іншими об’єктами IPC.

Робота зі спільною пам'яттю починається з того, що процес за допомогою системного виклику shmget() створює спільний сегмент з унікальним ідентифікатором і асоційовану з ним структуру даних. Унікальний ідентифікатор називається ідентифікатором спільної пам‘яті (shmid). Ідентифікатор спільної пам‘яті асоціюється зі структурою даних shmid_ds (рис.2.35), в якій поле shm_perm в свою чергу є структурою типу ipc_perm (рис.2.36).

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

Рис.2.35. Структура shmid_ds (sys/shm.h).

```
struct ipc_perm {
    key_t          __key;    /* Key supplied to shmget(2) */
    uid_t          uid;      /* Effective UID of owner */
    gid_t          gid;      /* Effective GID of owner */
    uid_t          cuid;     /* Effective UID of creator */
    gid_t          cgid;     /* Effective GID of creator */
    unsigned short mode;     /* Permissions + SHM_DEST and
                                SHM_LOCKED flags */
    unsigned short __seq;    /* Sequence number */
};
```

Рис.2.36. Структура ipc_perm.

Щоб одержати доступ до спільного сегмента, його потрібно приєднати за допомогою системного виклику shmat(), що розміщує сегмент у віртуальному просторі процесу. Після приєднання, відповідно до прав доступу, процеси можуть читати дані із сегмента і записувати їх (рис.2.37).

Коли сегмент спільної пам’яті стає непотрібним, його треба від’єднати, скориставшись системним викликом shmdt(). Для виконання керуючих дій над пам’яттю, що розділяється служить системний виклик shmctl(). Після того, як останній процес від’єднав сегмент спільної пам’яті, потрібно виконати керуючу дію по видаленню сегмента із системи (команда IPC_RMID).

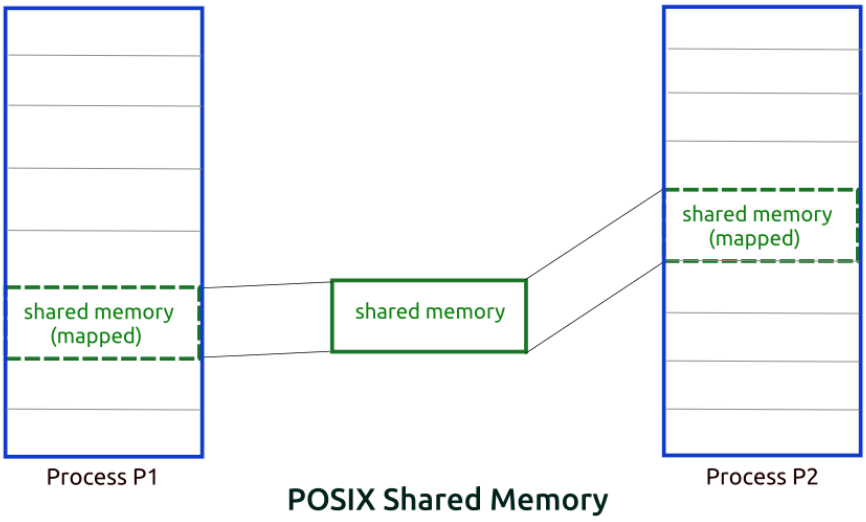

Рис.2.37. Схема використання сегменту спільної пам’яті двома процесами.

2.4.2. Створення сегменту спільної пам’яті

Для створення сегменту спільної пам’яті використовується системний виклик shmget() (рис.2.38). Першим параметром key визначається режим іменування сегменту спільної пам’яті (ключ IPC_PRIVATE або заданий ключ key, згенерований функцією ftok). Другим параметром size вказується розмір сегменту спільної пам’яті в байтах. Третім параметром msgflg вказуються потрібні прапорці (IPC_CREAT, IPC_EXCL) та права доступу до сегменту спільної пам’яті.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

Рис.2.38. Опис системного виклику shmget().

У разі створення сегменту спільної пам’яті, система ініціалізує його вміст нульовими значеннями. В разі успішного спрацювання системний виклик shmget() повертає ідентифікатор сегменту спільної пам’яті, асоційований із ключем key. Інакше у випадку помилки shmget() повертає -1. Приклад використання системного виклику shmget() наведено на рис.2.39.

```
int shared;

if((shared = shmget(IPC_PRIVATE, 4096, 0xfff)) == -1) {
   fprintf(stderr,"Can't allocate the shared memory segment.\n");
   exit(1);
}
```

Рис.2.39. Приклад створення сегменту спільної пам’яті з ключем IPC_PRIVATE.

2.4.3. Підключення та відключення сегменту спільної пам’яті

Для приєднання сегменту спільної пам’яті, асоційованого з ідентифікатором shmid, до сегменту даних процесу використовується системний виклик shmat() (рис.2.40). Параметр

shmaddr містить адресу, за якою потрібно підключити сегмент спільної пам’яті. Якщо значення аргументу shmaddr дорівнює нулю, то сегмент приєднується за адресою, обраною системою.

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Рис.2.40. Опис системного виклику shmat().

Параметр shmflg використовується для передачі системному виклику shmat() прапорців SHM_RND і SHM_RDONLY. Наявність першого з них означає, що адресу shmaddr треба заокруглити до деякої системно-залежної величини. Другий прапорець наказує приєднати сегмент тільки для читання. Якщо він не встановлений, приєднаний сегмент буде доступний і на читання, і на запис (якщо процес має відповідні права). При успішному завершенні системного виклику shmat() повертається початкова адреса приєднаного сегмента. У випадку помилки повертається -1.

Системний виклик shmdt() (рис.2.41) від'єднує сегмент спільної пам'яті, розташований за адресою shmaddr, від сегмента даних процесу. При успішному завершенні системного виклику shmdt результат дорівнює 0. У випадку помилки повертається -1.

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

Рис.2.41. Опис системного виклику shmdt().

2.4.4. Управління сегментом спільної пам’яті

Для виконання команд по управлінню сегментом спільної пам’яті використовується системний виклик shmctl() (рис.2.42). Першим параметром (shmid) вказується ідентифікатор сегменту спільної пам’яті, над яким виконується команда. Другим параметром (cmd) вказується команда по управлінню сегментом спільної пам’яті (IPC_STAT, IPC_SET, IPC_RMID, IPC_INFO, SHM_INFO, SHM_STAT, SHM_LOCK, SHM_UNLOCK). Третім параметром вказується адреса структури типу shmid_ds або структури типу shminfo, якщо це потрібно для виконання команди (як свого роду аргумент команди cmd).

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Рис.2.42. Опис системного виклику shmctl().

Над сегментом спільної пам’яті можна виконати наступні команди:

1) **IPC_STAT** - скопіювати поточне значення кожного поля структури даних (shmid_ds), асоційованої з ідентифікатором сегменту спільної пам’яті shmid, у структуру, на яку вказує buf (в даному випадку buf має бути структурою типу shmid_ds);

2) **IPC_SET** - у структурі даних (shmid_ds), асоційованій з ідентифікатором shmid, змінити значення діючих ідентифікаторів користувача (shm_perm.uid) і групи (shm_perm.gid), прав доступу (shm_perm.mode) на ті значення, що вказані у структурі buf (в даному випадку buf має бути структурою типу shmid_ds);

3) **IPC_RMID** - видалити з системи сегмент спільної пам’яті з ідентифікатором shmid та асоційовану з ним структуру даних (shmid_ds); сегмент буде видалено лише після того, як всі процеси від’єднають його від своїх сегментів даних (аргумент buf в даному випадку ігнорується);

4) **IPC_INFO** - повертає системні обмеження на використання сегментів спільної пам’яті у структуру, на яку вказує buf (в даному випадку buf має бути структурою типу shminfo) (рис.2.43);

5) **SHM_INFO** - повертає у buf структуру shm_info (рис.2.44), поля якої містять інформацію про споживання системних ресурсів сегментом спільної пам’яті з ідентифікатором shmid;

6) **SHM_STAT** - зробити те саме, що робить IPC_STAT, тільки параметром shmid задається не ідентифікатор сегменту, а індекс у внутрішньому масиві ядра, в якому зберігається інформація про всі сегменти спільної пам'яті в системі;

7) **SHM_LOCK** - заборонити операцію свопінгу для сегменту спільної пам’яті з ідентифікатором shmid;

8) **SHM_UNLOCK** - дозволити операцію свопінгу для сегменту спільної пам’яті з ідентифікатором shmid.

```
struct shminfo {
    unsigned long shmmax; /* Maximum segment size */
    unsigned long shmmin; /* Minimum segment size;
                             always 1 */
    unsigned long shmmni; /* Maximum number of segments */
    unsigned long shmseg; /* Maximum number of segments
                             that a process can attach;
                             unused within kernel */
    unsigned long shmall; /* Maximum number of pages of
                             shared memory, system-wide */
};
```

Рис.2.43. Структура shminfo (sys/shm.h).

```
struct shm_info {
    int           used_ids; /* # of currently existing
                               segments */
    unsigned long shm_tot;  /* Total number of shared
                               memory pages */
    unsigned long shm_rss;  /* # of resident shared
                               memory pages */
    unsigned long shm_swp;  /* # of swapped shared
                               memory pages */
    unsigned long swap_attempts;
                            /* Unused since Linux 2.4 */
    unsigned long swap_successes;
                            /* Unused since Linux 2.4 */
};
```

Рис.2.44. Структура shm_info (sys/shm.h).

Щоб виконати керуючі дії IPC_SET, IPC_RMID, SHM_LOCK та SHM_UNLOCK процес повинен мати діючий ідентифікатор користувача, рівний ідентифікаторам творця чи власника сегмента спільної пам‘яті, або ідентифікатору суперкористувача. Для виконання керуючої дії IPC_STAT процесу потрібно право на читання. При успішному завершенні shmctl() повертає 0. У випадку помилки shmctl() повертає -1.

2.4.5. Відображення файлів у пам’ять

В багатьох випадках роботу з файлом зручно організувати у режимі відображення файлу в пам’ять (рис.2.45). Для цього використовуються функції: mmap() - відобразити вказаний файл у адресний простір процесу, munmap() - відключити відображення файлу в адресний простір процесу, msync() - синхронізувати вміст пам’яті і файлу (зберегти дані з пам’яті у файлі). Цей механізм також часто використовують для організації взаємодії двох чи більше процесів за принципом спільної пам’яті, в ролі якої виступає файл, одночасно відображений в адресний простір цих процесів. В деяких джерелах такий спосіб використання механізму спільної пам’яті називають більш сучасним ніж спільна пам’ять - об’єкт System V IPC (shmget, shmat, shmdt, shmctl).

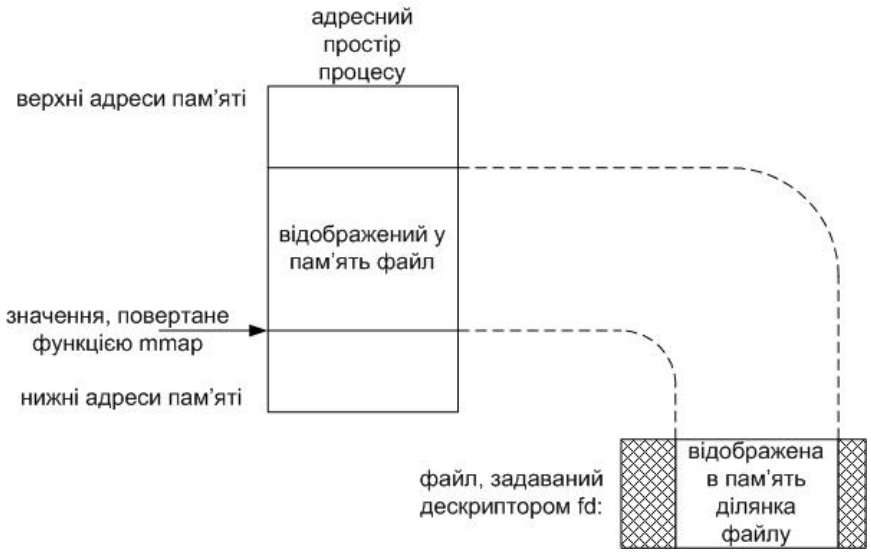

Рис.2.45. Схема відображення файлу в пам’ять.

Функція mmap() (рис.2.46) відображає файл в адресний простір процесу. В параметрі addr можна вказати початкову адресу ділянки пам'яті процесу, в яку потрібно відобразити вміст дескриптора fd. Зазвичай цьому параметру присвоюють значення нульового вказівника, що вказує ядру на необхідність вибрати початкову адресу самостійно. В будь-якому випадку функція mmap() повертає початкову адресу ділянки пам'яті, виділеного для відображення файлу. Параметр length задає довжину ділянки пам’яті в байтах. Ділянка пам'яті може починатися не з початку файлу, а з деякого місця, що задається параметром offset (як правило вказують offset=0).

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Рис.2.46. Опис функції mmap().

Режим захисту ділянки пам'яті з відображеним файлом задається параметром prot, який може бути комбінацією з наступних констант: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE. Параметр flags може приймати значення: MAP_SHARED – зміни передаються іншим процесам, MAP_PRIVATE – зміни не передаються іншим процесам і не впливають на відображений файл, MAP_FIXED – параметр addr інтерпретується як точне значення адреси пам’яті. У параметрі flags можна вказати тільки один з прапорців – MAP_SHARED або MAP_PRIVATE, додавши до нього за необхідності MAP_FIXED. Якщо вказано прапорець MAP_PRIVATE, всі зміни будуть відбуватися тільки з образом файлу в адресному просторі

процесу; іншим процесам вони доступні не будуть. Якщо вказано прапорець MAP_SHARED, зміни у відображених даних будуть бачити всі процеси, що спільно використовують файл.

Для відключення відображення файлу в адресний простір процесу використовують функцію munmap() (рис.2.47). Параметр addr повинен містити адресу, яку повернула відповідна функція mmap(), а length – довжину ділянки пам’яті. Після виклику munmap() будь-які спроби звернутися до цієї ділянки пам'яті призведуть до надсилання процесу сигналу SIGSEGV (передбачається, що ця ділянка пам'яті не буде знову відображена викликом mmap()). Якщо файл був відображений з прапорцем MAP_PRIVATE, всі внесені за час роботи процесу зміни у відображену ділянку пам’яті втрачаються.

```
#include <sys/mman.h>

int munmap(void *addr, size_t length);
```

Рис.2.47. Опис функції munmap().

Якщо ми змінюємо вміст ділянки пам'яті, у яку відображений файл, через деякий час вміст файлу буде відповідним чином змінений ядром. Однак у деяких випадках потрібно, щоб вміст файлу завжди був у відповідності із вмістом пам'яті. Тоді для здійснення миттєвої синхронізації використовується функція msync() (рис.2.48). В параметрі flags вказуються прапорці: MS_ASYNC - здійснити асинхронний запис, MS_SYNC - здійснити синхронний запис, MS_INVALIDATE - скинути кеш. З двох прапорців MS_ASYNC і MS_SYNC можна вказати лише один. Відмінність між ними полягає в тому, що повернення з функції в разі прапорця MS_ASYNC відбувається відразу ж, як тільки дані для запису у файл будуть розміщені в черзі ядром, а в разі прапорця MS_SYNC повернення з функції відбувається тільки після завершення операцій запису у файл. Якщо додатково вказано прапорець MS_INVALIDATE, всі копії файлу, вміст яких не збігається з його поточним вмістом, вважаються застарілими. Наступні звернення до цих копій призведуть до зчитування даних з файлу.

```
#include <sys/mman.h>

int msync(void *addr, size_t length, int flags);
```

Рис.2.48. Опис функції msync().

2.4.6. Взаємодія процесів за допомогою файлів відображених у пам’ять

Взаємодія процесів за допомогою файлів відображених у пам’ять (mmap) за принципом спільної пам’яті може відбуватись у двох режимах іменування:

1) З використанням існуючого файлу, ім’я якого грає роль «точки підключення». В цьому режимі кожний процес спочатку відкриває цей файл викликом open(), після чого виконує відображення цього файлу в свій адресний простір (mmap). Далі процеси взаємодіють з іншими процесами через відповідну ділянку спільної пам’яті. В цьому режимі перед використанням ділянки спільної пам’яті її, як правило, потрібно заповнити нульовими значеннями.

2) В анонімному режимі без існуючого файлу або з використанням спеціального файлу /dev/zero. В цьому режимі батьківський процес відображає «неіснуючий файл» (або файл /dev/zero) у свій адресний простір, породжує синівський процес (fork), який успадковує в нього ділянку спільної пам’яті. Далі батьківський та синівський процеси взаємодіють через цю ділянку спільної пам’яті.

В анонімному (неіменованому) режимі відображення файлу в пам'ять відсутня необхідність створювати або відкривати файл (рис.2.49). Замість цього вказуються прапорці

MAP_SHARED | MAP_ANON, а в якості дескриптора файлу передається -1. Зсув, що задається аргументом offset, ігнорується. Створена ділянка спільної пам'яті автоматично заповнюється нулями.

```
ptr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
           MAP_SHARED | MAP_ANON, -1, 0);
```

Рис.2.49. Відображення файлу у пам’ять в анонімному (неіменованому) режимі.

Схожого результату можна досягнути шляхом використання спеціального файлу /dev/zero, який необхідно відкрити в батьківському процесі, після чого передати отриманий дескриптор функції mmap (рис.2.50). Цей файл, що є символьним пристроєм, повертає нулі в разі спроби читання з нього, а весь вивід в нього йде «в нікуди». Відтак виділена ділянка спільної пам'яті буде гарантовано заповнена нулями.

```
fd = open(“/dev/zero”, O_RDWR);
ptr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
close(fd);
```

Рис.2.50. Відображення файлу у пам’ять з використанням /dev/zero.

2.4.7. Спільне використання файлів паралельними процесами

Взаємодію обчислювальних процесів за принципом спільної пам’яті (shared memory) можна також організувати шляхом спільного використання файлу, який одночасно відкритий цими процесами (без відображення файлу у пам’ять). Недоліком такого підходу є менша швидкість здійснення операцій читання-запису у порівнянні з спільною пам’яттю System V IPC та відображенням файлу у пам’ять (mmap). Відносною перевагою такого підходу є можливість синхронізувати спільний доступ процесів до даних у файлі за допомогою функцій fcntl() та lockf(). Особливістю цього підходу є те, що дані, якими обмінюються через файл процеси, залишаються в цьому файлі після завершення процесів (такого ж результату можна досягнути використовуючи mmap для існуючого файлу).

В рамках цього підходу зустрічається ситуація одночасного доступу процесів до одних і тих самих даних у файлі. Відповідно потрібно використовувати механізм синхронізації процесів, що дозволяє не пускати інший процес (процеси) читати і/або записувати дані в заданій ділянці файлу (fcntl(), lockf()). При цьому блокування файлу і його ділянок за допомогою fcntl() та lockf() носять в ОС Linux необов'язковий характер. Тобто, програма, яка не використовує виклики синхронізації (fcntl() або lockf()), матиме доступ до даних без будь-яких обмежень.

Першим способом обмеження доступу процесів до ділянок файлу «по домовленості» є використання функції fcntl() (рис.2.51), яка виконує команди управління (cmd) над заданим дескриптором файлу fd. Зокрема для синхронізації доступу процесів до файлу використовуються команди F_SETLK, F_SETLKW та F_GETLK, аргументом яких є змінна типу flock (рис.2.52) - структура контролю за блокуванням ділянки файлу.

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

Рис.2.51. Опис функції fcntl().

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

Рис.2.52. Структура flock.

Команди синхронізації доступу процесів до файлу з використанням fcntl():

1) **F_SETLK** – встановлює або знімає замок, заданий структурою flock. Виклик fcntl() з командою F_SETLK є не блокуючим, тобто у разі встановити замок на ділянку файлу, яка вже знаходиться під замком (повністю чи частково), то виклик fcntl() з командою F_SETLK відразу завершиться, і fcntl() поверне -1. Поле структури l_type визначає тип блокування доступу:

F_RDLCK – на читання,

F_WRLCK – на запис,

F_UNLCK – зняти усі замки.

Поля l_whence, l_start, l_len задають ділянку файлу, на яку ставиться замок: від позиції у файлі lseek (fd, l_start, l_whence), довжиною l_len байтів. Поле l_whence може приймати значення: SEEK_SET, SEEK_CUR, SEEK_END. Якщо поле l_len рівне нулю, то це означає “до кінця файлу”. Якщо всі три поля (l_whence, l_start, l_len) дорівнюють 0, то буде заблокований весь файл.

2) **F_SETLKW –** встановлює або знімає замок, заданий структурою flock. При цьому, якщо замок на ділянку файлу, що перетинається з вказаною ділянкою, вже кимось встановлено, то спершу дочекатися зняття цього замку. Тобто виклик fcntl() з командою F_SETLKW є блокуючим.

3) **F_GETLK** – запитує можливість встановити замок, описаний структурою flock. Можливі два варіанти:

3.1) Якщо є можливість встановити такий замок (не зачинено ніким), то в структурі flock поле l_type стає рівним F_UNLCK, а поле l_whence рівним SEEK_SET.

3.2) Якщо замок вже кимось встановлено (і виклик F_SETLKW заблокував би процес), ми отримуємо інформацію про чужий замок в структуру flock. При цьому в поле l_pid заноситься ідентифікатор процесу, який створив цей замок.

Замки автоматично знімаються при закритті дескриптора файлу. Замки не успадковуються породженим процесом при виклику fork(). Приклад використання функції fcntl() з командою F_SETLKW наведено на рис.2.53.

```
int fd;
struct flock lock;
            ...
/* Open a file descriptor to the file. */
fd = open (file, O_WRONLY);
printf ("locking\n");
/* Initialize the flock structure. */
memset (&lock, 0, sizeof(lock));
lock.l_type = F_WRLCK;
/* Place a write lock on the file. */
fcntl (fd, F_SETLKW, &lock);
            ...
/* Release the lock. */
lock.l_type = F_UNLCK;
```

```
fcntl (fd, F_SETLKW, &lock);
close (fd);
```

Рис.2.53. Приклад використання функції fcntl().

Другим способом обмеження доступу процесів до ділянок файлу «по домовленості» є використання функції lockf() (рис.2.54), яка виконує вказану команду синхронізації (F_LOCK, F_TLOCK, F_ULOCK, F_TEST) над дескриптором файлу fd (ділянка файлу визначається біжучою позицією у файлі і параметром len).

```
#include <unistd.h>

int lockf(int fd, int cmd, off_t len);
```

Рис.2.54. Опис функції lockf().

Команди синхронізації доступу процесів до файлу з використанням lockf():

1) **F_ULOCK** - розблокувати вказану ділянку файлу (зняти замок).

2) **F_LOCK** - встановити замок. Якщо є чужий замок (встановлений перед цим), то заблокувати викликаючий процес до моменту, коли чужий замок буде знято. Замок встановлюється на ділянку від поточної позиції у файлі плюс len байт (pos...pos+len-1). Якщо len має від'ємне значення, то мінус len байт (pos-len...pos-1). Якщо len=0, то від поточної позиції до кінця файлу. Замки, встановлені процесом автоматично знімаються після його завершення. Замки не успадковуються синівським процесом.

3) **F_TLOCK** - встановити замок. Якщо є чужий замок (встановлений перед цим), то завершити виклик функції без блокування (функція повертає -1).

4) **F_TEST** - перевірити наявність замку. Функція повертає 0, якщо замку немає, в іншому випадку функція повертає -1 (замкнено), а змінна errno приймає значення EAGAIN.

2.5. Семафори (semaphores)

2.5.1. Використання семафорів в ОС Linux

Семафор - це засіб синхронізації паралельних процесів або програмних потоків, в основі якого лежить лічильник, над яким можна виконувати дві атомарні операції: збільшення і зменшення значення лічильнику на одиницю. При цьому операція зменшення над лічильником з нульовим значенням є блокуючою (рис.2.55). Поняття семафору запропонував у 1965 році нідерландський вчений Едсгер Дейкстра (Edsger Dijkstra, 1930-2002). Програмна реалізація семафору складається з лічильнику (значення семафору) та черги процесів/потоків, заблокованих на семафорі.

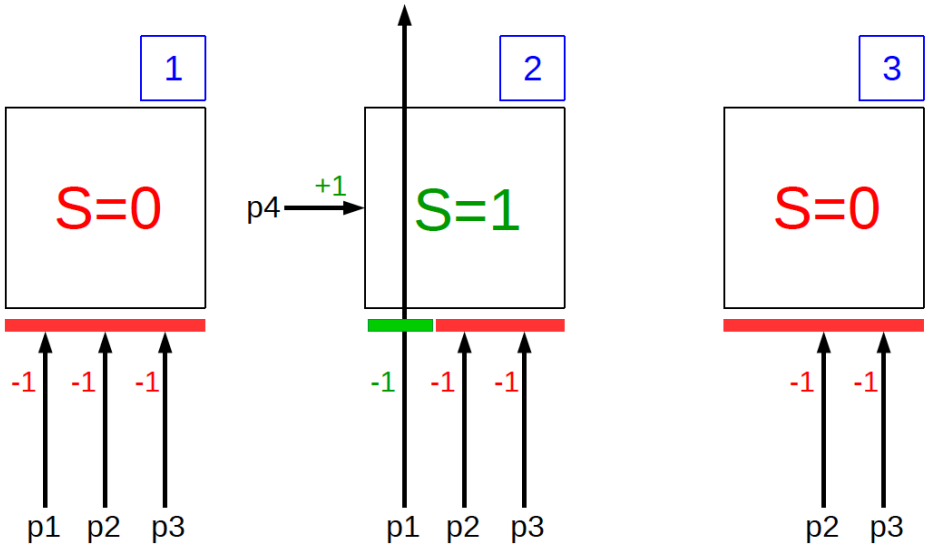

Рис.2.55. Схема використання семафору для синхронізації паралельних процесів.

В ОС Linux значення семафора – це ціле число в діапазоні від 0 до 32767. Оскільки в багатьох програмах потрібно більше одного семафора, ОС Linux надає доступ до цього механізму синхронізації (об’єкту IPC System V IPC) у вигляді множини семафорів (semaphore set). Максимальний розмір множини семафорів обмежений системним параметром SEMMSL. Кількість семафорів у множині визначає програміст в момент її створення.

Перед тим як використовувати семафори (виконувати операції чи команди управління), потрібно створити множину семафорів за допомогою системного виклику semget(), який повертає унікальний ідентифікатор множини семафорів. Ідентифікатор множини семафорів асоційований зі структурою даних semid_ds (рис.2.56) і використовується для звертання до множини семафорів при виконанні операцій чи команд управління (semop(), semtimedop(), semctl()).

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* Ownership and permissions */
    time_t          sem_otime; /* Last semop time */
    time_t          sem_ctime; /* Last change time */
    unsigned long   sem_nsems; /* No. of semaphores in set */
};
```

Рис.2.56. Структура semid_ds (sys/sem.h).

В структурі semid_ds (рис.2.56) асоційованій з множиною семафорів міститься наступна інформація: sem_perm – інформація про власника та права доступу до множини семафорів; sem_otime – час здійснення останньої операції над множиною семафорів; sem_ctime – час останньої зміни множини семафорів; sem_nsems – кількість семафорів у множині.

Множину семафорів можна розглядати як масив структур sem (рис.2.57), кожна з яких відповідає окремому семафору. З кожним семафором у множині семафорів асоціюються наступні значення: semval – поточне значення семафору; semncnt – кількість процесів, що очікують на збільшення значення семафору; semzcnt – кількість процесів, що очікують на нульове значення семафору; sempid – ідентифікатор процесу, який останнім виконав операцію над семафором. Крім цього для кожного процесу та кожного семафору, який він використовує, зберігається змінна semadj, яка застосовується для коригування значення семафору в разі несподіваного (аварійного) завершення процесу.

```
struct sem {
        unsigned short  semval;   /* semaphore value */
        unsigned short  semzcnt;  /* # waiting for zero */
        unsigned short  semncnt;  /* # waiting for increase */
        pid_t           sempid;   /* Process ID that did last operation */
};
```

Рис.2.57. Значення, які асоціюються з кожним семафором у множині семафорів.

Процес, що першим серед інших виконав системний виклик semget() з ключем key (в режимі іменування із заданим ключем), стає власником/творцем множини семафорів. Він визначає, скільки буде семафорів у множині та специфікує права на виконання операцій над множиною семафорів для всіх процесів, включаючи себе. Даний процес може в подальшому поступитися правом власності іншому процесу або змінити права на операції за допомогою системного виклику semctl(), призначеного для управління семафорами. Процеси, які виконують системний виклик semget() з ключем key після даного процесу, отримують доступ до відповідної множини семафорів (у вигляді її ідентифікатору).

Над кожним семафором у множині за допомогою системного виклику semop() можна виконати одну з трьох операцій:

1) збільшити значення на задане число,

2) зменшити значення на задане число,

3) дочекатися, коли значення буде рівне нулю.

Системний виклик semop() виконує операції над заданим набором семафорів у множині (наприклад, лише над одним семафором, або над першим і третім семафором, або над усіма семафорами у множині). Тобто надається можливість за один виклик semop() виконати операції над кількома семафорами у множині. Для кожного семафору задається своя операція за допомогою масиву операцій. Ядро ОС Linux виконує операції з масиву по черзі. Якщо чергова операція не може бути виконана, то ефект попередніх операцій анулюється.

На множини семафорів в ОС Linux накладаються наступні ресурсні обмеження:

1) SEMMNI – загальносистемне обмеження на кількість множин семафорів. У системах Linux до версії 3.19 значення за замовчуванням для цього обмеження становило 128. Починаючи з версії ядра Linux 3.19 значення за замовчуванням – 32000. У ОС Linux це обмеження можна прочитати та змінити у четвертому полі параметру ядра /proc/sys/kernel/sem.

2) SEMMSL – максимальна кількість семафорів у одній множині. В системах Linux до версії 3.19 значенням за замовчуванням було 250. Починаючи з версії ядра Linux 3.19 значення за замовчуванням - 32000. У ОС Linux це обмеження можна прочитати та змінити у першому полі параметру ядра /proc/sys/kernel/sem.

3) SEMMNS – загальносистемне обмеження на кількість окремих семафорів. У ОС Linux це обмеження можна прочитати та змінити у другому полі параметру ядра /proc/sys/kernel/sem. Загальна кількість семафорів у системі також обмежується добутком SEMMSL та SEMMNI.

Приклад визначення ресурсних обмежень, які накладаються в ОС Linux на семафори:

```
$ cat /proc/sys/kernel/sem
32000   1024000000  500   32000
```

Отримане значення параметру ядра /proc/sys/kernel/sem означає наступне:

```
SEMMNI = 32000
```

```
SEMMNS = 1024000000
SEMOPM = 500
SEMMSL = 32000
```

Параметр SEMOPM визначає максимальну кількість операцій над семафорами, які можна виконати одним системним викликом semop().

Інший спосіб визначення ресурсних обмежень, які накладаються в ОС Linux на семафори, – це скористатись командою ipcs, наприклад:

```
$ ipcs -ls
------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

Змінити значення параметрів, що задають ресурсні обмеження на використання семафорів, можна в наступний спосіб:

```
$ echo 250 32000 100 128 > /proc/sys/kernel/sem
```

Для цього також можна скористатись командою sysctl:

```
$ sysctl -w kernel.sem="250 32000 100 128"
```

Для того, щоб задані значення ресурсних обмежень на використання семафорів встановлювались кожного разу при запуску системи, потрібно додати наступний рядок у файл конфігурації системи /etc/sysctl.conf:

```
echo "kernel.sem=250 32000 100 128" >> /etc/sysctl.conf
```

2.5.2. Створення множини семафорів

Для створення множини семафорів використовується системний виклик semget() (рис.2.58). Першим параметром key визначається режим іменування множини семафорів (ключ IPC_PRIVATE або заданий ключ key, згенерований функцією ftok). Другим параметром nsems вказується кількість семафорів у множині. Якщо вказати нульове значення nsems=0, то множина семафорів не буде створена. Значення nsems має бути в діапазоні від 1 до SEMMSL. Третім параметром msgflg вказуються потрібні прапорці (IPC_CREAT, IPC_EXCL) та права доступу до множини семафорів.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Рис.2.58. Опис системного виклику semget().

У разі створення множини семафорів, система Linux ініціалізує значення семафорів нульовими значеннями. При цьому згідно стандарту POSIX значення семафорів у новоствореній множині є невизначеними. Разом з тим виклик semget() у ОС Linux та багатьох інших системах реалізовано в такий спосіб, що семафори ініціалізуються нульовими значеннями. В разі успішного спрацювання системний виклик semget() повертає ідентифікатор множини семафорів, асоційований із ключем key. Інакше у випадку помилки semget() повертає -1. Причиною помилки, наприклад, може бути перевищення ресурсних обмежень на використання семафорів. Приклад використання системного виклику semget() наведено на рис.2.59.

```
   int semID;

   if((semID = semget(IPC_PRIVATE, 3, 0xfff)) == -1) {
      fprintf(stderr,"Can't create a semaphore set.\n");
      exit(1);
   }
```

Рис.2.59. Приклад створення множини семафорів з ключем IPC_PRIVATE, яка містить три семафори (nsems=3).

2.5.3. Здійснення операцій над семафорами

Для здійснення операцій над семафорами використовуються системні виклики semop() та semtimedop(). Системний виклик semop() (рис.2.60) здійснює задані операції над підмножиною семафорів з вказаної множини семафорів. Перший параметр semid містить ідентифікатор множини семафорів. Другий параметр sops – це адреса масиву структур sembuf, кожна з яких задає номер окремого семафору у множині, операцію над ним та режим виконання операції (прапорці). У третьому параметрі nsops вказується кількість елементів у масиві sops. Максимально припустимий розмір масиву sops визначається системним параметром SEMOPM, тобто в окремому системному виклику semop() можна виконати не більш SEMOPM операцій. Процес, що здійснює операцію, повинен мати відповідний дозвіл: право на запис для здійснення операцій модифікації значення семафору та право на читання для операції очікування нульового значення семафору. В разі успішного виконання semop() повертає 0, а в разі помилки повертає -1.

```
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>

       int semop(int semid, struct sembuf *sops, size_t nsops);
```

Рис.2.60. Опис системного виклику semop().

Для кожного семафору, над яким потрібно виконати операцію, створюється свій примірник структури sembuf (рис.2.61) - елемент масиву sops. В полях цієї структури вказуються: sem_num - номер семафору у множині, sem_op - операція, яка буде здійснена над цим семафором, sem_flg - прапорці, які визначають режим виконання операції. Нумерація семафорів у множині починається з нуля. Операції з масиву sops виконуються в порядку слідування елементів масиву (in array order) та атомарно. При цьому операції з масиву sops або виконуються, або не виконуються усі разом (as a complete unit). Тобто, якщо серед них є хоча б одна операція, що не може бути виконана в даний момент (блокуюча операція), то усі інші операції з масиву sops також не виконуються.

```
struct sembuf {
      unsigned short  sem_num;      /* semaphore index in array */
      short       sem_op;           /* semaphore operation */
      short       sem_flg;          /* operation flags */
};
```

Рис.2.61. Структура sembuf (sys/sem.h).

Значення sem_op задає три типи операцій над семафором:

1) **sem_op > 0**: ціле додатне значення sem_op буде додано до значення семафору (semval). Якщо для цієї операції вказано прапорець SEM_UNDO (у полі sem_flg), система також відніме значення sem_op від значення змінної semadj (коригування значення семафору). В такому

варіанті (sem_op > 0) операція може бути виконана завжди без блокування викликаючого процесу.

2) **sem_op = 0**: якщо значення семафору дорівнює нулю (semval=0), то виклик semop() завершиться негайно, інакше (якщо semval>0) процес буде заблокований на виклику semop() до моменту, коли значення семафору прийме нульове значення (у разі, якщо семафор буде знищено, процес розблокується).

3) **sem_op < 0**: значення семафору буде зменшено на абсолютну величину sem_op, тобто від значення семафору semval буде віднято |sem_op|. Якщо значення семафору менше |sem_op| (semval<|sem_op|), то процес буде заблокований на виклику semop() до моменту, коли semval стане рівне або більше |sem_op|. Інакше (semval ≥ |sem_op|) після зменшення значення семафору semop() завершиться негайно, і якщо для цієї операції вказано прапорець SEM_UNDO (у полі sem_flg), система додасть значення |sem_op| до значення змінної semadj (коригування значення семафору).

Режим виконання операції над семафором задається за допомогою прапорців в полі sem_flg:

1) **IPC_NOWAIT** - якщо вказано цей прапорець, то операція над семафором здійснюється в асинхронному режимі (без блокування). Тобто якщо задано операцію sem_op, яка не може бути виконана в даний момент (sem_op=0 і semval>0, чи sem_op<0 і semval<|sem_op|), то виклик semop() не блокує викликаючий процес, а завершується негайно, повертаючи -1 (змінна errno приймає значення EAGAIN). Задана операція sem_op при цьому не виконується. Якщо в масиві sops разом з даною вказані інші операції, вони також не виконуються. Таким чином, вказавши прапорець IPC_NOWAIT, можна виконати перевірку можливості виконання операції над семафором без блокування (якщо така можливість є, то операція буде виконана).

2) **SEM_UNDO** - якщо вказано цей прапорець, то операція над семафором здійснюється в «безпечному» режимі зі збереженням змін, які вніс даний процес у значення даного семафору, у змінній semadj (**sem**aphore **adj**ustment). В результаті змінна semadj містить обернену суму усіх операцій, які зробив процес над семафором. Коли процес завершується, до кожного семафору, значення якого він змінював, додається відповідне значення semadj. Тобто значення семафорів корегуються таким чином, щоб скасувати ефект від операцій даного процесу над семафором. В результаті завершення процесу, як правило, не призводить до тупикових ситуацій, коли, наприклад, з'являється процес, назавжди заблокований на семафорі, значення якого може збільшити лише даний процес, який вже завершився. Причинами виникнення таких ситуацій можуть бути як помилки програмування, так і сигнали, що призводять до раптового завершення виконання процесу. Якщо після того, як процес зменшить значення семафору, він отримає сигнал SIGKILL, відновити колишнє значення йому вже не вдасться, оскільки цей сигнал не перехоплюється і призводить до негайного завершення процесу. Відтак, інші процеси, намагаючись звернутись до семафору, зіткнуться з тим, що він заблокований, хоча процес, який його заблокував, вже припинив своє існування. Щоб уникнути виникнення подібних ситуацій, потрібно вказувати прапорець SEM_UNDO при здійсненні операції над семафором.

На рис.2.62-2.64 наведені приклади виконання операцій усіх трьох типів над семафором. В цих прикладах масив sops (масив структур sembuf) зберігається у змінній sem_command і містить лише один елемент (один примірник структури sembuf). На рисунку 2.65 наведено приклад здійснення операцій одночасно над двома семафорами множини семафорів з ідентифікатором semID. В цьому прикладі масив sops також зберігається у змінній sem_command, але містить вже два елемента (два примірника структури sembuf по одній для кожного семафору).

```
struct sembuf sem_command;
.   .   .
sem_command.sem_num = 1; /* selects second semaphore in array */
```

```
sem_command.sem_op = 2; /* increments semval by 2 */
sem_command.sem_flg = SEM_UNDO; /* undoes when the process exits */

semop(semID, &sem_command, 1); /* activates one semaphore */
```

Рис.2.62. Приклад збільшення значення другого семафору
у множині семафорів semID.

```
struct sembuf sem_command;
.   .   .
sem_command.sem_num = 0; /* selects first semaphore in array */
sem_command.sem_op = 0; /* waits for semval==0 */

semop(semID, &sem_command, 1);
```

Рис.2.63. Приклад очікування нульового значення першого семафору
у множині семафорів semID.

```
struct sembuf sem_command;
.   .   .
sem_command.sem_num = 2; /* selects third semaphore in array */
sem_command.sem_op = -4; /* decrements semval by 4 */
sem_command.sem_flg = SEM_UNDO; /* undoes when the process exits */

semop(semID, &sem_command, 1);
```

Рис.2.64. Приклад зменшення значення третього семафору
у множині семафорів semID.

```
struct sembuf sem_command[3];
.   .   .
sem_command[0].sem_num = 0; /* selects first semaphore in array */
sem_command[0].sem_op = -1; /* decrements semval by 1 */
sem_command[0].sem_flg = SEM_UNDO; /* undoes when the process exits */

sem_command[1].sem_num = 2; /* selects third semaphore in array */
sem_command[1].sem_op = 3; /* increments semval by 3 */
sem_command[1].sem_flg = SEM_UNDO; /* undoes when the process exits */

semop(semID, sem_command, 2);
```

Рис.2.65. Приклад зменшення значення першого семафору та збільшення значення третього семафору у множині семафорів semID одним викликом semop().

Системний виклик semtimedop() (рис.2.66) ідентичний виклику semop(), за винятком того, що у випадку, коли викликаючий процес блокується, тривалість блокування обмежена часом, вказаним у четвертому параметрі timeout у вигляді структури timespec (рис.2.67). Час, вказаний у timeout, буде округлено до кроку системного годинника. Внаслідок затримок на роботу ядра по організації паралельного виконання процесів (kernel scheduling delays), реальний час очікування може трохи перевищити заданий час. Після завершення часу очікування, якщо виклик semtimedop() не може виконатись (тобто виконання операції заблоковано), то він завершується і повертає значення -1 (змінна errno приймає значення EAGAIN). При цьому жодна операція з масиву sops не виконується. Якщо параметр timeout=NULL, то semtimedop() виконується так само, як semop(). В разі успішного виконання semtimedop() повертає 0, а в разі помилки повертає -1.

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/sem.h>

int semtimedop(int semid, struct sembuf *sops, size_t nsops,
               const struct timespec *timeout);
```

Рис.2.66. Опис системного виклику semtimedop().

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

Рис.2.67. Структура timespec (time.h).

2.5.4. Управління множиною семафорів

Для виконання команд по управлінню множиною семафорів використовується системний виклик semctl() (рис.2.68). Першим параметром (semid) вказується ідентифікатор множини семафорів, над якою виконується команда. Другим параметром (semnum) вказується номер семафору у множині, над яким виконується команда (якщо команда виконується над усією множиною, цей параметр ігнорується). Третім параметром (cmd) вказується команда по управлінню множиною семафорів або окремим семафором (IPC_STAT, IPC_SET, IPC_RMID, IPC_INFO, SEM_INFO, SEM_STAT, GETALL, GETNCNT, GETPID, GETVAL, GETZCNT, SETALL, SETVAL). Четвертим параметром arg за потреби вказується аргумент команди cmd у вигляді об'єднання типу semun (рис.2.69). Серед полів об'єднання semun є структура типу semid_ds (рис.2.56) та структура типу seminfo (рис.2.70). При успішному завершенні semctl() повертає 0. У випадку помилки semctl() повертає -1.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
int semctl(int semid, int semnum, int cmd, struct semun arg);
```

Рис.2.68. Опис системного виклику semctl().

```
union semun {
    int              val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO
                                (Linux-specific) */
};
```

Рис.2.69. Об'єднання semun.

```
struct  seminfo {
    int semmap;  /* Number of entries in semaphore
                    map; unused within kernel */
    int semmni;  /* Maximum number of semaphore sets */
    int semmns;  /* Maximum number of semaphores in all
                    semaphore sets */
    int semmnu;  /* System-wide maximum number of undo
                    structures; unused within kernel */
    int semmsl;  /* Maximum number of semaphores in a
                    set */
```

```
                int semopm;  /* Maximum number of operations for
                                semop(2) */
                int semume;  /* Maximum number of undo entries per
                                process; unused within kernel */
                int semusz;  /* Size of struct sem_undo */
                int semvmx;  /* Maximum semaphore value */
                int semaem;  /* Max. value that can be recorded for
                                semaphore adjustment (SEM_UNDO) */
            };
```

Рис.2.70. Структура seminfo.

Над множиною семафорів можна виконати наступні команди:

1) **IPC_STAT** – скопіювати поточне значення кожного поля структури даних ядра (semid_ds), асоційованої з ідентифікатором множини семафорів semid, у структуру, на яку вказує arg.buf. В даному випадку у виклику semctl() вказується четвертий параметр arg, який має бути об'єднанням типу semun (рис.2.69) з полем buf, яке в свою чергу має бути структурою типу semid_ds (рис.2.56). Значення другого параметру semnum ігнорується.

2) **IPC_SET** – у структурі даних ядра (semid_ds), асоційованій з ідентифікатором semid, змінити значення діючих ідентифікаторів користувача (sem_perm.uid) і групи (sem_perm.gid), прав доступу (sem_perm.mode) на ті значення, що вказані у структурі arg.buf. В даному випадку buf має бути структурою типу shmid_ds (рис.2.56), а arg - об'єднанням типу semun (рис.2.69). Значення другого параметру semnum ігнорується.

3) **IPC_RMID** – негайно видалити з системи множину семафорів з ідентифікатором semid (рис.2.71) та асоційовану з нею структуру даних (semid_ds). Всі процеси, заблоковані на семафорах з цієї множини, будуть розблоковані. Значення другого параметру semnum ігнорується.

4) **IPC_INFO** – повертає системні обмеження на використання множин семафорів у структуру, на яку вказує arg.__buf. В даному випадку __buf має бути структурою типу seminfo (рис.2.70), а arg - об'єднанням типу semun (рис.2.69).

5) **SEM_INFO** – повертає структуру seminfo (рис.2.70) так само як і IPC_INFO, за виключенням окремих полів, які мають інші значення і відображають поточні витрати системних ресурсів, зокрема: поле semusz містить кількість множин семафорів, які на даний час існують в системі, а поле semaem містить загальну кількість семафорів у всіх множинах семафорів в системі на даний час.

6) **SEM_STAT** – зробити те саме, що робить IPC_STAT, тільки параметром semid задається не ідентифікатор множини семафорів, а індекс у внутрішньому масиві ядра, в якому зберігається інформація про всі множини семафорів в системі.

7) **GETALL** – повернути поточне значення (semval) усіх семафорів множини в arg.array, де arg - це об'єднання типу semun (рис.2.69). Параметр semnum ігнорується. Викликаючий процес повинен мати дозвіл на читання.

8) **GETNCNT** – повернути поточну кількість процесів (semncnt), що очікують на збільшення значення семафору під номером semnum у множині семафорів. Викликаючий процес повинен мати дозвіл на читання.

9) **GETPID** - повернути ідентифікатор процесу (sempid), що здійснив останню операцію над семафором під номером semnum у множині семафорів. Викликаючий процес повинен мати дозвіл на читання.

10) **GETVAL** – повернути поточне значення (semval) семафору під номером semnum у множині семафорів. Викликаючий процес повинен мати дозвіл на читання.

11) **GETZCNT** – повернути поточну кількість процесів (semzcnt), що очікують на нульове значення семафору під номером semnum у множині семафорів. Викликаючий процес повинен мати дозвіл на читання.

12) **SETALL** – 1) встановити значення всіх семафорів з множини у значення, вказані в масиві arg.array, де arg - це об'єднання типу semun (рис.2.69); 2) оновити значення поля

sem_ctime структури semid_ds, асоційованої з множиною семафорів; 3) очистити відповідні значення змінних semadj (коригування значення семафору) у всіх процесах. Якщо нові значення семафорів дозволяють виконати раніше заблоковані виклики semop() в інших процесах, тоді ці процеси розблоковуються. Параметр semnum ігнорується. Викликаючий процес повинен мати дозвіл на запис.

13) **SETVAL** – 1) встановити значення семафору під номером semnum у значення, вказане в полі arg.val, де arg - це об'єднання типу semun (рис.2.69); 2) оновити значення поля sem_ctime структури semid_ds, асоційованої з множиною семафорів; 3) очистити відповідні значення змінних semadj (коригування значення семафору) у всіх процесах. Якщо нове значення семафору дозволяє виконати раніше заблоковані виклики semop() в інших процесах, тоді ці процеси розблоковуються. Викликаючий процес повинен мати дозвіл на запис.

```
int semid;
union semun arg;

if (semctl(semID, 0, IPC_RMID, arg) == -1) {
    perror("semctl");
    exit(1);
}
```

Рис.2.71. Приклад виконання команди управління над множиною семафорів (видалення множини семафорів командою IPC_RMID).

2.5.5. Способи використання семафорів для синхронізації паралельних процесів

Розглянемо способи використання семафорів для синхронізації паралельних процесів на прикладі моделей паралельних обчислень Мастер-Виконавці та Виробник-Споживач.

В моделі паралельних обчислень Мастер-Виконавці (Master-Workers model) можлива ситуація, коли Мастер має дочекатися, щоб усі Виконавці (у кількості N) паралельно виконали якусь дію (наприклад, завершили початкову ініціалізацію змінних вхідними даними перед початком обчислень). В даному випадку синхронізувати роботу Мастера та Виконавців можна за допомогою одного семафору (рис.2.72). Початкове значення цього семафору встановлюється рівним нулю. Кожен з Виконавців після виконання дії додає до значення семафору одиницю. Мастер намагається відняти від значення семафору N, тобто виконати операцію -N. До моменту коли всі N Виконавців додадуть по одиниці до значення семафору, Мастер буде заблокований на виклику операції -N. Після того, як усі Виконавці "відзвітуються" про виконання дії додаванням 1 до значення семафору, Мастер зможе виконати операцію -N і розблокується.

```
// псевдо-код Мастера
створити семафор alldone;
встановити значення семафору alldone рівним нулю;
запустити N процесів-Виконавців;
виконати над семафором alldone операцію -N;
перейти до наступного кроку управління Виконавцями;

// псевдо-код Виконавця
виконати задані обчислення;
виконати над семафором alldone операцію +1;
чекати на подальші вказівки Мастера;
```

Рис.2.72. Псевдо-код Мастера та Виконавця в моделі Мастер-Виконавці.

В моделі паралельних обчислень Виробник-Споживач (Producer–consumer model) один процес "виробляє" і "постачає" іншому вхідні дані для обробки. В найбільш загальному

варіанті моделі розглядається ситуація, коли є багато виробників та багато споживачів (рис.2.73). Дані передаються через буфер, розмір якого, як правило, рахується обмеженим (наприклад, це може бути кільцевий буфер). Синхронізувати роботу процесів-виробників та процесів-споживачів можна за допомогою трьох семафорів (рис.2.74): 1) emptyCount – кількість порожніх комірок у буфері; 2) fullCount - кількість комірок з даними у буфері; 3) lockBuffer – замок, який закриває буфер для інших процесів на час здійснення операції запису чи читання даних.

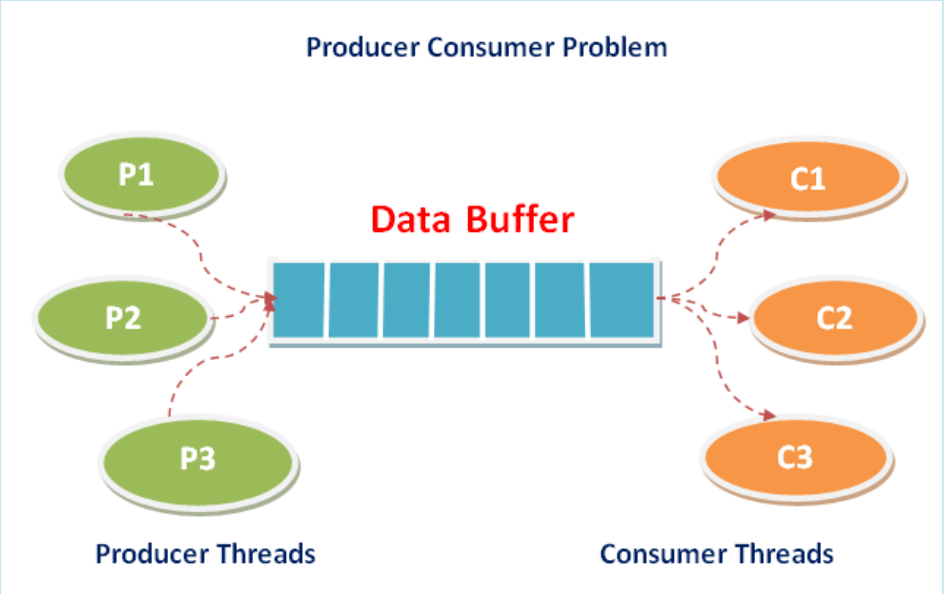

Рис.2.73. Модель паралельних обчислень Постачальник-Споживач.

```
створити семафори emptyCount, fullCount, lockBuffer;
встановити значення семафору emptyCount рівним M;
встановити значення семафору fullCount рівним 0;
встановити значення семафору lockBuffer рівним 1;

// псевдо-код Виробника
Цикл {
   підготувати дані (produce);
   виконати над семафором emptyCount операцію -1;
   виконати над семафором lockBuffer операцію -1;
   покласти дані у буфер;
   виконати над семафором lockBuffer операцію +1;
   виконати над семафором fullCount операцію +1;
}

// псевдо-код Споживача
Цикл {
   виконати над семафором fullCount операцію -1;
   виконати над семафором lockBuffer операцію -1;
   забрати дані з буферу;
   виконати над семафором lockBuffer операцію +1;
   виконати над семафором emptyCount операцію +1;
   обробити дані (consume);
}
```

Рис.2.74. Псевдо-код Виробника та Споживача в моделі Виробник-Споживач.

Одна з основних відмінностей між семафором та м'ютексом полягає в тому, що змінити значення семафору може будь який з процесів (чи потоків), а значення “захопленого” м'ютексу може змінити (“звільнити”) лише той процес (потік), який його “захопив”. Обираючи поміж

семафором (semaphore), м’ютексом (mutex) та спінлоком (spinlock) для синхронізації паралельного виконання процесів чи потоків, потрібно враховувати наступне.

1) Семафор використовується в ситуаціях, коли потрібно, щоб процес/потік був заблокований, поки якийсь інший процес/потік його не розблокує. Наприклад у моделі Виробник-Споживач Виробник має бути заблокований, поки принаймні один слот буфера не стане порожнім (вільним). При цьому лише Споживач може забрати дані з буферу та повідомити що слот буфера звільнився.

2) М’ютекс використовується в ситуаціях, коли процесу/потоку потрібно виконати код, який одночасно з цим не повинен виконуватися жодним іншим процесом/потоком. Операція захоплення та звільнення м’ютекса відбувається в одному і тому самому процесі/потоці. Наприклад, якщо потрібно видалити вузол з глобального зв'язаного списку, потрібно виключити можливість операцій над списком інших процесів/потоків в цей момент. Для цього потрібно захопити відповідний м’ютекс, і тоді будь-який інший процес/потік, що звернувся до списку буде чекати на звільнення м’ютексу.

3) Спінлок використовується в ситуаціях, коли нібито потрібен м’ютекс, але призупинка процесу/потоку на блокуючому виклику функції lock() неприпустима. Наприклад, обробник переривання у ядрі ОС ніколи не повинен блокуватись. Якщо це станеться, система зависне (вийде з ладу). Якщо потрібно вставити вузол у загальнодоступний зв'язаний список з обробника переривання, треба захопити спінлок, вставити вузол, звільнити спінлок.

2.5.6. Програмний інтерфейс для роботи з семафорами стандарту POSIX

Стандарт POSIX визначає тип даних семафору sem_t та набір функцій для роботи з ним (заголовний файл semaphore.h). Перелік функцій наведено в табл.2.1.

Таблиця 2.1. Перелік функцій для роботи з семафорами (стандарт POSIX).

Функція	**Опис**
sem_init()	Ініціалізація неіменованого семафору, встановлення його початкового значення та режиму використання (на рівні процесів чи на рівні процесів і потоків).
sem_destroy()	Звільнення семафору.
sem_open()	Створення нового або підключення до існуючого іменованого семафору.
sem_close()	Закриття семафору після закінчення роботи за ним.
sem_unlink()	Знищити іменований семафор (ім’я видаляється негайно, сам семафор знищується після того, як всі процеси його закриють).
sem_wait()	Зменшити значення семафору на 1.
sem_timedwait()	Зменшити значення семафору на 1 із заданим часом блокування, по завершенню якого повертається помилка.
sem_trywait()	Спроба зменшити значення семафору на 1 в режимі без блокування; повертає помилку, якщо зменшення без блокування неможливе.
sem_post()	Збільшити значення семафору на 1.
sem_getvalue()	Отримати поточне значення семафору.

2.5.7. Реалізація механізму семафорів за допомогою системного виклику eventfd()

В ОС Linux існує альтернативна можливість реалізації механізму семафорів, зокрема за допомогою системного виклику eventfd() (рис.2.75). Семафор реалізується у вигляді лічильника, зв'язаного з файловим дескриптором системним викликом eventfd() з прапорцем EFD_SEMAPHORE. При "читанні" такого лічильника функцією read() він зменшується на 1, якщо його значення було ненульовим. Якщо він має нульове значення, то відбувається блокування (якщо не вказано прапорець EFD_NONBLOCK), як і у випадку зі звичайним семафором. Функція write() збільшує значення лічильника на число, яке "записується" за відповідним файловим дескриптором. Перевагою такого семафору є можливість очікування сигнального стану семафору разом з іншими подіями за допомогою системних викликів select() або poll().

```
#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);
```

Рис.2.75. Опис системного виклику eventfd().

Системний виклик eventfd() створює об'єкт eventfd, який може використовуватися як механізм очікування подій та сповіщення про події прикладними програмами, або для сповіщення прикладних програм про події ядром. Першим параметром initval задається початкове значення лічильника, який буде міститися в об'єкті eventfd. У другому параметрі (flags) вказуються прапорці. системний виклик eventfd() повертає новий файловий дескриптор, який використовується для виконання операцій над відповідним об'єктом eventfd. В разі помилки eventfd() повертає -1. На рис.2.76 наведено приклад створення об'єкту eventfd, що використовується в режимі семафору.

```
int semfd;

if((semfd = eventfd(1, EFD_SEMAPHORE)) == -1) {
   fprintf(stderr,"Can't create an eventfd object.\n");
   exit(1);
}
```

Рис.2.76. Приклад створення об'єкту eventfd, що використовується в режимі семафору з початковим значенням 1.

2.6. Сигнали (signals)

2.6.1. Використання сигналів в ОС Linux

Сигнал (signal) в ОС Linux – це повідомлення обчислювального процесу про те, що сталася деяка подія. Сигнали іноді називають програмними перериваннями (software interrupts). Сигнали є аналогом апаратних переривань, оскільки вони переривають нормальний потік виконання програми. У більшості випадків неможливо точно передбачити, коли надійде сигнал.

Один процес може (якщо він має відповідні дозволи) надіслати сигнал іншому процесу. При цьому сигнали можуть використовуватися як механізм синхронізації процесів або як проста форма міжпроцесної взаємодії (IPC). Також можливою є ситуація, коли процес надсилає сигнал сам собі. Однак в більшості випадків джерелом сигналів, що надходять до процесу, є ядро операційної системи. Серед типів подій, які призводять до генерування ядром сигналу розрізняють наступні:

1) Виникла виняткова ситуація на апаратному рівні (hardware exception), про яку було повідомлено ядро, яке, в свою чергу, надіслало відповідний сигнал процесу. Приклади: апаратне забезпечення виявило стан несправності, виконання неправильної машинної інструкції, ділення на 0, посилання на комірку пам'яті, яка є недоступною процесу.

2) Користувач ввів з клавіатури один зі спеціальних символів терміналу, які генерують сигнали. Приклади: символ переривання (Ctrl+C), символ призупинення (Ctrl+Z).

3) Сталася деяка програмна подія. Приклади: для дескриптора файлу став можливим ввід, було змінено розмір вікна терміналу, спрацював таймер, перевищено обмеження на використання процесорного часу даним процесом, завершився синівський процес.

Кожен сигнал має свій номер у вигляді унікального цілого числа, починаючи з 1. Ці номери визначено у заголовному файлі <signal.h>, де для кожного сигналу також визначені назви у вигляді SIG****. Оскільки номери сигналів можуть різнитись у версіях ОС Linux для різних апаратних платформ (alpha, sparc, x86, arm, mips), в програмах завжди використовуються їх назви. Наприклад, коли користувач вводить з клавіатури Ctrl+C, система надсилає процесу сигнал SIGINT (сигнал номер 2 у версіях ОС Linux для всіх архітектур).

Сигнали поділяються на дві групи. Перша група - це, так звані, традиційні або стандартні сигнали (standard signals, POSIX reliable signals), які використовуються ядром для сповіщення процесів про події. В ОС Linux стандартні сигнали нумеруються від 1 до 31. Перелік стандартних сигналів наведено в табл.2.2. Друга група сигналів - це сигнали реального часу (POSIX real-time signals), які були добавлені пізніше (у стандарті POSIX.1b), і покликані зняти деякі обмеження, які притаманні стандартним сигналам. Сигнали реального часу в ОС Linux нумеруються від 32 до 63.

Сигнали реального часу відрізняються від стандартних сигналів наступним:

1) Більший діапазон сигналів, які можуть бути використані програмістом у своїх цілях (серед стандартних таких сигналів лише два: SIGUSR1 та SIGUSR2).

2) Сигнали реального часу буферизуються у чергу. Якщо процесу надсилаються кілька примірників деякого сигналу реального часу, то вони передаються йому відповідну кількість разів. На відміну від цього, якщо надіслати кілька примірників стандартного сигналу процесу, який вже отримує цей сигнал (pending), то він буде отриманий процесом лише один раз.

3) При надсиланні сигналу реального часу можна додатково передати дані (ціле число або значення покажчика), що супроводжують сигнал. Обробник сигналу в процесі, що отримав сигнал, може дістати ці дані.

4) Порядок доставки різних сигналів реального часу гарантований. Якщо надходять декілька різних сигналів реального часу, то спочатку передається сигнал з найменшим номером. Іншими словами, сигнали мають пріоритет (сигнали з меншими номерами мають вищий пріоритет). Коли кілька сигналів одного типу буферизуються у черзі, вони передаються разом із супровідними даними в тому порядку, в якому вони були відправленні.

Кожний сигнал генерується якоюсь подією. Після генерування, сигнал надходить до процесу, який виконує певні дії у відповідь на сигнал. Між часом, коли сигнал згенеровано, і часом, коли він надійшов до процесу, він знаходиться в стані передавання (pending signal).

Як правило, сигнал надходить до процесу майже відразу, як тільки процес отримує черговий квант часу процесора, або негайно, якщо процес вже виконується в процесорі (наприклад, якщо процес надіслав сигнал сам собі). Однак іноді потрібно зробити так, щоб виконання деякого сегменту коду не переривалося сигналом. Для цього можна додати сигнал до маски сигналів процесу (signal mask) - набору сигналів, отримання яких в даний момент заблоковано. Якщо згенерований сигнал заблоковано, він продовжує знаходитись в стані передавання (pending) до моменту, коли пізніше він буде розблокований (видалений з маски сигналів).

За замовчуванням після отримання процесом сигналу, виконується одна з наступних дій (залежно від номера сигналу, див. табл.2.2):

1) Сигнал ігнорується, тобто відхиляється ядром і ніяк не впливає на процес. Процесу навіть “невідомо”, що надходив сигнал.

2) Процес завершується (“вбивається”). Іноді це називають аномальним завершенням процесу, на відміну від звичайного завершення, коли процес завершується за допомогою exit().

3) Створюється файл дампу ядра (core dump file) і процес завершується. Файл дампу ядра містить образ віртуальної пам'яті процесу, який можна завантажити в налагоджувач (debugger) для аналізу стану процесу на момент його завершення.

4) Виконання процесу призупиняється (stopped).

5) Виконання процесу відновлюється (resumed) після попередньої призупинки.

У програміста є можливість змінити дії, які виконуються після отримання того чи іншого сигналу, шляхом встановлення диспозиції сигналу (disposition of the signal). В програмі можна встановити одну з наступних диспозицій сигналу:

1) Виконати дію за замовчуванням. Це, як правило, робиться для скасування попередньої диспозиції сигналу, яка була встановлена перед цим.

2) Ігнорувати сигнал. Це, наприклад, застосовуються для сигналів, які за замовчуванням завершують виконання процесу.

3) Виконати власний обробник сигналу (перехопити сигнал). В цьому випадку замість стандартного обробника сигналу (дія за замовчуванням) виконується обробник сигналу, який задається програмістом у вигляді функції. Сигнали SIGKILL та SIGSTOP не можуть бути перехоплені.

Таблиця 2.2. Стандартні сигнали (ОС Linux).

Ім'я	**Опис**	**Дія за замовчуванням**
SIGABRT	Аварійне завершення (*abort*)	Завершити + core
SIGALRM	Збіг час таймера (*alarm*)	Завершити
SIGBUS	Апаратна помилка	Завершити + core
SIGCHLD	Зміна стану дочірнього процесу	Ігнорувати
SIGCONT	Відновити роботу призупиненого процесу	Продовжити/ігнорувати
SIGFPE	Арифметична помилка	Завершити + core
SIGHUP	Обрив зв'язку	Завершити
SIGILL	Неприпустима інструкція	Завершити + core
SIGINT	З терміналу введений символ переривання	Завершити
SIGIO	Асинхронне введення-виведення	Завершити
SIGIOT	Апаратна помилка	Завершити + core
SIGKILL	Завершення	Завершити
SIGPIPE	Запис в канал, з якого ніхто не читає	Завершити
SIGPOLL	Подія опитування (*poll*)	Завершити
SIGPROF	Збіг час профілюючого таймера (*setitimer*)	Завершити
SIGPWR	Падіння напруги живлення / перезапуск	Завершити

SIGQUIT	З терміналу введений символ завершення	Завершити + core
SIGSEGV	Помилка доступу до пам'яті	Завершити + core
SIGSTKFLT	Помилка, пов’язана зі стеком процесора	Завершити
SIGSTOP	Призупинити процес	Зупинити процес
SIGSYS	Невірний системний виклик	Завершити + core
SIGTERM	Завершення	Завершити
SIGTRAP	Апаратна помилка	Завершити + core
SIGTSTP	З терміналу введений символ призупинення	Зупинити процес
SIGTTIN	Читання з керуючого терміналу фоновим процесом	Зупинити процес
SIGTTOU	Запис в керуючий термінал фоновим процесом	Зупинити процес
SIGURG	Екстрена подія (гнізда)	Ігнорувати
SIGUSR1	Сигнал, що визначається користувачем	Завершити
SIGUSR2	Сигнал, що визначається користувачем	Завершити
SIGVTALRM	Збіг час віртуального таймера (*setitimer*)	Завершити
SIGWINCH	Зміна розмірів вікна терміналу	Ігнорувати
SIGXCPU	Вичерпано ліміт процесорного часу (*setrlimit*)	Завершити + core
SIGXFSZ	Перевищено обмеження на розмір файлу (*setrlimit*)	Завершити + core

2.6.2. Надсилання сигналу обчислювальному процесу

Для надсилання сигналу процесу (процесам) використовуються системні виклики kill(), killpg() та raise(). За допомогою функції alarm() процес надсилає сам собі сигнал SIGALRM, який надійде через вказаний час. За допомогою функції abort() процес надсилає сам собі сигнал SIGABRT (аварійне завершення процесу), який буде отримано негайно.

За допомогою системного виклику kill() (рис.2.77) один процес може надіслати сигнал іншому процесу або групі процесів. Перший параметр pid визначає ідентифікатор процесу чи групи процесів, яким надсилається сигнал, зокрема:

1) pid > 0 – процес з номером pid,

2) pid = 0 – всім процесам у групі процесів, в яку входить даний процес,

3) pid = -1 – всім процесам, яким даний процес має право надіслати сигнал, за виключенням процесу init (PID = 1),

4) pid < -1 – всім процесам у групі процесів з ідентифікатором |pid|.

Якщо процесу з вказаним pid не існує, системний виклик kill() поверне помилку.

Другим параметром sig визначається номер сигналу у вигляді його назви (табл.2.2). Якщо параметр sig дорівнює 0 (порожній сигнал), то виконується перевірка коректності звертання до kill(), але сигнал у дійсності не надсилається. Це може бути використано для перевірки того, чи правильно заданий аргумент pid (чи існує відповідний процес) або чи є у даного процесу дозвіл на надсилання сигналу процесу із вказаним pid.

Процеси, що відправляють та отримують сигнал, повинні мати той самий реальний чи діючий ідентифікатор користувача, якщо тільки діючий ідентифікатор користувача процесу, що надсилає сигнал, не є ідентифікатором суперкористувача (root). При успішному завершенні kill() повертає 0. У випадку помилки kill() повертає -1 (змінній errno присвоюється код помилки).

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Рис.2.77. Опис системного виклику kill().

Функція killpg() (рис.2.78) дозволяє надіслати сигнал вказаній групі процесів. Першим параметром вказується ідентифікатор групи процесів, а другим номер сигналу. Якщо pgrp = 0, то сигнал надсилається всім процесам у групі процесів, в яку входить даний процес. При успішному завершенні killpg() повертає 0. У випадку помилки killpg() повертає -1 (змінній errno присвоюється код помилки).

```
#include <signal.h>

int killpg(int pgrp, int sig);
```

Рис.2.78. Опис системного виклику killpg().

Функція raise() (рис.2.79) надсилає вказаний сигнал (параметр sig) викликаючому процесу чи програмному потоку, тобто за допомогою raise() процес може надіслати сигнал сам собі. Якщо процес складається лише з основного потоку, то raise(sig) еквівалентно виклику kill(getpid(),sig). Якщо raise(sig) викликається в окремому програмному потоці процесу, то еквівалентом є pthread_kill(pthread_self(),sig). Якщо обробник сигналу перехоплено, то raise() завершиться лише після завершення власного обробника сигналу. В разі успішного завершення raise() повертає 0, а в разі помилки - від'ємне значення.

```
#include <signal.h>

int raise(int sig);
```

Рис.2.79. Опис системного виклику raise().

Функція alarm() (рис.2.80) надсилає сигнал SIGALRM викликаючому процесу через задану кількість секунд (параметр seconds). Якщо параметр seconds дорівнює нулю, то встановлене перед цим очікування на сигнал SIGALRM скидається (відміняється). При виникненні будь якої події раніше встановлене очікування на сигнал SIGALRM скидається. Функція alarm() повертає кількість секунд, що лишилися до надходження сигналу SIGALRM, встановленого попереднім викликом alarm(), або 0, якщо попереднього виклику alarm() не було.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Рис.2.80. Опис системного виклику alarm().

Функція abort() (рис.2.81) надсилає сигнал SIGABRT викликаючому процесу, що призводить до його негайного аварійного завершення. Якщо сигнал SIGABRT перед цим був заблокований, то функція abort() його розблокує. Так само якщо була встановлена диспозиція сигналу SIGABRT - ігнорувати чи виконати власний обробник сигналу (перехоплення), то функція abort() скасує цю диспозицію і сигнал SIGABRT все одно призведе до завершення процесу.

```
#include <stdlib.h>

void abort(void);
```

Рис.2.81. Опис системного виклику abort().

2.6.3. Встановлення диспозиції сигналу

Встановити диспозицію сигналу можна за допомогою двох системних викликів - signal() та sigaction(). Оскільки реалізація системного виклику signal() іноді відрізняється в різних версіях ОС UNIX та ОС Linux, рекомендується для переносимості програмного коду замість нього використовувати системний виклик sigaction().

Системний виклик signal() (рис.2.82) дозволяє викликаючому процесу обрати одну з трьох можливих диспозицій сигналу (способів реакції на сигнал, який буде отримано). Перший параметр signum задає номер сигналу, а другий параметр handler - диспозицію (SIG_DFL, SIG_IGN або адресу функції). Змінити диспозицію сигналів SIGKILL та SIGSTOP неможливо (тобто вони не можуть бути проігноровані чи перехоплені). Макроси SIG_DFL і SIG_IGN визначені у файлі <signal.h>. Кожний з макросів - це унікальна константа типу "покажчик на функцію типу void".

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Рис.2.82. Опис системного виклику signal().

Диспозиції, які можуть бути вказані параметром handler наступні:

1) **SIG_DFL** – встановити дію за замовчуванням (стандартну реакцію на сигнал).

2) **SIG_IGN** – ігнорувати сигнал signum.

3) **адреса_функції** – встановити режим перехоплення сигналу signum (при його отриманні буде виконуватись функція обробника сигналу handler, якій в якості єдиного параметру функції буде передано номер сигналу signum).

З точки зору логіки роботи системного виклику signal() можливі два основних варіанти розвитку подій після перехоплення сигналу відповідним викликом signal():

1) після надходження сигналу signum його диспозиція скидається у значення за замовчуванням (SIG_DFL), після чого виконується власний обробник сигналу handler; на час виконання обробника handler надходження сигналу signum не блокується; така логіка роботи реалізована у класичному UNIX, System V та всередині ядра Linux;

2) після надходження сигналу signum його диспозиція зберігається як handler (перехоплення); виконується власний обробник сигналу handler; на час виконання обробника handler надходження сигналу signum блокується; така логіка роботи реалізована у BSD та функції signal() бібліотеки glibc 2 (яка не викликає системний виклик signal(), а є обгорткою sigaction() з відповідними прапорцями).

З огляду на це, використовувати signal() для перехоплення сигналу не рекомендується. Замість цього рекомендується використовувати sigaction().

Після завершення функції обробки сигналу процес, що одержав сигнал, відновлює своє виконання з точки переривання. При успішному завершенні системного виклику signal() повертається попереднє значення handler для зазначеного сигналу signum. У разі помилки повертається значення SIG_ERR, а змінній errno присвоюється код помилки.

Альтернативою виклику signal() є системний виклик sigaction() (рис.2.83), який є більш складним у використанні, але разом з тим дає більшу гнучкість та більший контроль над режимом перехоплення сигналу. Першим параметром вказується номер сигналу (signum). Другим параметром (act) вказується нова диспозиція сигналу у вигляді структури sigaction (рис.2.84). Через третій параметр (oldact) повертається попередня диспозиція сигналу, також у вигляді структури sigaction. Якщо другий чи третій параметр непотрібні, то замість них можна вказати NULL.

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Рис.2.83. Опис системного виклику sigaction().

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Рис.2.84. Структура sigaction.

Для того, щоб встановити нову диспозицію сигналу signum за допомогою системного виклику sigaction(), потрібно перед його викликом у полях структури sigaction, на яку вказує змінна act, вказати:

1) sa_handler – диспозицію сигналу (SIG_DFL, SIG_IGN або адресу функції handler),

2) sa_mask – набір сигналів, які будуть заблоковані на час виконання обробника сигналу handler (після його завершення вони будуть розблоковані),

3) sa_flags – прапорці, які управляють різними аспектами режиму перехоплення сигналу (табл.2.3).

Приклад використання системного виклику sigaction() наведено на рис.2.92.

Таблиця 2.3. Прапорці sa_flags.

Прапорець	**Опис**
SA_NOCLDSTOP	Для сигналу *SIGCHLD* – не генерувати цей сигнал під час призупинення синівського процесу
SA_NOCLDWAIT	Для сигналу *SIGCHLD* – запобігти створенню процесів-зомбі по завершенні синівських процесів
SA_NODEFER	Не блокувати сигнал автоматично під час виклику функції обробника сигналу (якщо сигнал не включений до складу маски *sa_mask*)
SA_ONSTACK	Викликати обробник сигналу в додатковому стеку сигналів, отриманому з допомогою функції *sigaltstack*. Якщо додатковий стек є недоступним, то буде використаний стек за замовчуванням
SA_RESETHAND	На вході до функції обробника встановити диспозицію сигналу в значення *SIG_DFL* і скинути прапорець *SA_SIGINFO*
SA_RESTART	Здійснювати автоматичний перезапуск системних викликів, що перервані цим сигналом
SA_SIGINFO	За наявності цього прапорця в обробник сигналу передається додаткова інформація: вказівники на структуру *siginfo* та контекст процесу

2.6.4. Маскування сигналів

Для кожного процесу ядро зберігає маску сигналів (signal mask) – набір сигналів, отримання яких процесом в даний момент заблоковано. Якщо сигнал надіслано і він входить до числа заблокованих, то його отримання відкладається (pending) до моменту, коли він буде розблокований (видалений з маски сигналів).

В будь який момент часу за допомогою системного виклику sigprocmask() (рис.2.85) сигнал можна додати до маски сигналів. Одночасно з цим sigprocmask() можна використати

для отримання інформації про поточну маску сигналів. Першим параметром (how) задається режим встановлення маски:

1) **SIG_BLOCK** – сигнали, зазначені в наборі сигналів, на який вказує параметр set, додаються до маски сигналів; тобто, виконується об'єднання поточної маски сигналів та набору сигналів set;

2) **SIG_UNBLOCK** – сигнали в наборі сигналів, на який вказує параметр set, видаляються з маски сигналів; спроба розблокувати сигнал, який на даний момент не заблоковано, не призводить до помилки;

3) **SIG_SETMASK** – сигнали в наборі сигналів, на який вказує параметр set, стають новою маскою сигналів.

В третьому параметрі oldset повертається попередня маска сигналів. Якщо другий параметр (set) встановлено в NULL, то sigprocmask() просто поверне поточну маску сигналів (oldset), не змінюючи її.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Рис.2.85. Опис системного виклику sigprocmask().

Переглянути надіслані процесу але заблоковані на даний момент сигнали (pending signals) можна за допомогою системного виклику sigpending() (рис.2.86). Перелік таких сигналів повертається в параметрі set. Після цього можна перевірити, чи входить даний сигнал у набір set, за допомогою функції sigismember().

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Рис.2.86. Опис системного виклику sigpending().

2.6.5. Створення обробника сигналу

Обробник сигналу створюється у вигляді окремої функції з одним параметром (номером сигналу). Адреса цієї функції вказується в параметрі handler системного виклику signal() або в полі sa_handler структури sigaction (параметр act) перед викликом sigaction(). Приклад опису функції обробника сигналу наведено на рис.2.87. В момент отримання сигналу виконання програми призупиняється на час виконання функції обробника сигналу (рис.2.88).

```
void catch_int(int sig_num)
{
    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
    /* and print the message */
    printf("Don't do that");
    fflush(stdout);
}

. . .

/* set the INT (Ctrl-C) signal handler to 'catch_int' */
signal(SIGINT, catch_int);

. . .
```

Рис.2.87. Приклад опису функції обробника сигналу.

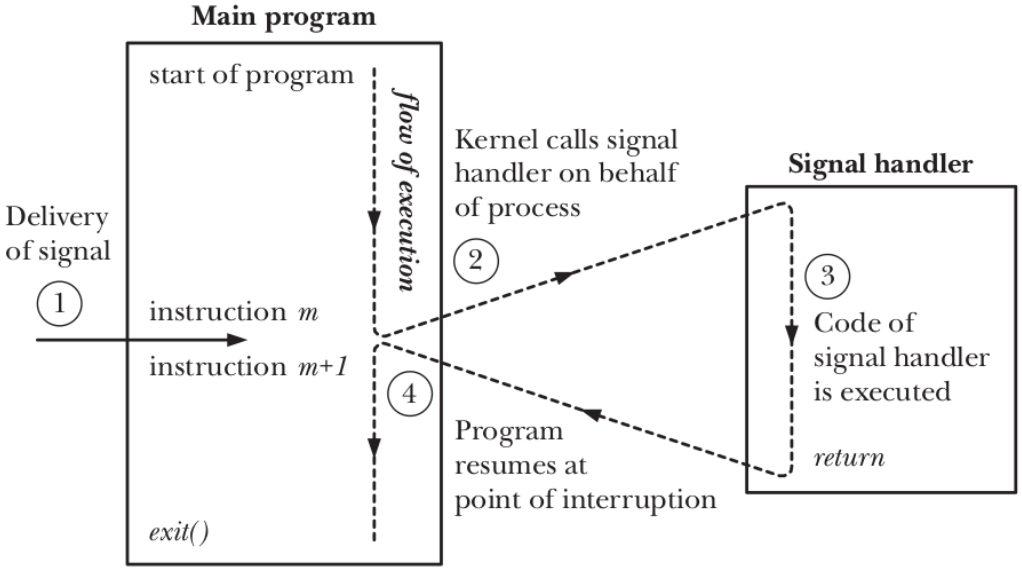

Рис.2.88. Схема виконання функції обробника сигналу.

При створенні обробника сигналу потрібно зважати на той факт, що місце у програмі, на якому вона була зупинена для виконання обробника (рис.2.88), є невідомим. Більше того робота програми може бути перервана в момент виконання бібліотечної функції чи навіть системного виклику. Якщо в обробнику викликається та сама функція, що була перервана, то виникає загроза некоректного виконання цієї функції (наприклад, функція виділення пам'яті). З цієї точки зору функції поділяються на реентерабельні та нереентерабельні функції (reentrant and nonreentrant functions).

Реентерабельна функція може безпечно (без збоїв) виконуватися одночасно кількома програмними потоками одного процесу (і в тому числі обробником сигналу). Нереентерабельні функції, до яких, наприклад, відносяться функції що працюють з глобальними структурами даних, не можуть використовуватись у обробнику сигналу. Для бібліотечних функцій (в тому числі системних викликів) їх тип (реентерабельна чи ні) визначається відповідними стандартами. Реентерабельні функції та функції, виконання яких не може бути перервано обробкою сигналу, називаються функціями захищеними від асинхронних сигналів (async-signal-safe functions). Перелік відповідних бібліотечних функцій визначено стандартами POSIX.1-1990, SUSv2 та SUSv3.

2.6.6. Отримання сигналу обчислювальним процесом

Окрім звичайного "несподіваного" отримання сигналу в асинхронному режимі, можна організувати очікування сигналу та його отримання в синхронному режимі за допомогою функцій pause(), sigsuspend(), sigwaitinfo(), sigtimedwait(), signalfd().

Функція pause() призупиняє виконання процесу до моменту отримання сигналу. Якщо цей сигнал перехоплений, то після виконання обробника, процес продовжить виконання (виклик pause() поверне -1), інакше під впливом сигналу процес завершиться (дія за замовчуванням).

В деяких ситуаціях, коли сигнал раніше був заблокований (доданий до маски сигналів), потрібно його розблокувати і дочекатись отримання цього сигналу. Якщо робити це "неатомарно" окремими викликами функцій (sigprocmask() та pause()), то зберігається можливість збою - сигнал може прийти після виклику sigprocmask() і перед викликом pause(), внаслідок чого після повернення з обробника процес призупиниться на виклику pause() і буде

чекати на повторне отримання сигналу. В таких ситуаціях використовується системний виклик sigsuspend() (рис.2.89), який виконує розблокування і очікування сигналу атомарно.

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

Рис.2.89. Опис системного виклику sigsuspend().

Системний виклик sigsuspend() замінює маску сигналів процесу на набір сигналів, на який вказує mask, і призупиняє виконання процесу, поки сигнал не буде отриманий, і його обробник буде завершений. Після завершення роботи обробника sigsuspend() відновлює ту маску сигналів процесу, якою вона була до виклику sigsuspend().

Для очікування надходження сигналів в синхронному режимі використовуються системні виклики sigwaitinfo() та sigtimedwait() (рис.2.90). Виклик sigwaitinfo() призупиняє процес до моменту, коли сигнал з набору set потрапить у надіслані процесу але заблоковані на даний момент сигнали (pending signals). Відтак перед викликом sigwaitinfo() потрібно заблокувати потрібні сигнали за допомогою sigprocmask(). Отриманий сигнал видаляється з набору надісланих процесу сигналів (list of pending signals), і sigwaitinfo() повертає номер цього сигналу (в параметрі info повертається інформація про сигнал). Виклик sigtimedwait() виконується так само як sigwaitinfo(), але очікує надходження сигналу лише на протязі часу, який задається параметром timeout.

```
#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);

int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);
```

Рис.2.90. Опис системних викликів sigwaitinfo() та sigtimedwait().

Ще один спосіб організувати очікування надходження сигналів в синхронному режимі - це скористатись системним викликом signalfd() (рис.2.91). За допомогою цього виклику створюється спеціальний файловий дескриптор fd, з яким пов’язується набір сигналів, вказаний у mask. В подальшому за допомогою системного виклику read() з файловим дескриптором fd ми можемо очікувати на надходження сигналу з набору mask до множини надісланих процесу але заблокованих на даний момент сигналів (pending signals).

```
#include <sys/signalfd.h>

int signalfd(int fd, const sigset_t *mask, int flags);
```

Рис.2.91. Опис системного виклику signalfd().

2.6.7. Організація взаємодії процесів за допомогою сигналів

Сигнали можна розглядати як форму міжпроцесної взаємодії (IPC). Наприклад, за допомогою надсилання перевизначених (перехоплених) стандартних сигналів SIGUSR1 та SIGUSR2 процеси можуть повідомляти один одного про свій стан чи якусь подію в асинхронному режимі. На рис.2.92 наведено приклад використання сигналів для синхронізації дій батьківського та синівського процесів. В наведеній програмі батьківський процес очікує, доки синівський виконає якусь дію (в цьому плані “ролі” батьківського та синівського процесу можуть бути легко змінені). Зверніть увагу, що сигнал синхронізації (SIGUSR1) блокується

перед викликом fork(). Якщо блокувати його після fork(), то можуть виникнути “перегони”, які призведуть до помилкової роботи програми.

Однак сигнали, як механізм IPC, мають ряд обмежень. По-перше, порівняно з іншими механізмами IPC використання сигналів є громіздким і складним, внаслідок наступних причин.

1) Асинхронний характер сигналів породжує низку проблем, включаючи спеціальні вимоги до реентерабельності функцій, можливість виникнення “перегонів” (race conditions), забезпечення коректної роботи обробників сигналів з глобальними змінними. (Більшість з цих проблем не виникає, якщо ми використовуємо sigwaitinfo() або signalfd() для отримання сигналів в синхронному режимі.)

2) Стандартні сигнали не буферизуються в черзі. І навіть для сигналів реального часу існують обмеження на кількість сигналів, що буферизуються в черзі. Тому для того, щоб уникнути втрати інформації, процес, який отримує сигнал, повинен мати можливість інформувати відправника про те, що він готовий прийняти наступний сигнал. Найбільш очевидний спосіб реалізувати це, зробити так, щоб отримувач в свою чергу надсилав сигнал відправнику.

По-друге проблема полягає в тому, що сигнали несуть лише обмежену кількість інформації: номер сигналу, а у випадку сигналів реального часу – додаткові дані у вигляді цілого числа або покажчика. Така низька “пропускна здатність” робить сигнали повільними порівняно з іншими механізмами IPC (наприклад, неіменованими каналами). Як наслідок зазначених обмежень, сигнали рідко використовуються як механізм IPC.

```
#include <signal.h>
#include "curr_time.h"
#include "tlpi_hdr.h" /* Declaration of currTime() */
#define SYNC_SIG SIGUSR1 /* Synchronization signal */

/* Signal handler - does nothing but return */
static void handler(int sig)
{
}

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

  pid_t childPid;
  sigset_t blockMask, origMask, emptyMask;
  struct sigaction sa;
  setbuf(stdout, NULL);

  /* Disable buffering of stdout */
  sigemptyset(&blockMask);
  sigaddset(&blockMask, SYNC_SIG);

  /* Block signal */
  if (sigprocmask(SIG_BLOCK, &blockMask, &origMask) == -1)
    errExit("sigprocmask");
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sa.sa_handler = handler;

  if (sigaction(SYNC_SIG, &sa, NULL) == -1)
    errExit("sigaction");

  switch (childPid = fork()) {

  case -1:
    errExit("fork");

  case 0: /* Child */
```

```
    /* Child does some required action here... */
    printf("[%s %ld] Child started - doing some work\n",
           currTime("%T"), (long) getpid());
    sleep(2);     /* Simulate time spent doing some work */


    /* And then signals parent that it's done */
    printf("[%s %ld] Child about to signal parent\n",
           currTime("%T"), (long) getpid());
    if (kill(getppid(), SYNC_SIG) == -1)
      errExit("kill");

    /* Now child can do other things... */
    _exit(EXIT_SUCCESS);

  default: /* Parent */

    /* Parent may do some work here, and then waits for child to
    complete the required action */
    printf("[%s %ld] Parent about to wait for signal\n",
           currTime("%T"), (long) getpid());
    sigemptyset(&emptyMask);

    if (sigsuspend(&emptyMask) == -1 && errno != EINTR)
      errExit("sigsuspend");
    printf("[%s %ld] Parent got signal\n", currTime("%T"), (long) getpid());

    /* If required, return signal mask to its original state */
    if (sigprocmask(SIG_SETMASK, &origMask, NULL) == -1)
      errExit("sigprocmask");

    /* Parent carries on to do other things... */
    exit(EXIT_SUCCESS);
  }
}
```

Рис.2.92. Приклад використання сигналів для синхронізації дій батьківського та синівського процесів.

2.7. Сокети Берклі (Berkeley sockets)

2.7.1. Взаємодія процесів за допомогою сокетів Берклі

Сокети Берклі (Berkeley sockets) – це механізм взаємодії процесів (IPC), що, в загальному випадку, виконуються на різних обчислювальних вузлах комп'ютерної мережі. Для програміста цей механізм представлений у вигляді відповідного програмного інтерфейсу. Метою створення сокетів Берклі було спрощення організації обміну даними між обчислювальними процесами через мережу, за рахунок абстрагування від деталей обміну мережними повідомленнями транспортного рівня OSI (протокол IP).

Механізм сокетів Берклі був розроблений в Університеті Каліфорнії у Берклі (University of California at Berkeley), і був вперше реалізований у UNIX-подібній операційній системі BSD Unix 4.2 (1983 р.). Механізм виявився настільки вдалим, що з часом став де-факто стандартом мережного програмного забезпечення. На даний час цей механізм реалізований в усіх UNIX-подібних операційних системах, зокрема в ОС Linux, а також з деякими відмінностями у ОС сімейства Windows. Механізм сокетів Берклі лежить в основі переважної більшості сучасних Інтернет-технологій та використовується майже в усіх програмах, робота яких пов'язана з Інтернетом.

Слово "socket" можна дослівно перекласти, як "гніздо роз'єму". Відповідно у сокетах Берклі використовується абстракція мережних роз'ємів, за допомогою яких один процес підключається до іншого, після чого вони обмінюються даними по встановленій лінії зв'язку. Абстракція реалізує модель клієнт-сервер. Процес, який очікує на підключення, називається сервером. Процес, який підключається до іншого процесу, називається клієнтом. Після встановлення з'єднання, забезпечується повноцінний дуплексний зв'язок між клієнтом та сервером.

В схемі взаємодії процесів за допомогою сокетів Берклі (рис.2.93) паралельно виконуються процес сервера та процес клієнта. Процес сервера створює серверний сокет (socket()), призначає йому адресу (bind()), визначає довжину черги очікування на підключення (listen()) і переходить в режим очікування підключень (accept()). Процес клієнта створює клієнтській сокет і намагається підключитись до серверного сокету за вказаною адресою (connect()). Після успішного підключення, в процесі сервера створюється комунікаційний сокет (його дескриптор повертає accept()). В такий спосіб утворюється "лінія зв'язку" між сервером та клієнтом з двома "роз'ємами": комунікаційним сокетом на стороні сервера та клієнтським сокетом на стороні клієнта. В цьому з'єднанні відбувається двонаправлений обмін даними між клієнтом та сервером (send()/recv() або write()/read()). Після завершення роботи з сокетом він знищується викликом close().

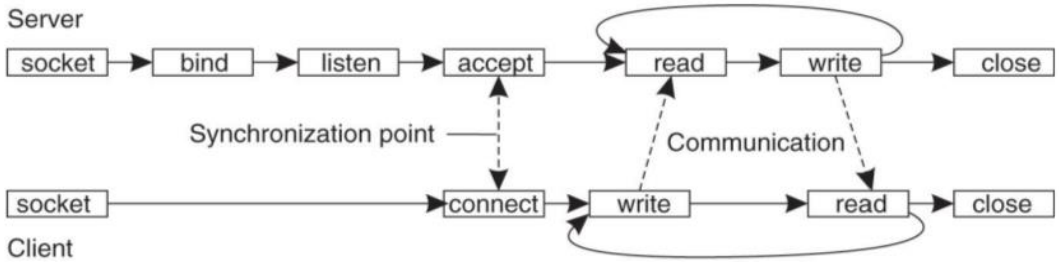

Рис.2.93. Схема взаємодії процесів за допомогою сокетів Берклі.

В механізмі сокетів Берклі підтримуються два основних режими взаємодії процесів (domains):

1) локальна взаємодія з відображенням сокетів на файлову систему (AF_UNIX або AF_LOCAL – домен системи UNIX), коли процеси клієнта та сервера виконуються локально на одному обчислювальному вузлі;

2) взаємодія процесів в мережі (AF_INET, AF_INET6 – домен Internet), коли процеси клієнта та сервера виконуються на різних обчислювальних вузлах комп'ютерної мережі (стек протоколів TCP/IP).

У випадку взаємодії процесів за допомогою сокетів в мережі використовують два основних типи сокетів:

1) віртуальний канал (stream sockets, SOCK_STREAM), коли мережна взаємодія відбувається за протоколом TCP (Transmission Control Protocol) (рис.2.94);

2) дейтаграмний зв'язок (datagram sockets, SOCK_DGRAM), коли мережна взаємодія відбувається за протоколом UDP (User Datagram Protocol) (рис.2.95).

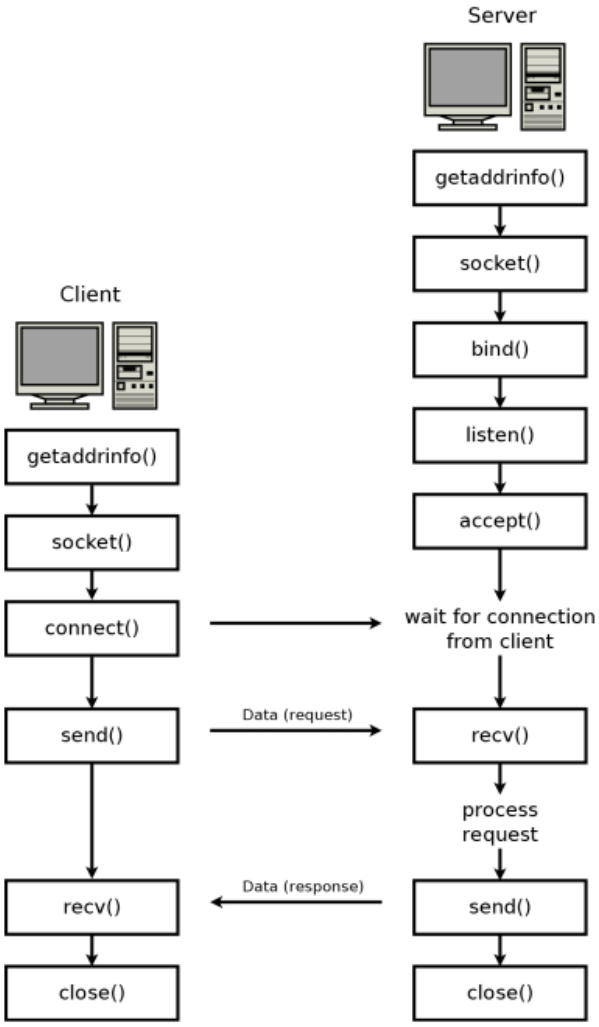

Рис.2.94. Схема взаємодії процесів за допомогою віртуального каналу (stream sockets, TCP).

2.7.2. Структура адреси сокету

Є три ситуації коли в програмах, що використовують сокети, відбувається робота з адресою сокета:

1) Призначення (прив'язка) адреси серверному сокету в процесі сервера за допомогою виклику bind(). В даному випадку потрібно, наприклад, дізнатись ip-адресу комп'ютера, на якому виконується програма, та зберегти цю інформацію у відповідному полі структури інтернет-адреси серверу.

2) Підключення до серверу зі сторони процесу клієнта за допомогою виклику connect(). В даному випадку, наприклад, потрібно коректно вказати адресу віддаленого сервера у структурі інтернет-адреси, що передається як параметр виклику connect().

3) В разі успішного підключення клієнта до сервера виклик accept() повертає дескриптор відповідного комунікаційного сокету, а у параметрі addr – інтернет-адресу клієнта, з яким встановлено зв'язок. В разі необхідності цю адресу можна проаналізувати і відповідним чином змінити хід роботи програми (наприклад, встановити той чи інший режим обміну даними з клієнтом або розірвати встановлене з'єднання).

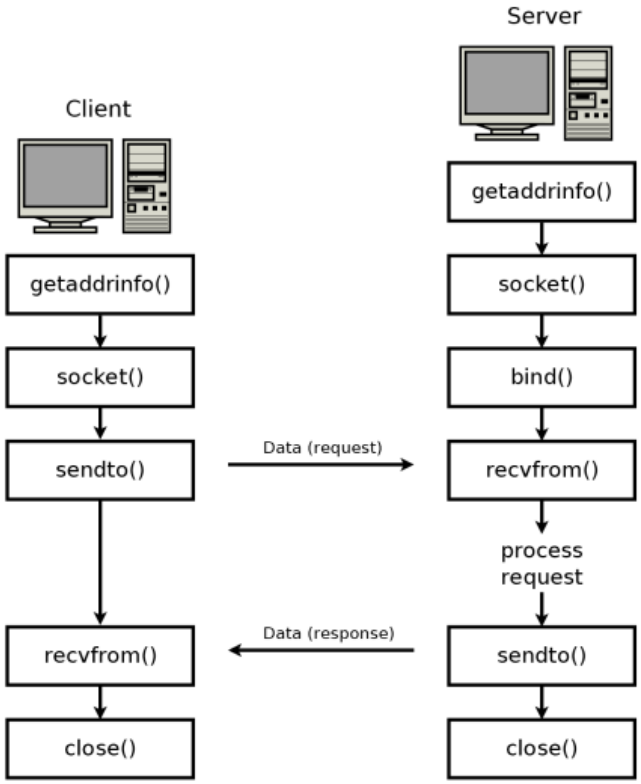

Рис.2.95. Схема взаємодії процесів за допомогою дейтаграмного зв'язку (datagram sockets, UDP).

Універсальна узагальнена структура адреси сокету sockaddr (рис.2.96) використовується для коректного визначення параметрів викликів bind(), accept(), connect(). В програмі на її місце підставляється конкретна структура адреси, яка відповідає типу сокету (локальний або мережний, IPv4 або IPv6 і т.п.). Найбільш часто це такі структури:

1) struct sockaddr_un (рис.2.97) – адреса сокету для локальної взаємодії з відображенням сокетів на файлову систему: sun_family=AF_UNIX, поле sun_PATH містить повний шлях до файлу;

2) struct sockaddr_in (рис.2.98) – адреса мережного сокету IPv4: sin_family=AF_INET, поле sin_port містить номер порту, поле sin_addr містить ip-адресу у форматі IPv4 (у вигляді структури in_addr);

3) struct sockaddr_in6 (рис.2.99) – адреса мережного сокету IPv6: sin_family=AF_INET6, поле sin6_port містить номер порту, поле sin6_addr містить ip-адресу у форматі IPv6 (у вигляді структури in6_addr).

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
```

Рис.2.96. Структура sockaddr.

```
struct sockaddr_un {
    short    sun_family;    /* AF_UNIX */
    char     sun_PATH[108]; /* path name */
};
```

Рис.2.97. Структура sockaddr_un (AF_UNIX, AF_LOCAL).

```
struct in_addr {
    unsigned long s_addr;     // that's a 32-bit int (4 bytes), load with inet_pton()
};

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
```

Рис.2.98. Структура sockaddr_in (IPv4, AF_INET).

```
struct in6_addr {
    unsigned char   s6_addr[16];   // load with inet_pton()
};

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
```

Рис.2.99. Структура sockaddr_in6 (IPv6, AF_INET6).

Для роботи зі структурою адреси мережного сокету використовуються наступні функції:

1) inet_pton() – перетворює IP-адресу в нотації цифр і крапок у структуру in_addr або структуру in6_addr залежно від того, що вказано: AF_INET або AF_INET6 ("pton" означає "presentation to network") (рис.2.100);

2) inet_ntop() – перетворює struct in_addr або struct in6_addr в IP-адресу в нотації цифр і крапок ("ntop" означає "network to presentation") (рис.2.101);

3) getpeername() – повертає мережну адресу вузла, підключеного до сокету sockfd, у буфері, на який вказує addr (рис.2.102).

4) getaddrinfo() – визначити адресу комп'ютера для використання у функціях bind() чи connect() (рис.2.103);

5) getnameinfo() – визначити назву комп'ютера за його адресою (рис.2.104).

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

Рис.2.100. Приклад використання функції inet_pton().

```
// IPv4:
char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
```

```
printf("The IPv4 address is: %s\n", ip4);


// IPv6:
char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

Рис.2.101. Приклад використання функції inet_ntop().

```
  #include <sys/socket.h>

  int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Рис.2.102. Опис функції getpeername().

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node,     // (in) e.g. "www.example.com" or IP
                const char *service,  // (in) e.g. "http" or port number
                const struct addrinfo *hints, // (in)
                struct addrinfo **res); // (out)
```

Рис.2.103. Опис функції getaddrinfo().

```
       #include <sys/socket.h>
       #include <netdb.h>

       int getnameinfo(const struct sockaddr *addr, socklen_t addrlen,
                       char *host, socklen_t hostlen,
                       char *serv, socklen_t servlen, int flags);
```

Рис.2.104. Опис функції getnameinfo().

Два основних способи використання функції getaddrinfo(): 1) дізнатися адресу вузла для створення серверного сокету в процесі сервера, який виконується на цьому вузлі (тобто у виклику bind()); 2) дізнатися адресу віддаленого вузла для підключення до нього з процесу клієнта (тобто у виклику connect()). В першому випадку в параметрі node вказується NULL. В другому випадку в параметрах node та service задаються інтернет-адреса вузла та інтернет-сервіс відповідно. Адреса повертається в параметрі res у вигляді структури addrinfo (рис.2.105). Оскільки адрес може бути декілька, res – це покажчик на масив структур addrinfo. В параметрі hints можна вказати попередньо заповнену структуру addrinfo, значення полів якої специфікують (обмежують) тип адреси, яка повернеться в параметрі res. На рис.2.106 наведено приклад використання функції getaddrinfo().

```
           struct addrinfo {
               int              ai_flags;
               int              ai_family;
               int              ai_socktype;
               int              ai_protocol;
               socklen_t        ai_addrlen;
               struct sockaddr *ai_addr;
```

```
        char            *ai_canonname;
        struct addrinfo *ai_next;
    };
```

Рис.2.105. Структура addrinfo.

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}

// servinfo now points to a linked list of 1 or more struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list
```

Рис.2.106. Приклад використання функції getaddrinfo() для визначення адреси вузла, на якому виконується програма.

2.7.3. Створення сокету

Для створення нового програмного сокету використовується функція socket() (рис.2.107). Значення параметра domain визначає домен даного сокету (AF_UNIX – UNIX-домен або AF_INET – Internet-домен), параметр type вказує тип створюваного сокета (SOCK_STREAM – віртуальний канал (TCP) або SOCK_DGRAM – дейтаграмний зв’язок (UDP)), а значення параметра protocol визначає бажаний мережний протокол. Зауважимо, що якщо значенням параметра protocol є нуль, то система сама обирає відповідний протокол для комбінації значень параметрів domain і type (це найбільш поширений спосіб використання функції socket()). Функція socket() повертає значення дескриптора сокету, яке використовується у всіх наступних функціях. Виклик функції close(sd) призводить до закриття (знищення) зазначеного сокету.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Рис.2.107. Опис функції socket().

Для призначення адреси раніше створеному сокету використовується функція bind() (рис.2.108). Параметр sockfd – це дескриптор раніше створеного сокета; addr – адреса структури, яка містить адресу сокета, що відповідає вимогам домену даного сокету (зокрема, для домену UNIX адреса – це ім'я об'єкту файлової системи, і при створенні сокету дійсно створюється файл); параметр addrlen містить довжину в байтах структури addr (цей параметр необхідний, оскільки довжина імені може суттєво відрізнятися для різних комбінацій "домен-

протокол"). Приклад використання функцій socket() та bind() для створення нового сокету наведено на рис.2.109.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Рис.2.108. Опис функції bind().

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);


// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(AF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

Рис.2.109. Приклад використання функцій socket() та bind().

Функція listen() (рис.2.110) призначена для інформування системи про те, що процес серверу планує встановлення віртуальних з'єднань через вказаний сокет. Ця функція зазвичай викликається після функцій socket і bind. Вона повинна викликатися перед викликом функції accept. Параметр sockfd – це дескриптор існуючого сокету, а значенням параметра backlog є максимальна довжина черги запитів на встановлення з'єднання, які повинні буферизуватись системою, поки їх не опрацює процес-сервер.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Рис.2.110. Опис функції listen().

2.7.4. Встановлення з’єднання

Функція accept() (рис.2.111) використовується для отримання на опрацювання процесом сервера чергового підключення до вказаного серверного сокету. Параметр sockfd задає дескриптор серверного сокету, для якого раніше була виконана функція listen(); параметр addr вказує на структуру з адресою віддаленого клієнта, підключення якого відбулося; addrlen –

адреса, за якою знаходиться довжина структури addr. Якщо на момент виклику функції accept() черга запитів на підключення порожня, то виконання процесу сервера призупиняється (блокується) до надходження запиту. Виконання функції accept() призводить до встановлення віртуального з'єднання. Функція accept() повертає дескриптор комунікаційного сокету, який в подальшому використовується на стороні сервера для взаємодії з клієнтом для обміну даними по встановленому з'єднанню. Як правило, виклик accept() розміщують у нескінченному циклі "детектування" нових підключень, а для кожного чергового з'єднання створюють новий програмний потік, в який передають дескриптор комунікаційного сокету. Функція accept4() працює так само як accept() з тою відмінністю, що у параметрі flags можна вказати прапорці, які задають той чи інший режим роботи функції. Наприклад, за допомогою прапорця SOCK_NONBLOCK роботу accept4() можна перевести в асинхронний (неблокуючий) режим.

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

int accept4(int sockfd, struct sockaddr *addr,
            socklen_t *addrlen, int flags);
```

Рис.2.111. Опис функцій accept() та accept4().

За допомогою функції connect() (рис.2.112) процес клієнта скеровує системі запит на підключення до віддаленого вузла з вказаною адресою (тобто до серверного сокету в процесі сервера). Параметри мають те саме значення, що у функції bind(), проте в якості адреси addr вказується адреса серверного сокету (віддаленого вузла), який знаходиться на іншій стороні каналу зв'язку. Для нормального виконання функції необхідно, щоб у сокету з дескриптором sockfd і у серверного сокету з адресою addr були однакові домен і протокол. Якщо тип сокету з дескриптором sockfd є дейтаграмним (UDP), то функція connect() служить лише для інформування системи про адресу призначення пакетів, які в подальшому будуть надсилатися за допомогою функції send(); ніякі дії по встановленню з'єднання в цьому випадку не виконуються. В разі успіху, коли з'єднання встановлено, функція connect() повертає нуль. В разі помилки connect() повертає -1 (змінна errno містить код помилки). При використанні функції connect(), як правило, потрібно передбачити спроби повторного з'єднання.

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Рис.2.112. Опис функції connect().

2.7.5. Відправлення та отримання повідомлень

Для відправлення та отримання даних через сокети зі встановленим віртуальним з'єднанням (SOCK_STREAM) використовуються функції send() і recv() (рис.2.113). В функції send() параметр sockfd задає дескриптор існуючого сокету з встановленим з'єднанням; параметр buf вказує на буфер з даними, які потрібно відправити через канал зв'язку; параметр len задає довжину цього буфера; параметр flags містить прапорці. За допомогою прапорця MSG_OOB можна відправити дані у канал в режимі "позачергового" (out-of-band) відправлення. В такому випадку дані надсилаються окремо, "обганяючи" усі раніше непрочитані з каналу дані. Потенційний отримувач даних може отримати спеціальний сигнал і в ході його обробки негайно прочитати "позачергові" дані. Функція send() повертає кількість

реально відправлених байтів даних, яка в нормальних ситуаціях збігається зі значенням параметру len.

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Рис.2.113. Опис функцій send() та recv().

В функції recv() параметр sockfd задає дескриптор сокету з встановленим з'єднанням; параметр buf вказує на буфер, в який слід помістити отримані дані; параметр len задає максимальну довжину цього буфера. Вказавши у параметрі flags прапорець MSG_PEEK, можна прочитати дані з системного буферу (socket receive bufer) в користувацький буфер buf без їх видалення з системного буферу. Функція recv() повертає кількість реально отриманих в buf байтів даних.

Зауважимо, що в разі використання сокетів з віртуальним з'єднанням замість функцій send і recv можна використовувати звичайні файлові системні виклики read() і write(). Для сокетів цього типу вони виконуються абсолютно аналогічно функціям send() і recv(). Це дозволяє створювати програми, які не залежать від того, чи працюють вони зі звичайними файлами, каналами (pipes, named pipes) або сокетами.

Для відправлення та отримання даних в дейтаграмному режимі (SOCK_DGRAM) використовуються функції sendto() та recvfrom() (рис.2.114). Параметри sockfd, buf і len аналогічні за змістом до відповідних параметрів функцій send() і recv(). Параметри dest_addr і addrlen функції sendto() задають адресу серверного сокету (віддаленого вузла), якому відправляються дані, і можуть бути опущені, якщо до цього викликалася функція connect(). Параметри src_addr і addrlen функції recvfrom() дозволяють процесу сервера отримати адресу вузла, від якого отримані дані, надіслані процесом клієнта.

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Рис.2.114. Опис функцій sendto() та recvfrom().

2.7.6. Розрив з’єднання та знищення сокету

Після завершення роботи з сокетом, потрібно закрити відповідний дескриптор за допомогою функції close(), тим самим повідомивши систему про звільнення цього системного ресурсу. Тобто викликом close(sfd) ми закриваємо і знищуємо сокет з дескриптором sfd. Відповідне мережне з’єднання розривається. У випадку віртуального каналу (stream sockets, TCP) перед знищенням сокету система спробує відправити дані, які вже поставлені в чергу, і після того як відправлення відбудеться, здійснить нормальну послідовність дій по завершенню TCP-з'єднання.

Для негайної ліквідації (розриву) встановленого з'єднання без знищення сокету використовується функція shutdown() (рис.2.115). Виклик shutdown() дозволяє негайно зупинити обмін даними в одному з трьох режимів, які задаються значенням параметра how:

1) SHUT_RD – забороняє приймання даних через сокет,

2) SHUT_WR – забороняє відправлення даних через сокет,

3) SHUT_RDWR – забороняє приймання та відправлення даних через сокет.

Дія функції shutdown() відрізняється від дії функції close() тим, що, по-перше, виконання останньої "затримується" до закінчення спроб системи завершити доставку вже відправлених даних. По-друге, функція shutdown() розриває з'єднання, але не ліквідує дескриптори раніше з'єднаних сокетів. Для їх остаточного знищення все одно потрібно викликати функцію close().

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

Рис.2.115. Опис функції shutdown().

Контрольні питання

1. В чому полягає взаємодія обчислювальних процесів (Inter-Process Communication) в ОС Linux?
2. Які механізми взаємодії обчислювальних процесів використовуються в ОС Linux?
3. В який спосіб організована взаємодія процесів з використанням неіменованого каналу (anonymous pipe)?
4. Чому неіменований канал (anonymous pipe) називається “неіменованим”?
5. Чим іменований канал (named pipe) відрізняється від неіменованого (anonymous pipe)?
6. Який режим використання каналів (з блокуванням чи без блокування) встановлено за замовчуванням?
7. Яке значення повертає системний виклик pipe()?
8. Який системний виклик використовується для для одночасного запуску синівського процесу та створення неіменованого каналу для зв’язку з ним?
9. В якому режимі іменування об’єктів System V IPC об’єкт IPC використовується лише батьківським процесом та породженими ним синівськими процесами?
10. В який спосіб організована взаємодія процесів з використанням черги повідомлень (message queue)?
11. Які системні виклики використовуються для відправлення та отримання повідомлень через чергу повідомлень?
12. Чим черга повідомлень (message queue) відрізняється від каналів (pipes)?
13. Яка команда використовується в системному виклику msgctl() для видалення з системи черги повідомлень?
14. В який спосіб організована взаємодія процесів з використанням спільної пам’яті (shared memory)?
15. Які системні виклики і в якій послідовності використовуються для роботи зі спільною пам’яттю (shared memory)?
16. Для чого використовується механізм відображення файлів у пам’ять в ОС Linux в контексті організації взаємодії обчислювальних процесів?
17. Яка функція використовується для відображення вказаного файлу в адресний простір процесу в ОС Linux?
18. Для чого призначений механізм семафорів (semaphores)?
19. В чому різниця між класичним семафором та його реалізацією в ОС Linux?
20. Який системний виклик призначений для здійснення операції над одним чи декількома семафорами з множини семафорів?
21. Який прапорець потрібно вказати в полі структури sem_command.sem_flg для того, щоб операція над семафором: semop(semID, &sem_command, 1) була виконана в асинхронному режимі (без блокування)?
22. Що таке сигнали (signals) в ОС Linux і для чого вони використовуються?
23. В який спосіб встановлюється диспозиція сигналу (disposition of the signal) в ОС Linux?
24. Що потрібно вказати у першому параметрі системного виклику kill(pid, sig) для того, щоб надіслати сигнал всім процесам у групі процесів, в яку входить даний процес?
25. Чим завершується системний виклик kill(pid, sig), якщо параметр sig дорівнює нулю?
26. Яка функція в ОС Linux призначена для того, щоб обчислювальний процес міг надіслати сигнал сам собі?

27. Які варіанти диспозиції сигналу можна встановити в ОС Linux системним викликом signal() або sigaction()?
28. Що потрібно вказати першим параметром системного виклику sigprocmask(how, &set, &oldset), щоб сигнали, зазначені в наборі сигналів set, додалися до маски сигналів?
29. Для чого використовується маскування сигналів в ОС Linux?
30. Для чого призначений механізм сокетів Берклі (Berkeley sockets)?
31. Який тип сокетів Берклі використовується для організації мережної взаємодії процесів за протоколом TCP?
32. Для чого використовується системний виклик socket() і яке значення від повертає?
33. Чим відрізняються серверний, комунікаційний та клієнтський сокети?
34. Чим відрізняється взаємодія процесів з використанням сокетів в режимі віртуального каналу (SOCK_STREAM), режимі дейтаграмного зв’язку (SOCK_DGRAM) та у режимі локальної взаємодії (AF_LOCAL)?
35. Який системний виклик ОС Linux використовується для інформування системи про те, що процес серверу планує встановлення віртуальних з'єднань через вказаний сокет із заданою максимальною довжиною черги запитів на встановлення з'єднання?
36. В якому порядку використовуються системні виклики та функції при роботі з серверним сокетом в режимі віртуального каналу (stream sockets)?
37. В якому порядку використовуються системні виклики та функції при роботі з клієнтським сокетом в режимі віртуального каналу (stream sockets)?

3. Програмні потоки в ОС Linux

3.1. Програмні потоки (threads)

3.1.1. Паралельні обчислення на рівні програмних потоків

Як і процеси, програмні потоки (threads) – це механізм, який дозволяє виконувати одночасно декілька обчислювальних завдань. Один процес може містити кілька потоків (рис.3.1). Усі потоки незалежно виконують одну і ту ж програму, а також мають однакову глобальну пам'ять, включаючи ініціалізовані дані, неініціалізовані дані та область пам'яті, що виділяється динамічно (heap area) (рис.3.2).

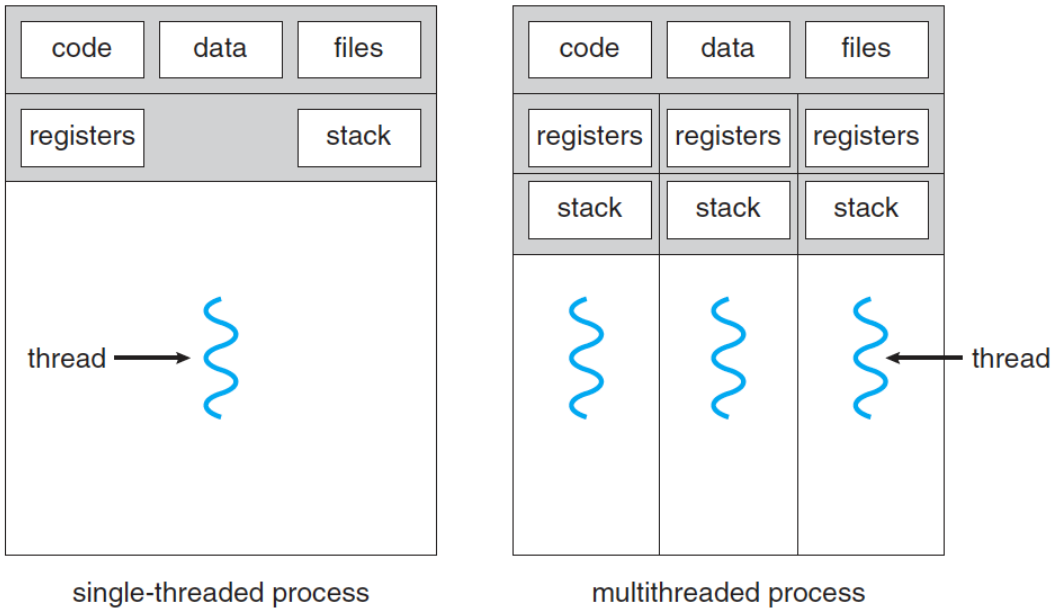

Рис.3.1. Розпаралелювання на рівні програмних потоків.

Програмні потоки - це одиниці розпаралелювання в межах одного обчислювального процесу. Іншими словами, програмні потоки – це частини коду одного процесу, які виконуються паралельно в його контексті. Разом з процесом створюється основний (первинний) потік (initial/primary/main thread) (рис.3.3). Усі потоки мають однакові права щодо доступу до віртуальної пам'яті процесу. Також всі ресурси, виділені системою процесу, є спільними для його програмних потоків.

У кожного потоку є свій регістровий контекст (вміст регістрів процесора в тому числі лічильник команд і вказівник стеку) та свій стек (який однак не «захищений» від вільного доступу до нього інших потоків). Також кожний потік має унікальний ідентифікатор (унікальний в межах процесу) та свій пріоритет. Майже все інше у програмних потоків одного обчислювального процесу спільне.

Особливості використання програмних потоків:

1) при використанні потоків зменшуються витрати на переключення контексту (context switch);

2) створення нового потоку займає значно менше часу ніж створення нового процесу;

3) всі потоки одного процесу виконуються у спільному адресному просторі віртуальної пам'яті процесу;

4) потоки можуть створювати інші потоки в рамках цього ж процесу;

5) потоки можуть також призупинити, відновити чи завершити будь-які інші потоки у своєму процесі.

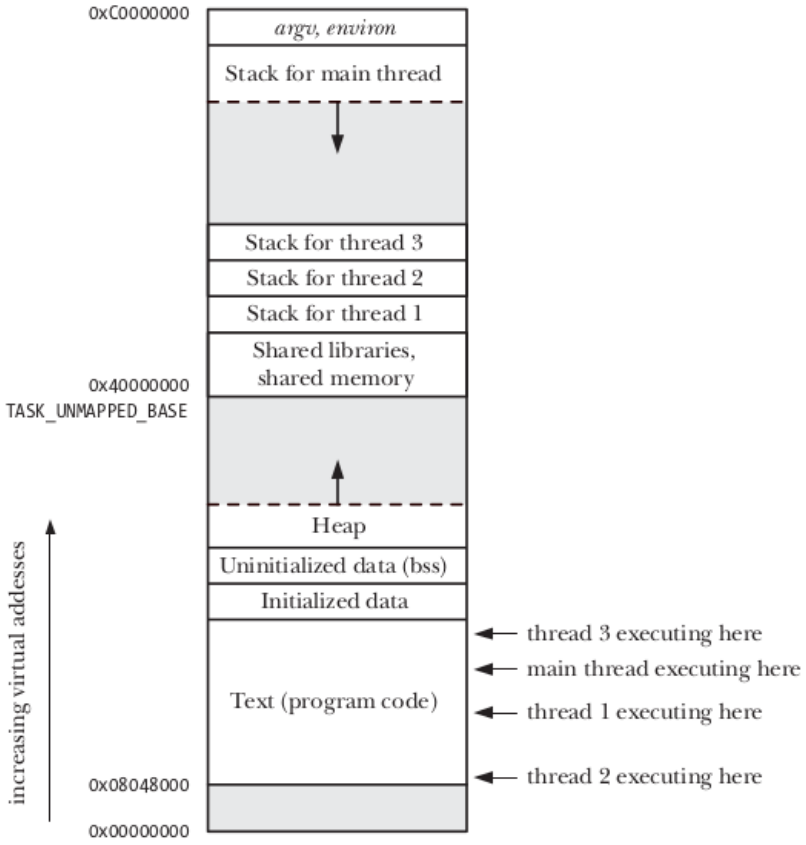

Рис.3.2. Розподіл пам’яті процесу між програмними потоками.

З точки зору реалізації програмних потоків на системному рівні розрізняють три моделі відображення потоків рівня користувача у потоки рівня ядра: Many-to-One Model, One-to-One Model та Many-to-Many Model. Розглянемо ці моделі.

1) В моделі Many-to-One Model (User level threads) всі потоки рівня користувача відображаються в один потік рівня ядра. Управління потоками виконує стороння бібліотека в режимі користувача, яка реалізує тільки багатозадачність з розділенням у часі. Відсутні обмеження на кількість потоків. Переключення контексту потоків займає менше часу в порівнянні з іншими моделями. Приклади реалізації цієї моделі: Green threads (scheduled by VM) та GNU Portable threads.

2) В моделі One-to-One Model (Kernel level threads) кожний потік рівня користувача відображається в один потік рівня ядра (рис.3.4). ОС накладає обмеження на максимальну кількість потоків в одному процесі. Переключення контексту потоків займає більше часу ніж в Many-to-One Model. Ця модель реалізована в ОС Linux та в ОС сімейства Windows.

3) В моделі Many-to-Many Model (Hybrid threads) N потоків рівня користувача відображаються в $X \leq N$ потоків рівня ядра. Перша мета використання цієї моделі - знайти компроміс між перевагами над-швидкого переключення контексту (user threads) та перевагами фізичного («просторового») розпаралелювання (kernel threads). Друга мета використання цієї моделі – зберегти перевагу необмеженої кількості потоків (необмежена кількість user threads відображається у обмежену кількість kernel threads). В дворівневій моделі Many-to-Many реалізована додаткова можливість прив’язати окремий потік рівня користувача до одного потоку рівня ядра. Дворівнева модель підтримується в ОС HP-UX, ОС Solaris (до версії 9).

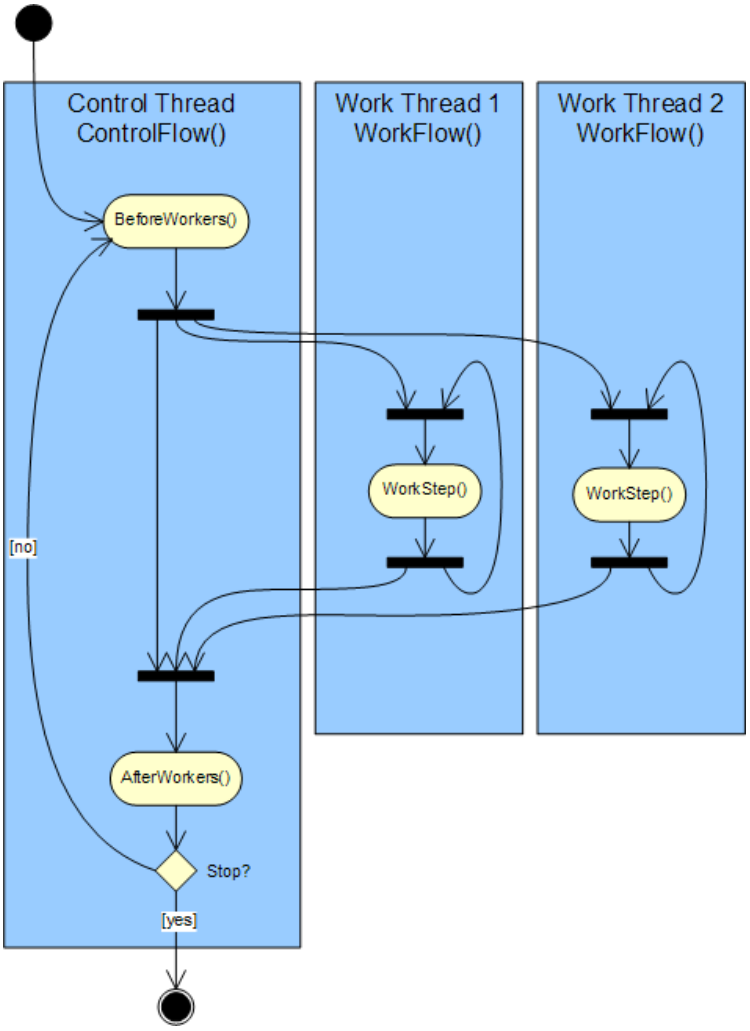

Рис.3.3. Приклад розпаралелювання на рівні програмних потоків.

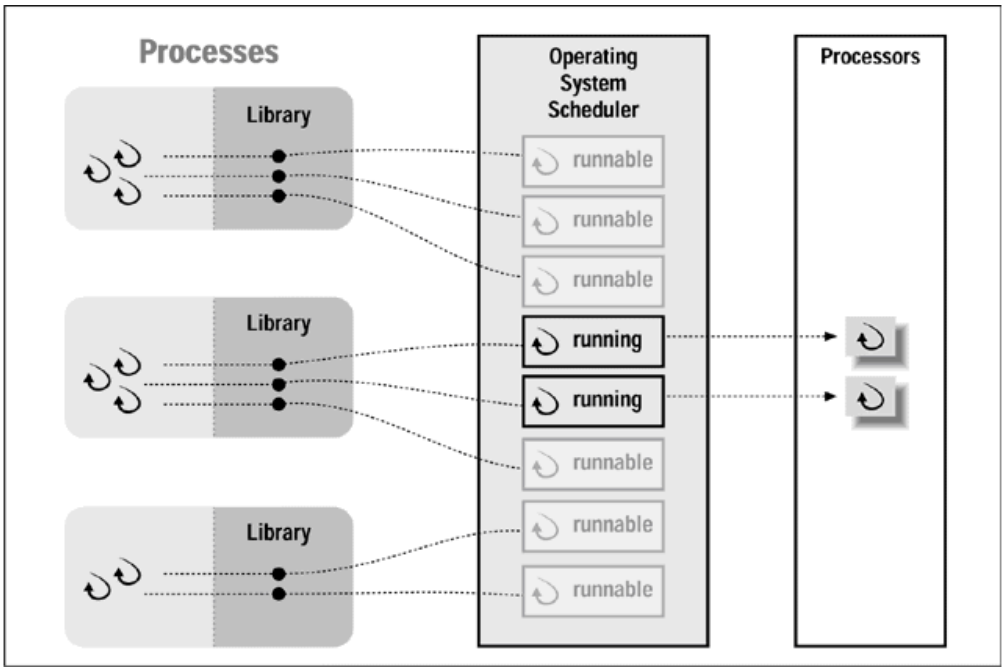

Рис.3.4. Приклад реалізації моделі One-to-One.

3.1.2. Переваги та недоліки використання програмних потоків

До переваг використання програмних потоків можна віднести:

1) Зменшення витрат на переключення контексту між програмними потоками у порівнянні з переключенням контексту між процесами. Особливої ваги це набуває в умовах обмеженої доступності процесора, коли він, наприклад, перезавантажений обчисленнями.

2) Значно менший час створення нового програмного потоку у порівнянні зі створенням нового обчислювального процесу.

3) Проста схема інформаційної взаємодії між потоками (не потрібні складні механізми між-процесної взаємодії (IPC)). Спільний доступ до даних між потоками є простим. Навпаки, обмін даними між процесами вимагає більшої роботи (наприклад, створення сегмента спільної пам'яті або використання каналів (pipes)).

4) З'являються широкі можливості щодо підвищення продуктивності паралельної програми за рахунок більшої "свободи" в організації одночасного виконання та взаємодії програмних потоків (більша гнучкість організації паралельних обчислень).

До недоліків використання програмних потоків можна віднести:

1) Завжди існує загроза того, що потоки почнуть некоректно працювати зі спільною пам'яттю, помилково змінюючи чи знищуючи потрібні дані.

2) Потоки потрібно спеціальним чином синхронізувати, якщо вони працюють зі спільними структурами даних (наприклад, за допомогою м'ютексів).

3) Кожен з потоків може ліквідувати цілий процес (або помилковим знищенням основного потоку, або своїм «аварійним» завершенням).

4) Помилка в одному потоці (наприклад, модифікація даних з використанням неправильного покажчика) може пошкодити всі потоки в процесі, оскільки вони мають однаковий адресний простір та інші атрибути. Навпаки, процеси більш ізольовані один від одного.

5) Під час програмування за допомогою потоків потрібно слідкувати, щоб функції, які викликаються в потоках, були потоко-безпечні (thread-safe) або викликались у потоко-безпечний спосіб. При розпаралелюванні на рівні процесів такої проблеми немає.

6) Кожен потік конкурує за використання віртуального адресного простору процесу з іншими потоками. Зокрема, стек кожного потоку та його локальні дані споживають частину віртуального адресного простору процесу, яка, відтак, недоступна для інших потоків. Хоча, як правило, доступний віртуальний адресний простір в сучасних системах є достатньо великим, цей фактор може бути суттєвим обмеженням для процесів, що використовують велику кількість потоків або потоки, що потребують великих об'ємів пам'яті. На відміну від цього, окремий процес може використовувати весь діапазон доступної віртуальної пам'яті.

3.2. Інтерфейс прикладного програмування POSIX Threads (Pthreads)

3.2.1. Використання POSIX Threads

POSIX Threads (Pthreads) – це стандарт POSIX, який визначає програмний інтерфейс для створення та використання програмних потоків [15]. Існує кілька реалізацій цього стандарту, зокрема

1) LinuxThreads – перша реалізація Pthreads в ОС Linux; починаючи з glibc 2.4 не підтримується.

2) Native POSIX Thread Library (NPTL) – сучасна реалізація Pthreads в ОС Linux, яка у порівнянні з LinuxThreads більш повно відповідає стандарту POSIX Threads.

3) GNU Portable Threads (GNU Pth);

4) Open Source POSIX Threads for Win32 (pthreads-w32) та ін.

В ОС Linux за допомогою команди getconf можна визначити реалізацію POSIX Threads та її версію, наприклад:

```
$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.28
```

Pthreads визначає набір типів, функцій та констант для роботи з програмними потоками на мові програмування С. Для роботи з Pthreads необхідно мати файл pthread.h та відповідну бібліотеку функцій. Один процес може містити кілька потоків, усі вони виконують одну і ту ж програму. Ці потоки мають однакову глобальну пам’ять (сегменти даних та область пам’яті, що виділяється динамічно), але кожен потік має власний стек.

Стандартом Pthreads визначено більше 70-ти функцій, які за призначенням можна поділити на наступні основні групи (в дужках вказані префікси у назвах відповідних функцій):

1) Функції управління потоками (pthread_);

2) Функції управління атрибутами потоків(pthread_attr_, pthread_get<paramname>, pthread_set<paramname>);

3) Функції управління м’ютексами (pthread_mutex_, pthread_mutexattr_);

4) Функції управління умовними змінними (pthread_cond_, pthread_condattr_);

5) Функції бар’єрної синхронізації (pthread_barrier_, pthread_barrierattr_);

6) Функції блокування читання-запису (pthread_rwlock_, pthread_rwlockattr_)

7) Функції спін-блокування (spinlock) (pthread_spin_).

Більшість функцій Pthreads повертають 0 в разі успіху або номер помилки в протилежному випадку. Функції pthreads не встановлюють значення змінної помилки errno.

Важливим питанням при програмуванні на рівні потоків є використання потоко-безпечних функцій (thread-safe functions). Потоко-безпечна функція – це функція, яку безпечно викликати у декількох потоках одночасно (тобто вона буде давати однакові результати незалежно від того, в якому потоці вона була викликана). Стандарти POSIX.1-2001 та POSIX.1-2008 визначають перелік стандартних функцій, які мають бути потоко-безпечними.

3.2.2. Створення та ідентифікація програмних потоків

Для створення нового програмного потоку використовується функція pthread_create() (рис.3.5). В параметрі thread (покажчик на ідентифікатор потоку типу pthread_t) повертається ідентифікатор створеного потоку; параметр attr – це вказівник на атрибутний об’єкт (змінна типу pthread_attr_t), яким задаються атрибути нового потоку; параметр start_routine – це назва початкової функції, яку буде виконувати потік; параметр arg – це вказівник на аргумент (типу void), який передається функції start_routine.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Рис.3.5. Опис функції pthread_create().

В разі успішного виконання функція pthread_create() повертає 0, а в разі помилки повертає не нульове значення, яке вказує на тип помилки. Також в разі успішного виконання через параметр thread функція поверне ідентифікатор створеного потоку. За допомогою другого параметру attr можна змінити атрибути нового потоку, що встановлені по замовченню. Як правило, замість цього другим аргументом вказують NULL, користуючись атрибутами потоку, які встановлені по замовченню. Щойно створений потік починає виконання з функції start_routine. Ця функція приймає єдиний аргумент, arg, який є нетипізованим вказівником. Якщо необхідно передати функції start_routine значний обсяг інформації, то її слід зберегти у вигляді структури й передати вказівник на структуру в аргументі arg. При створенні нового потоку не можна заздалегідь припускати, хто першим отримає керування – щойно створений потік або потік, що викликав функцію pthread_create.

Згідно стандарту POSIX.1 програмні потоки розділяють (мають однакові значення) наступних атрибутів процесу:

- ідентифікатор процесу (process ID),
- ідентифікатор батьківського процесу (parent process ID);
- ідентифікатори групи процесів та сесії (process group ID and session ID);
- термінал запуску процесу (controlling terminal);
- ідентифікатори користувача та групи користувачів (user and group IDs);
- дескриптори відкритих файлів (open file descriptors);
- замки на записи (record locks);
- диспозиції сигналів (signal dispositions);
- маска режиму створення файлу (file mode creation mask);
- поточна і коренева директорія (current and root directory);
- інтервальні таймери (interval timers) і таймери POSIX (POSIX timers);
- значення поправки до статичного пріоритету (nice value);
- ресурсні обмеження (resource limits);
- виміри споживання процесорного часу та системних ресурсів (measurements of the consumption of CPU time and resources).

Разом зі стеком потоку, стандарт POSIX.1 визначає наступні унікальні для кожного програмного потоку атрибути:

- ідентифікатор потоку (thread ID);
- маска сигналів (signal mask);
- змінна помилки (the errno variable);
- альтернативний стек сигналів (alternate signal stack);
- алгоритм диспетчеризації в режимі реального часу та відповідний пріоритет (real-time scheduling policy and priority);
- можливості здійснення операцій з точки зору прав доступу (capabilities);
- прив’язка до процесора або ядер процесора (CPU affinity).

На рис.3.6. наведено приклад створення програмних потоків управління. В програмі створюються п’ять нових потоків (окрім основного потоку еквівалентного самому процесу) за допомогою функції pthread_create(). Кожний зі створених потоків виводить повідомлення «Hello World!» та свій ідентифікатор (функція PrintHello()).

```
// This program creates five new threads,
// each of which prints its thread number to standard output

#include <pthread.h>
```

```
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;

   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;

   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);

      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Рис.3.6. Приклад створення п’яти потоків, кожний з яких виводить свій номер.

Кожен з потоків у процесі має свій унікальний ідентифікатор (зберігається у змінній типу pthread_t). Цей ідентифікатор повертається викликом pthread_create(), і, крім цього, потік може отримати власний ідентифікатор викликом функції pthread_self(). Унікальність ідентифікаторів потоків гарантується лише в межах відповідного процесу. У всіх функціях Pthreads, де в якості параметру використовується ідентифікатор, він відноситься до потоку в цьому ж процесі. Система може повторно використовувати ідентифікатор потоку після приєднання завершеного потоку або завершення від’єднаного потоку. В стандарті POSIX зазначено: «Якщо програма намагається використати ідентифікатор потоку, термін дії якого закінчився, то відповідна поведінка не визначена».

Функція pthread_self() (рис.3.7) повертає викликаючому програмному потоку його ідентифікатор. Ця функція є прямим аналогом функції getpid() для процесів. За допомогою функції pthread_equal() (рис.3.8) можна перевірити рівність двох заданих ідентифікаторів потоків (t1 і t2). Якщо ідентифікатори рівні, то функція pthread_equal() поверне ненульове значення, інакше поверне 0.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Рис.3.7. Опис функції pthread_self().

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Рис.3.8. Опис функції pthread_equal().

3.2.3. Завершення виконання програмних потоків

Створений програмний потік може завершити своє виконання одним з наступних способів:

1) викликом функції pthread_exit();

2) звичайним завершенням стартової функції <start_routine>;

3) завершитись у відповідь на запит на завершення, отриманий від іншого потоку (функція pthread_cancel());

4) викликом функції exit() в будь-якому потоці, який також зупинить сам процес разом з усіма іншими потоками.

Функція pthread_exit() (рис.3.9) завершує виконання викликаючого програмного потоку. Ця функція ніколи не повертає керування. В параметрі retval можна передати код завершення потоку, який буде отриманий іншим потоком за допомогою функції очікування на завершення потоку pthread_join().

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Рис.3.9. Опис функції pthread_exit().

Функція pthread_cancel() (рис.3.10) надсилає запит на завершення вказаному потоку (thread). Відправивши запит на завершення потоку, виклик pthread_cancel() негайно завершується, не чекаючи на завершення вказаного потоку (thread). Як потік відреагує на цей запит (коли і як він завершиться), залежить від стану завершення (cancellation state) та типу завершення (cancellation type), що встановлені для цього потоку.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Рис.3.10. Опис функції pthread_cancel().

Функція pthread_testcancel() (рис.3.11) перевіряє наявність запиту на завершення, який знаходиться в черзі очікування на надходження (pending cancellation request). Виклик pthread_testcancel() створює точку завершення (cancellation point) у викликаючому потоці, внаслідок чого потік, який до цього часу виконував код без точок завершення, відповість на запит на завершення у разі його наявності.

Якщо можливість завершення по запиту відключена (за допомогою функції pthread_setcancelstate()), або запит на завершення відсутній, то виклик phread_testcancel() не має ефекту.

```
#include <pthread.h>

void pthread_testcancel(void);
```

Рис.3.11. Опис функції pthread_testcancel().

За допомогою функції pthread_setcancelstate() (рис.3.12) можна встановити один з двох станів реагування на надходження запитів на завершення:

1) PTHREAD_CANCEL_DISABLE – в цьому стані потік не реагує на запити на завершення; якщо такі запити надходять, то вони ставляться в чергу очікування на надходження;

2) PTHREAD_CANCEL_ENABLE – в цьому стані потік реагує на запити на завершення (цей стан встановлено за замовчуванням).

Стан, який потрібно встановити, задається в параметрі state. Попередній стан повертається в параметрі oldstate. Тимчасове відключення реагування на запити на завершення (PTHREAD_CANCEL_DISABLE) корисно, коли потік виконує ділянку коду, в якій повинні бути виконані всі кроки.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Рис.3.12. Опис функції pthread_setcancelstate().

За допомогою функції pthread_setcanceltype() (рис.3.13) можна встановити один з двох типів реагування на запит на завершення (для стану PTHREAD_CANCEL_ENABLE):

1) PTHREAD_CANCEL_ASYNCHRONOUS - у відповідь на запит, потік завершується в будь який момент часу, в тому числі негайно після надходження запиту на завершення;

2) PTHREAD_CANCEL_DEFERRED - у відповідь на запит, потік відкладає своє завершення до точки завершення (cancellation point) (цей тип реагування на запит встановлено за замовчуванням).

Тип реагування, який потрібно встановити, задається в параметрі type. Попередній тип реагування повертається в параметрі oldtype. Точка завершення (cancellation point) - це виклик будь-якої стандартної функції з набору функцій, визначених реалізацією стандарту POSIX Threads.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Рис.3.13. Опис функції pthread_setcanceltype().

3.2.4. Очікування на завершення програмних потоків

Для очікування на завершення іншого потоку використовується функція pthread_join() (рис.3.14). Функція pthread_join() призупиняє виконання викликаючого потоку до моменту завершення виконання іншого потоку з ідентифікатором thread. Якщо покажчик retval не нульовий, то через нього повертається код завершення відповідного потоку. Якщо потік був примусово завершений (у відповідь на запит на завершення), то за адресою retval буде записано значення PTHREAD_CANCELED. Усі потоки в процесі є одноранговими: будь-який потік може приєднатись (join) до будь-якого іншого потоку в процесі. Функція pthread_join() є прямим аналогом функції waitpid() для процесів.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Рис.3.14. Опис функції pthread_join().

Функція pthread_detach() (рис.3.15) робить вказаний потік від’єднаним, тобто таким на завершення якого не може очікувати жоден інший потік (викликом pthread_join). Від’єднаний потік є віддаленим аналогом процеса-демона, який виконується у фоновому режимі. Після «самостійного» завершення від’єднаного потоку всі захоплені ним системні ресурси автоматично звільняються. По замовченню усі потоки створюються як приєднані. При необхідності змінити власний статус потоку параметром функції pthread_detach() вказується функція pthread_self(): pthread_detach(pthread_self()).

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Рис.9.15. Опис функції pthread_detach().

Якщо нас не цікавить момент часу і код завершення деякого потоку, ми можемо звернутися до функції pthread_detach(), щоб дозволити операційній системі утилізувати ресурси, захоплені цим потоком, після його завершення. Будь-який потік в кінці кінців в момент завершення має “виходити” на pthread_join() чи pthread_detach(). Виклик pthread_detach() може використовуватись для виключення тупикових ситуацій (deadlocks), коли, наприклад, два потоки очікують на завершення один одного.

3.2.5. Управління атрибутами програмних потоків

Атрибути потоку задаються у вигляді атрибутного об’єкту (змінна типу pthread_attr_t). Атрибутний об’єкт представляє собою структуру (рис.3.16), яка дозволяє зберігати наступні атрибути потоку:

1) stackaddr – місце розташування стеку потоку (за замовчуванням адресу стеку обирає система);

2) stacksize – розмір стеку потоку (за замовчуванням новий потік має встановлене у системі максимальне значення розміру стеку);

3) detachstate - тип потоку: від’єднаний чи приєднаний (за замовчуванням потік є приєднаним – PTHREAD_CREATE_JOINABLE);

4) inheritsched – режим наслідування параметрів диспетчеризації (за замовчуванням потік наслідує параметри диспетчеризації батьківського потоку – PTHREAD_INHERIT_SCHED);

5) scope – область конкуренції потоку за час процесора (contention scope);

6) schedpolicy – стратегія диспетчеризації (планування паралельного виконання потоку, значення за замовчуванням – SCHED_OTHER);

7) priority – пріоритет потоку (за замовчуванням потік наслідує пріоритет основного потоку свого процесу).

```
typedef struct __pthread_attr pthread_attr_t;

struct __pthread_attr
{
  struct sched_param __schedparam;
  void *__stackaddr;
  size_t __stacksize;
  size_t __guardsize;
  enum __pthread_detachstate __detachstate;
  enum __pthread_inheritsched __inheritsched;
  enum __pthread_contentionscope __contentionscope;
  int __schedpolicy;
};
```

Рис.3.16. Структура pthread_attr_t.

Реалізація стандарту POSIX Threads в системі ОС Linux (Native POSIX Thread Library, NPTL) за замовчуванням визначає значення атрибуту stacksize рівним встановленому у системі максимальному значенню розміру стеку ("stack size" resource limit). Це значення можна дізнатись командою ulimit:

```
$ ulimit -s
8192
```

В наведеному прикладі це значення складає 8M (0x800000 байт).

Атрибут scope – область конкуренції потоку за час процесора (contention scope) – показує спосіб відображення потоку рівня користувача у потік рівня ядра (модель Many-to-One або модель One-to-One) і може приймати одне з двох значень:

1) PTHREAD_SCOPE_PROCESS – локальна область конкуренції в межах процесу, коли потоки одного процесу конкурують за доступ до процесора лише між собою (модель Many-to-One);

2) PTHREAD_SCOPE_SYSTEM – глобальна область конкуренції в межах всієї системи, коли потоки одного процесу конкурують за доступ до процесора з усіма іншими потоками в системі (модель One-to-One).

Значення атрибуту scope визначається типом моделі відображення потоку рівня користувача у потік рівня ядра, яка реалізована у відповідній бібліотеці функцій потоків. Зокрема в Native POSIX Thread Library (NPTL) реалізована модель One-to-One. Відповідно scope має значення PTHREAD_SCOPE_SYSTEM.

Для початкової ініціалізації та знищення атрибутного об’єкту використовуються функції int pthread_attr_init() та pthread_attr_destroy() (рис.3.17)

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Рис.3.17. Опис функцій pthread_attr_init() та pthread_attr_destroy().

Після ініціалізації атрибутного об’єкту значення окремих атрибутів можна
1) подивитися за допомогою функцій типу pthread_attr_get<attr_name>(), та
2) встановити за допомогою функцій типу pthread_attr_set<attr_name>().
Для отримання значень атрибутів потоку використовуються наступні функції:

```
int pthread_attr_getdetachstate(const pthread_attr_t *, int *);
int pthread_attr_getguardsize(const pthread_attr_t *, size_t *);
int pthread_attr_getinheritsched(const pthread_attr_t *, int *);
int pthread_attr_getschedparam(const pthread_attr_t *, struct sched_param *);
int pthread_attr_getschedpolicy(const pthread_attr_t *, int *);
int pthread_attr_getscope(const pthread_attr_t *, int *);
int pthread_attr_getstackaddr(const pthread_attr_t *, void **);
int pthread_attr_getstacksize(const pthread_attr_t *, size_t *);
```

Для встановлення значень атрибутів потоку використовуються наступні функції:

```
int pthread_attr_setdetachstate(pthread_attr_t *, int);
int pthread_attr_setguardsize(pthread_attr_t *, size_t);
int pthread_attr_setinheritsched(pthread_attr_t *, int);
int pthread_attr_setschedparam(pthread_attr_t *,const struct sched_param *);
int pthread_attr_setschedpolicy(pthread_attr_t *, int);
int pthread_attr_setscope(pthread_attr_t *, int);
int pthread_attr_setstackaddr(pthread_attr_t *, void *);
int pthread_attr_setstacksize(pthread_attr_t *, size_t);
```

Як правило, встановлення необхідних значень атрибутів потоку у атрибутному об’єкті (pthread_attr_t) виконується до виклику функції pthread_create() з метою створити новий потік з потрібними атрибутами (вказавши адресу атрибутного об’єкту другим аргументом функції pthread_create()) (рис.3.18).

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Рис.3.18. Приклад створення “від’єднаного” потоку
(з атрибутом PTHREAD_CREATE_DETACHED).

3.3. Синхронізація програмних потоків в POSIX Threads

3.3.1. Синхронізація програмних потоків за допомогою м’ютексів

М’ютекс (lock, mutex, mutex lock) - (від англ. **mut**ual **ex**clusion — взаємне виключення) об'єкт синхронізації паралельних задач (процесів, потоків) для організації взаємного виключення при здійсненні операцій доступу до даних (запис, читання). М’ютекс може перебувати в одному з двох станів: закритий (lock) та відкритий (unlock). Відповідно над м’ютексом можна здійснити дві операції, які змінюють його стан на протилежний (рис.3.19): закрити (lock) та відкрити (unlock).

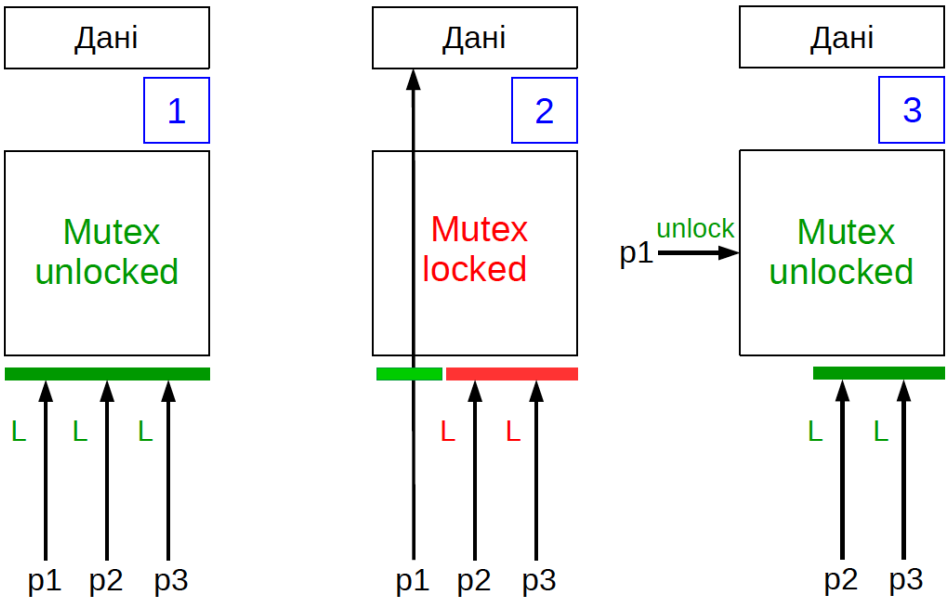

Рис.3.19. Схема використання м’ютексу.

Спроба потоку виконати операцію lock над закритим (захопленим іншим потоком) м’ютексом призводить до блокування потоку до моменту часу, коли м’ютекс буде відкритий (звільнений іншим потоком). На відміну від семафору, логіка використання м’ютексу передбачає, що його може відкрити (звільнити) лише той потік, який його перед цим закрив (захопив). Як правило, у програмі м’ютекс оголошується як глобальна змінна і використовується для “захищеного” доступу потоків до відповідного об’єкту даних. Основне використання м’ютексів - це синхронізація доступу програмних потоків до спільних даних.

Взаємне виключення можна розглядати, як метод організації послідовного (узгодженого) доступу потоків до спільних ресурсів. Приклади ситуацій, які потребують використання взаємного виключення: 1) один потік модифікує змінну, значення якої в цей же час модифікується іншим потоком; 2) потік використовує змінну, значення якої знаходиться в процесі оновлення іншим потоком, відтак отримуючи її старе значення.

Механізм взаємного виключення дозволяє програмісту створити власний протокол послідовного (узгодженого) доступу потоків до спільних даних або ресурсів. При цьому м’ютекс - це свого роду замок, яким можна віртуально захистити якийсь ресурс. Якщо потік хоче змінити або прочитати значення спільного ресурсу, він повинен спочатку отримати до нього доступ, закривши (захопивши) відповідний м’ютекс. Після цього він може виконувати будь які операції зі спільним ресурсом, не турбуючись про те, що інші потоки отримають до нього доступ, оскільки інші потоки будуть чекати своєї черги на доступ. Після того, як потік завершить роботу зі спільним ресурсом, він відкриє (звільнить) м’ютекс, тим самим дозволивши іншим потокам отримати доступ до ресурсу (рис.3.20).

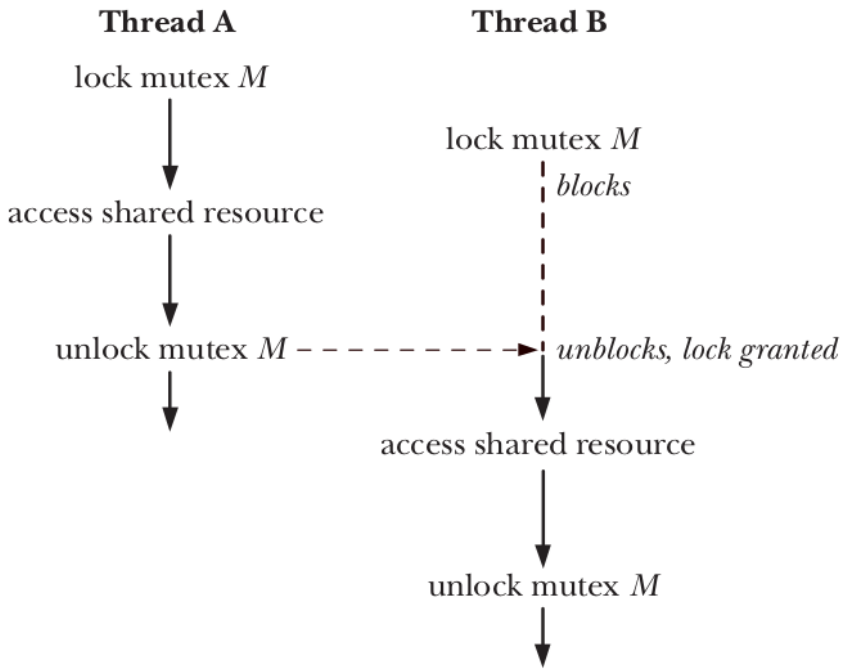

Рис.3.20. Організація послідовного (узгодженого) доступу потоків до спільного ресурсу з використанням м'ютекса.

Наведений протокол послідовного (узгодженого) доступу має бути реалізований у всіх потоках, які виконують операції зі спільним ресурсом. Якщо протокол порушується (наприклад, потік модифікує спільний ресурс без попереднього захоплення м'ютекса), то це, як правило, призводить до помилки. Аналіз програми на наявність помилок такого типу є достатньо складною задачею (складність тим більше, чим більш складну структуру має спільний ресурс, та чим більшу кількість м'ютексів використано у програмі).

В POSIX Threads (Pthreads) об'єкт синхронізації "м'ютекс" створюється у вигляді змінної типу pthread_mutex_t. Після оголошення змінної, м'ютекс потрібно ініціалізувати: або 1) присвоївши їй значення константи PTHREAD_MUTEX_INITIALIZER (рис.3.21), якщо це статична змінна, або 2) викликавши функцію pthread_mutex_init() (рис.3.22), якщо м'ютекс зберігається в області пам'яті, що виділяється динамічно. В останньому випадку перед завершенням програми потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_mutex_destroy() (рис.3.22). Після ініціалізації м'ютекс знаходиться в стані "відкритий" (незахоплений). Як об'єкт даних, м'ютекс може бути розташований або у локальній пам'яті процесу, або у ділянці спільної пам'яті (shared memory) двох чи більше процесів.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Рис.3.21. Ініціалізація м'ютексу як статичної змінної.

```
#include <pthread.h>

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);

int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Рис.3.22. Опис функцій pthread_mutex_init() та pthread_mutex_destroy().

Для здійснення операцій lock та unlock над м'ютексом використовуються функції pthread_mutex_lock() та pthread_mutex_unlock() (рис.3.23). В якості параметру вказується м'ютекс, над яким здійснюється операція. За допомогою функції pthread_mutex_trylock() (рис.3.23) можна виконати спробу захоплення м'ютекса в режимі без блокування (в асинхронному режимі). Якщо на момент виклику pthread_mutex_trylock() м'ютекс вже захоплений іншим потоком, то функція завершиться негайно і поверне код помилки EBUSY, інакше функція захопить м'ютекс і поверне 0. За допомогою функції pthread_mutex_timedlock() (рис.3.23) можна виконати спробу захоплення м'ютексу на протязі заданого проміжку часу (abs_timeout). Приклад використання м'ютексів для забезпечення послідовного (узгодженого) доступу потоків до глобального лічильника наведено на рис.3.24.

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);

int pthread_mutex_trylock (pthread_mutex_t *mutex);

int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                             const struct timespec *abs_timeout);

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Рис.3.23. Опис функцій pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_timedlock() та pthread_mutex_unlock().

```
#include <pthread.h>

pthread_mutex_t count_mutex;
int count = 0;

void increment_count() {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

int get_count() {
    int c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

Рис.3.24. Приклад використання м'ютексів для забезпечення послідовного (узгодженого) доступу потоків до глобального лічильника count.

POSIX Threads (Pthreads) надає можливість створювати м'ютекси із заданими атрибутами. Для цього використовується відповідний атрибутний об'єкт - змінна типу pthread_mutexattr_t. Перед використанням, атрибутний об'єкт потрібно ініціалізувати за допомогою функції pthread_mutexattr_init() (рис.3.25). Після завершення роботи з атрибутним об'єктом потрібно звільнити відповідний ресурс викликом pthread_mutexattr_destroy() (рис.3.25). Для отримання значень атрибутів використовуються функції, наведені на рис.3.26. Для встановлення значень атрибутів використовуються функції, наведені на рис.3.27. Типова послідовність кроків по використанню атрибутного об'єкту: створити атрибутний об'єкт,

встановити потрібні значення атрибутів, передати атрибутний об'єкт другим параметром функції pthread_mutex_init () для створення м'ютексу з відповідними атрибутами.

```
#include<pthread.h>
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

Рис.3.25. Опис функцій pthread_mutexattr_init() та pthread_mutexattr_destroy().

```
#include<pthread.h>

int pthread_mutexattr_getpshared (const pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_gettype (const pthread_mutexattr_t  *attr,  int *type);
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_getprioceiling (const pthread_mutexattr_t *attr, int *prioceiling);
int pthread_mutexattr_getrobust (const pthread_mutexattr_t *attr, int *robust);
int pthread_mutexattr_getrobust_np (const pthread_mutexattr_t *attr, int *robust_np);
```

Рис.3.26. Функції для отримання значень атрибутів м'ютекса.

```
#include <pthread.h>

int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);
int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int protocol);
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutexattr_setrobust (pthread_mutexattr_t *attr, int robust);
int pthread_mutexattr_setrobust_np (pthread_mutexattr_t *attr, int robust_np);
```

Рис.3.27. Функції для встановлення значень атрибутів м'ютекса.

За допомогою атрибуту pshared (process_shared) визначається область використання м'ютекса. Цей атрибут може мати два значення:

1) PTHREAD_PROCESS_PRIVATE – м'ютекс використовується потоками одного процесу (це значення встановлено за замовчуванням).

2) PTHREAD_PROCESS_SHARED – м'ютекс може бути використаний потоками двох чи більше процесів, що взаємодіють за допомогою механізму спільної пам'яті (shared memory), яка містить цей м'ютекс (така можливість має бути реалізована в операційній системі).

За допомогою атрибуту type визначається тип м'ютекса, який може бути одним з наступних:

1) PTHREAD_MUTEX_NORMAL – "звичайний" м'ютекс, при звертанні до якого не здійснюється перевірка наявності помилок або тупикових ситуацій (deadlocks). Якщо потік намагається захопити м'ютекс, який він вже захопив раніше, то виникає тупикова ситуація (deadlock). Звільнення м'ютекса, який не захоплений або захоплений іншим потоком, має невизначений результат (згідно стандарту Pthreads). У ОС Linux обидві ці операції успішні для цього типу м'ютекса.

2) PTHREAD_MUTEX_ERRORCHECK – м'ютекс, при звертанні до якого здійснюється перевірка наявності помилок або тупикових ситуацій (deadlocks). У всіх трьох ситуаціях (повторне захоплення, звільнення незахопленого м'ютекса, звільнення м'ютекса захопленого іншим потоком) відповідна функція Pthreads повертає помилку. Цей тип м'ютексів, як правило, повільніший за звичайний м'ютекс, але може бути корисним як інструмент відлагодження паралельної програми (наприклад, для того, щоб виявити, де програма порушує правила використання м'ютексів).

3) PTHREAD_MUTEX_RECURSIVE – рекурсивний м'ютекс, який підтримує підрахунок кількості захоплень (locks). Коли потік вперше захоплює м'ютекс, кількість захоплень встановлюється у 1. Кожна наступна операція захоплення м'ютекса тим самим потоком збільшує кількість захоплень, а кожна операція звільнення зменшує цю кількість. М'ютекс звільняється (тобто стає доступним для захоплення іншими потоками) лише тоді, коли

кількість захоплень падає до 0. Звільнення незахопленого м’ютексу не вдається, як і звільнення м’ютексу, який на даний момент захоплений іншим потоком.

4) PTHREAD_MUTEX_DEFAULT - тип м’ютекса, реалізований у системі та встановлений за замовчуванням (спеціальний тип, визначений у стандарті Pthreads задля можливості реалізації м’ютексів інших типів). У ОС Linux м’ютекс PTHREAD_MUTEX_DEFAULT еквівалентний м’ютексу PTHREAD_MUTEX_NORMAL.

3.3.2. Синхронізація програмних потоків за допомогою умовних змінних

Умовна змінна (condition variable) – це об’єкт синхронізації, за допомогою якого один потік сигналізує іншому про подію, що відбулася. Цього сигналу може очікувати один або декілька потоків (в заблокованому стані) від деякого іншого потоку. Тобто умовна змінна дозволяє одному потоку інформувати інші потоки про зміну стану деякої спільної змінної (або іншого спільного ресурсу), а також дозволяє іншим потокам очікувати (з блокуванням) на отримання такого повідомлення. Основне використання умовних змінних - це узгодження деякої послідовності операцій, що виконуються різними потоками.

Умовні змінні дозволяють синхронізувати потоки на основі фактичного значення спільних даних. Наприклад, у варіанті програми без умовної змінної потоку потрібно було б постійно перевіряти в режимі опитування (polling), чи не виконалась деяка умова, щоб продовжити своє виконання. На таку перевірку (скажімо у циклі) витрачається багато системних ресурсів, оскільки потік буде постійно зайнятий цією “роботою”. Умовна змінна - це спосіб досягнення цієї ж мети без опитування.

Механізм умовної змінної передбачає використання власне умовної змінної та відповідного м’ютексу для забезпечення атомарності операцій переходу до очікування сигналу, виходу з очікування та перевірки і сигналізації про виконання умови. Типова послідовність дій по використанню умовної змінної у двох потоках наступна (рис.3.28):

1) Потік **A** (очікує на сигнал): виконує обчислення до моменту, коли потрібно дочекатись виконання певної умови; захоплює м’ютекс M; переходить до очікування виконання умови (pthread_cond_wait()), тобто очікує на сигнал від потоку **B** про виконання умови (після виклику функція pthread_cond_wait() звільняє м’ютекс M); очікує, перебуваючи в заблокованому (призупиненому) стані; після отримання сигналу про виконання умови, потік **A** розблоковується (перед завершенням функція pthread_cond_wait() знову захоплює м’ютекс M); звільняє м’ютекс M; продовжує виконання.

2) Потік **B** (сигналізує): виконує обчислення до моменту, коли внаслідок його “дій” змінюється умова; захоплює м’ютекс M; виконує дії, що призводять до зміни умови; перевіряє умову, і якщо вона виконується, то сигналізує про це потоку **A** (pthread_cond_signal()); звільняє м’ютекс M; продовжує виконання.

Thread A		**Thread B**
• Do work up to the point where a certain condition must occur (such as "count" must reach a specified value) • Lock associated mutex and check value of a global variable • Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B. • When signalled, wake up. Mutex is automatically and atomically locked. • Explicitly unlock mutex • Continue		• Do work • Lock associated mutex • Change the value of the global variable that Thread-A is waiting upon. • Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A. • Unlock mutex. • Continue

Рис.3.28. Типова послідовність дій по використанню умовної змінної у двох потоках.

Умовну змінну можна використати, наприклад, для узгодження виконання двох потоків, один з яких (потік А) записує дані у спільну чергу, а інший (потік В) читає дані з цієї черги (рис.3.29). В цьому прикладі потік А сигналізує потоку В про те, що в черзі з'явилися дані, які можна прочитати. У потоці В перед переходом до очікування на сигнал від потоку А виконується перевірка (не)виконання умови (while(queue is empty)), щоб уникнути нескінченного блокування на cond_wait(C,M), оскільки, якщо чекати на умову, яка вже виконалась, то є всі шанси, що про її виконання ніхто ніколи не просигналізує.

Thread A

1. Read data [B]
2. Lock *M* [b]
3. Put item on queue
4. cond_signal(*C*)
5. Unlock *M*
6. Goto step 1

Thread B

1. Lock *M* [b]
2. while (queue is empty) {
   cond_wait(*C*, *M*) [B]
   }
3. Remove item; update database
4. Unlock *M*
5. Goto step 1

Рис.3.29. Приклад узгодження виконання двох потоків, які записують та читають дані зі спільної черги, за допомогою умовної змінної С.

При використанні умовних змінних потрібно зважати на можливість помилкового сигналізування про виконання умови (spurious wakeup). В цьому випадку потік, який дочекався сигналу, розблоковується, але перед його подальшим виконанням варто додатково перевірити виконання умови. У наведеному прикладі (рис.3.29) така перевірка реалізована у вигляді циклу while (while(queue is empty)). Якщо умова все ж таки не виконана, то потік знову переходить до очікування (cond_wait(C,M)). Причиною помилкового сигналізування (spurious wakeup) може бути те, що за час передавання сигналу якийсь інший потік встиг виконати дії, які змінили умову, про виконання якої було просигналізовано.

В POSIX Threads (Pthreads) об'єкт синхронізації "умовна змінна" створюється у вигляді змінної типу pthread_cond_t. Після оголошення змінної, умовну змінну потрібно ініціалізувати: або 1) присвоївши їй значення константи PTHREAD_COND_INITIALIZER, якщо це статична змінна, або 2) викликавши функцію pthread_cond_init() (рис.3.30), якщо умовна змінна зберігається в області пам'яті, що виділяється динамічно. В останньому випадку перед завершенням програми потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_cond_destroy() (рис.3.30). Як об'єкт даних, умовна змінна може бути розташована або у локальній пам'яті процесу, або у ділянці спільної пам'яті (shared memory) двох чи більше процесів. Разом з умовною змінною створюється відповідний м'ютекс, який необхідний для роботи з нею (рис.3.31).

```
#include <pthread.h>

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);

int pthread_cond_destroy (pthread_cond_t *cond);
```

Рис.3.30. Опис функцій pthread_mutex_init() та pthread_mutex_destroy().

```
int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

```
int main(int argc, char *argv[])
{
.  .  .

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init(&count_threshold_cv, NULL);

.  .  .

  /* Clean up and exit */
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);
}
```

Рис.3.31. Приклад створення, ініціалізації та знищення умовної змінної.

Для очікування сигналу про виконання умови використовується функція pthread_cond_wait() (рис.3.32). Першим параметром вказується умовна змінна cond, а другим параметром (mutex) вказується відповідний м'ютекс. Для очікування сигналу про виконання умови на заданому проміжку часу (timeout) використовується функція pthread_cond_timedwait() (рис.3.32). Функція pthread_cond_wait() блокує викликаючий потік доти, доки якийсь інший потік не просигналізує про виконання умови за допомогою функції pthread_cond_signal(). Функція pthread_cond_wait() викликається лише після захоплення м'ютекса mutex. Сама функція автоматично і атомарно звільняє цей м'ютекс на час очікування. Після надходження сигналу про виконання умови, потік розблоковується і при виході з функції pthread_cond_wait() mutex автоматично захоплюється знову. Тому після розблокування потоку потрібно звільнити цей м'ютекс викликом pthread_mutex_unlock() (рис.3.33).

```
#include <pthread.h>
#include <time.h>

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait (pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *timeout);
```

Рис.3.32. Опис функцій pthread_cond_wait() та pthread_cond_timedwait().

```
  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n",
my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.
Count= %d\n", my_id,count);
    printf("watch_count(): thread %ld Updating the value of count...\n",
my_id,count);
    count += 125;
```

```
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
```

Рис.3.33. Приклад використання функції pthread_cond_wait()
для очікуваня сигналу про виконання умови.

Для надсилання сигналу про виконання умови використовується функція pthread_cond_signal() (рис.3.34), параметром якої вказується відповідна умовна змінна cond. Виклик функції pthread_cond_signal() розблокує один потік, що очікує на сигнал (тобто заблокований викликом pthread_cond_wait()). Якщо надходження цього сигналу також очікують інші потоки, то вони залишаться заблокованими. Для надсилання сигналу одночасно усім потокам, які чекають на виконання відповідної умови, використовується функція pthread_cond_broadcast() (рис.3.34). Перед викликом pthread_cond_signal() також потрібно спочатку захопити, а після завершення виклику звільнити м'ютекс, прив'язаний до умовної змінної (рис.3.35).

```
#include <pthread.h>

int pthread_cond_signal (pthread_cond_t *cond);

int pthread_cond_broadcast (pthread_cond_t *cond);
```

Рис.3.34. Опис функцій pthread_cond_signal() та pthread_cond_broadcast().

```
    pthread_mutex_lock(&count_mutex);
    count++;
    /*
    Check the value of count and signal waiting thread when condition is
    reached. Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
         my_id, count);
    pthread_mutex_unlock(&count_mutex);
```

Рис.3.35. Приклад використання функції pthread_cond_signal() для
надсилання сигналу про виконання умови.

POSIX Threads (Pthreads) надає можливість створювати умовну змінну із заданими атрибутами. Для цього використовується відповідний атрибутний об'єкт - змінна типу pthread_condattr_t. Перед використанням, атрибутний об'єкт потрібно ініціалізувати за допомогою функції pthread_condattr_init() (рис.3.36). Після завершення роботи з атрибутним об'єктом потрібно звільнити відповідний ресурс викликом pthread_condattr_destroy() (рис.3.36). Для отримання та встановлення значень атрибутів використовуються функції, наведені на рис.3.36. Типова послідовність кроків по використанню атрибутного об'єкту: створити атрибутний об'єкт, встановити потрібні значення атрибутів, передати атрибутний об'єкт другим параметром функції pthread_cond_init() для створення умовної змінної з відповідними атрибутами. Атрибут pshared у випадку умовної змінної має те саме значення, що відповідний атрибут м'ютекса.

```
#include<pthread.h>
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);

int pthread_condattr_getpshared (const pthread_condattr_t *attr, int *pshared);
int pthread_condattr_getclock (const pthread_condattr_t *attr, clockid_t *clock_id);

int pthread_condattr_setpshared (pthread_condattr_t *attr, int pshared);
int pthread_condattr_setclock (pthread_condattr_t *attr, clockid_t clock_id);
```

Рис.3.36. Опис функцій для роботи з атрибутами умовної змінної.

3.3.3. Блокування читання-запису (readers–writer lock)

Блокування читання-запису (readers–writer lock) - це механізм синхронізації, який дозволяє одночасне читання та ексклюзивний запис захищених спільних даних. Блокування читання-запису реалізовано у вигляді відповідного об'єкту синхронізації rwlock, який можна захопити або в режимі читання (read lock), або в режимі запису (write lock). Щоб змінити спільні дані, потік повинен спочатку отримати ексклюзивний доступ для запису (захопити rwlock в режимі запису). Ексклюзивне захоплення для запису не дозволяється, поки не будуть звільнені всі захоплення даного rwlock в режимі читання.

Блокування читання-запису (readers–writer lock) використовується у випадках, коли на відміну від використання м'ютексів, є можливість розмежувати операції читання та запису (за умов що дозволено одночасне читання захищених даних декількома потоками). За рахунок цього досягається прискорення паралельного виконання (одночасні операції читання не блокуються), тобто досягається вищий ступінь паралелізму. В деяких паралельних програмах дані зчитуються частіше, ніж змінюються, тому такі програми виграють у швидкості виконання, якщо в них використовуються блокування читання-запису замість м'ютексів.

На відміну від м'ютекса, який може мати два стани (відкритий, закритий), об'єкт блокування читання-запису rwlock може мати три стани: блокування читання (закритий для операцій читання), блокування запису (закритий для операцій запису) і відсутність блокування (відкритий). При цьому між захопленням rwlock на читання чи запис є різниця. Зокрема тут діють наступні правила:

1) Будь-яка кількість потоків може заблокувати ресурс для зчитування (read lock), якщо жоден інший потік не заблокував його для запису.

2) Блокування ресурсу для запису (write lock) може бути встановлено, лише коли жоден інший потік не заблокував ресурс для читання або для запису.

Іншими словами, довільна кількість потоків може зчитувати спільні дані, якщо жоден потік не змінює їх у поточний момент. Спільні дані можуть бути змінені лише за умови, що ніхто інший їх не зчитує й не змінює.

Якщо блокування читання-запису встановлено в режимі для запису, всі потоки, які будуть намагатися також встановити блокування для запису, будуть призупинені доти, поки блокування не буде знято. Якщо блокування читання-запису встановлено в режимі для читання, всі потоки, які будуть намагатися також встановити блокування для читання, отримають доступ до ресурсу, але якщо який-небудь потік спробує встановити блокування для запису, він буде призупинений доти, поки не буде знято останнє блокування для читання.

В POSIX Threads (Pthreads) об'єкт синхронізації rwlock створюється у вигляді змінної типу pthread_rwlock_t. Після оголошення змінної, rwlock потрібно ініціалізувати одним з двох способів:

1) присвоївши змінній значення константи PTHREAD_RWLOCK_INITIALIZER, якщо це статична змінна, або

2) викликавши функцію pthread_rwlock_init() (рис.3.37), якщо rwlock зберігається в області пам'яті, що виділяється динамічно.

В останньому випадку перед завершенням програми потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_rwlock_destroy() (рис.3.37). Після ініціалізації rwlock знаходиться в стані “відкритий” (незахоплений). Як об’єкт даних, rwlock може бути розташований або у локальній пам’яті процесу, або у ділянці спільної пам’яті (shared memory) двох чи більше процесів.

```
#include <pthread.h>

int pthread_rwlock_init (pthread_rwlock_t *rwlock,
                         const pthread_rwlockattr_t *attr);

int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
```

Рис.3.37. Опис функцій pthread_rwlock_init() та pthread_rwlock_destroy().

Для здійснення операцій read lock, write lock та unlock над rwlock використовуються функції pthread_rwlock_rdlock(), pthread_rwlock_wrlock() та pthread_rwlock_unlock() (рис.3.38). В якості параметру вказується rwlock, над яким здійснюється операція. За допомогою функцій pthread_rwlock_tryrdlock() та pthread_rwlock_trywrlock() (рис.3.38) можна виконати спробу захоплення rwlock в режимі без блокування (в асинхронному режимі). Якщо на момент виклику цих функцій rwlock вже захоплений іншим потоком (у відповідному режимі), то функція завершиться негайно і поверне код помилки EBUSY, інакше функція захопить rwlock і поверне 0. За допомогою функцій pthread_rwlock_timedrdlock() та pthread_rwlock_timedwrlock() (рис.3.38) можна виконати спробу захоплення rwlock на протязі заданого проміжку часу (abs_timeout).

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_timedrdlock (pthread_rwlock_t *rwlock,
                                const struct timespec *abs_timeout);

int pthread_rwlock_timedwrlock (pthread_rwlock_t *rwlock,
                                const struct timespec *abs_timeout);

int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Рис.3.38. Опис функцій для здійснення операцій над rwlock.

POSIX Threads (Pthreads) надає можливість створювати rwlock із заданими атрибутами. Для цього використовується відповідний атрибутний об’єкт – змінна типу pthread_rwlockattr_t. Перед використанням, атрибутний об’єкт потрібно ініціалізувати за допомогою функції pthread_rwlockattr_init() (рис.3.39). Після завершення роботи з атрибутним об’єктом потрібно звільнити відповідний ресурс викликом pthread_rwlockattr_destroy() (рис.3.39). Для отримання та встановлення значень атрибутів використовуються функції, наведені на рис.3.39. Типова послідовність кроків по використанню атрибутного об’єкту: створити атрибутний об’єкт, встановити потрібні значення атрибутів, передати атрибутний об’єкт другим параметром функції pthread_rwlock_init() для створення rwlock з відповідними атрибутами. Атрибут pshared у випадку rwlock має те саме значення, що відповідний атрибут м’ютекса.

```
#include<pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_getkind_np(const pthread_rwlockattr_t *attr, int *pref);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t *attr, int pref);
```

Рис.3.39. Опис функцій для роботи з атрибутами rwlock.

3.3.4. Спін-блокування (spinlock)

Спін-блокування (spinlock) використовуєтья в тих самих ситуаціях, де потрібен м’ютекс, але блокування (призупинка) потоку на виклику операції lock неприпустиме. Наприклад, обробник переривання у ядрі ОС ніколи не повинен блокуватись. Якщо це станеться, система зависне (вийде з ладу). Якщо в обробнику переривання потрібно вставити елемент даних у загальнодоступний зв'язаний список, то для цього захоплюється відповідний спінлок, вставляється елемент даних, звільняється спінлок.

Спінлок (spinlock) - це механізм синхронізації низького рівня, який, головним чином, використовується у мультипроцесорних системах зі спільною пам'яттю (shared memory multiprocessors). Коли викликаючий потік намагається захопити спінлок, який вже захоплений іншим потоком, він не призупиняється, а виконує спеціальний цикл, в якому весь час перевіряє, чи спінлок не звільнився. Коли спінлок захоплено, його слід “утримувати” лише короткий час, оскільки виконання потоків, що в циклі очікують на звільнення спінлоку, витрачає процесорний час. Перед тим, як потік з якоїсь причини призупиняється, він повинен звільнити усі захоплені спінлоки, щоб надати іншим потокам можливість їх захопити.

При виконанні будь-якого блокування виникає дилема між 1) витратою ресурсів процесора для налаштування блокування потоку на м’ютексі та 2) витратою ресурсів процесора на виконання потоку в циклі, коли він заблокований на спінлоку. Спінлоки вимагають небагато ресурсів для налаштування, після чого виконується простий цикл, в якому повторюється атомарна операція захоплення, поки її виконання не стане можливим. Тобто під час очікування потік продовжує споживати ресурси процесора. В цілому спінлоки витрачають менше системних ресурсів в разі короткочасного блокування потоків, а м’ютекси витрачають менше системних ресурсів в разі порівняно тривалого блокування потоків.

В POSIX Threads (Pthreads) об’єкт синхронізації “спінлок” створюється у вигляді змінної типу pthread_spinlock_t. Після оголошення змінної, спінлок потрібно ініціалізувати викликом функції pthread_spin_init() (рис.3.40). Другим параметром pshared задається значення відповідного атрибуту спінлоку (process_shared). Атрибут pshared у випадку спінлоку має те саме значення, що і відповідний атрибут м’ютекса. Після завершення роботи зі спінлоком потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_spin_destroy() (рис.3.40).

```
#include<pthread.h>

int pthread_spin_init (pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy (pthread_spinlock_t *lock);
```

Рис.3.40. Опис функцій pthread_spin_init() та pthread_spin_destroy().

Для здійснення операцій lock та unlock над спінлоком використовуються функції pthread_spin_lock() та pthread_spin_unlock() (рис.3.41). В якості параметру вказується спінлок, над яким здійснюється операція. За допомогою функції pthread_spin_trylock() (рис.3.41) можна

виконати спробу захоплення спінлока в режимі без блокування (в асинхронному режимі). Якщо на момент виклику pthread_spin_trylock() спінлок вже захоплений іншим потоком, то функція завершиться негайно і поверне код помилки EBUSY, інакше функція захопить спінлок і поверне 0.

```
#include <pthread.h>

int pthread_spin_lock (pthread_spinlock_t *lock);

int pthread_spin_trylock (pthread_spinlock_t *lock);

int pthread_spin_unlock (pthread_spinlock_t *lock);
```

Рис.3.41. Опис функцій pthread_spin_lock(), pthread_spin_trylock() та pthread_spin_unlock().

3.3.5. Бар’єрна синхронізація програмних потоків

Бар’єрна синхронізація використовується у випадках, коли потрібно дочекатися завершення виконання обчислень у декількох програмних потоках, перш ніж продовжити виконання програми одночасно у всіх цих потоках (рис.3.42). У стандарті POSIX Threads (Pthreads) визначено відповідний об’єкт синхронізації barrier та функції з префіксом pthread_barrier для роботи з ним. Ці функції дозволяють створити бар'єр, вказавши кількість потоків, які мають зібратися на бар'єрі, і реалізувати у потоках очікування моменту часу, коли зрештою всі потоки “підійдуть до бар’єру”. Коли останній потік “підходить до бар'єру”, всі потоки відновлюють виконання.

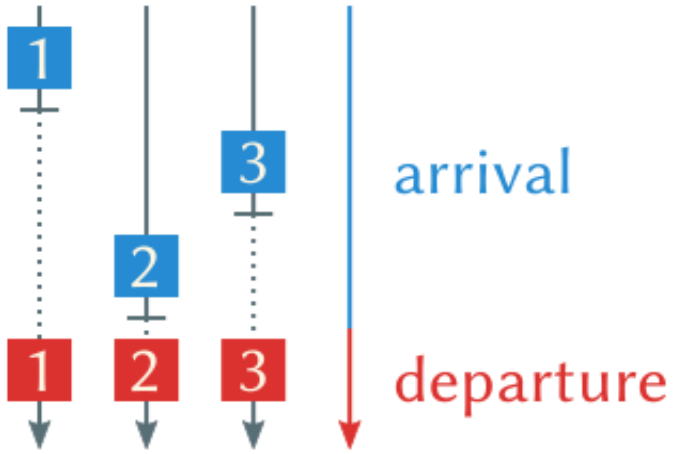

Рис.3.42. Схема бар’єрної синхронізації трьох потоків.

В POSIX Threads (Pthreads) об’єкт синхронізації barrier створюється у вигляді змінної типу pthread_barrier_t. Після оголошення змінної, бар’єр потрібно ініціалізувати викликом функції pthread_barrier_init() (рис.3.43). Третім параметром count задається кількість потоків, які будуть збиратися у бар’єра. Для бар’єрної синхронізації потоку потрібно викликати функцію pthread_barrier_wait() (рис.3.44). Коли count потоків зрештою викличуть функцію pthread_barrier_wait() для заданого бар’єру, то усі вони одночасно розблокуються і продовжать своє виконання. Після завершення роботи з бар’єром потрібно звільнити відповідний системний ресурс, викликавши функцію pthread_barrier_destroy() (рис.3.43).

```
#include <pthread.h>
```

```
int pthread_barrier_init (pthread_barrier_t *restrict barrier,
                          const pthread_barrierattr_t *restrict attr,
                          unsigned count);

int pthread_barrier_destroy (pthread_barrier_t *barrier);
```

Рис.3.43. Опис функцій pthread_mutex_init() та pthread_mutex_destroy().

```
#include <pthread.h>

int pthread_barrier_wait (pthread_barrier_t  *barrier);
```

Рис.3.44. Опис функції pthread_barrier_wait().

POSIX Threads (Pthreads) надає можливість створювати бар'єри із заданим атрибутом pshared (process_shared). Для цього використовується відповідний атрибутний об'єкт - змінна типу pthread_barrierattr_t. Перед використанням, атрибутний об'єкт потрібно ініціалізувати за допомогою функції pthread_barrierattr_init() (рис.3.45). Після завершення роботи з атрибутним об'єктом потрібно звільнити відповідний ресурс викликом pthread_barrierattr_destroy() (рис.3.45). Для отримання та встановлення значення атрибуту pshared використовуються функції pthread_barrierattr_getpshared() та pthread_barrierattr_setpshared() (рис.3.45). Типова послідовність кроків по використанню атрибутного об'єкту: створити атрибутний об'єкт, встановити потрібне значення атрибуту pshared, передати атрибутний об'єкт другим параметром функції pthread_barrier_init() для створення бар'єру з відповідним атрибутом. Атрибут pshared у випадку бар'єра має те саме значення, що відповідний атрибут м'ютекса.

```
#include<pthread.h>

int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr,
                                   int *pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Рис.3.45. Опис функцій для роботи з атрибутним об'єктом бар'єру.

Контрольні питання

1. Які основні переваги розпаралелювання обчислень на рівні програмних потоків у порівнянні з розпаралелюванням на рівні обчислювальних процесів?
2. Для чого призначений інтерфейс прикладного програмування POSIX Threads?
3. Яка модель відображення потоків рівня користувача у потоки рівня ядра реалізована в Native POSIX Thread Library (NPTL)?
4. Яку функцію можна вважати потоко-безпечною (thread-safe)?
5. Яка функція призначена для створення нового програмного потоку в POSIX Threads і що вказується у її параметрах?
6. Яка функція POSIX Threads надсилає вказаному програмному потоку запит на завершення?
7. Як в POSIX Threads організовано очікування на завершення програмного потоку?
8. За допомогою якої функції POSIX Threads вказаний потік робиться від'єднаним, тобто таким на завершення якого не може очікувати жоден інший потік?
9. Які функції POSIX Threads використовуються для управління атрибутами потоків?
10. Які основні атрибути має програмний потік в POSIX Threads?
11. Які функції POSIX Threads використовуються для управління м'ютексами?
12. Який тип м'ютекса в POSIX Threads передбачає перевірку наявності помилок або ситуацій взаємного блокування (deadlocks) при звертанні до м'ютекса?
13. Який тип м'ютекса в POSIX Threads підтримує підрахунок кількості операцій захоплення (locks) та звільнення (unlocks) м'ютекса?
14. Для чого використовуються умовна змінна (conditional variable)?
15. Як організовано синхронізацію програмних потоків за допомогою умовної змінної?
16. Які функції POSIX Threads використовуються для управління умовними змінними?
17. Яка функція POSIX Threads призначена для очікування сигналу про виконання умови при використанні умовної змінної?
18. В яких станах може знаходитись об'єкт синхронізації "блокування читання-запису" (readers–writer lock)?
19. Яка функція POSIX Threads використовуються для здійснення операції «блокування запису» над об'єктом синхронізації readers–writer lock (rwlock)?
20. Для чого використовується засіб синхронізації спінлок (spinlock) і чим він відрізняється від м'ютекса?
21. Які функції POSIX Threads використовуються для бар'єрної синхронізації програмних потоків?

4. Технології паралельного програмування OpenMP та oneTBB

4.1. Інтерфейс прикладного програмування OpenMP

4.1.1. Організація паралельних обчислень за допомогою OpenMP

OpenMP (Open Multi-Processing) – це прикладний програмний інтерфейс паралельного програмування на рівні програмних потоків (базові мови програмування: C, C++, Fortran), який призначений для спрощення створення паралельних програм для апаратних платформ типу “мультипроцесорна система зі спільною пам'яттю” (shared memory multiprocessing) (рис.4.1) [16-20,25]. Існують реалізації цього програмного інтерфейсу для всіх основних операційних середовищ, в тому числі ОС Linux, інших Unix-подібних ОС та ОС Windows.

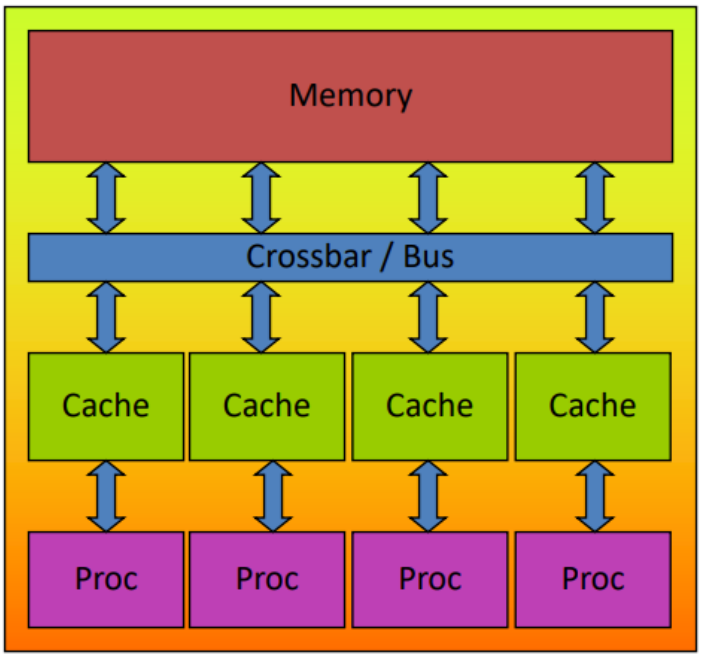

Рис.4.1. Узагальнена архітектура мультипроцесорних систем зі спільною пам'яттю (shared memory multiprocessing).

OpenMP складається з набору директив компілятора (compiler directives), бібліотеки функцій та змінних оточення (environment variables), які впливають на поведінку паралельної програми під час її виконання. Специфікація OpenMP визначається консорціумом основних постачальників апаратного та програмного забезпечення у вигляді переносимої масштабованої моделі, яка надає програмістам простий та гнучкий інтерфейс для розроблення паралельних програм для широкого спектру апаратних платформ (від персональних комп'ютерів до суперкомп'ютерів).

Обчислювальні завдання, що виконуються програмними потоками паралельно, так само як і дані, необхідні для виконання цих завдань, описуються за допомогою спеціальних директив препроцесора (директив компілятора) відповідної мови програмування, так званих, прагм. За допомогою цих директив програміст вказує, яка ділянка (блок) програмного коду буде виконуватись паралельно декількома програмними потоками (рис.4.2). Директиви OpenMP також можуть бути використані для управління ходом паралельних обчислень, управління кількістю програмних потоків, управління класами змінних, розподілу обчислювальних завдань між потоками, розподілу обчислювального навантаження між потоками та синхронізації паралельного виконання потоків.

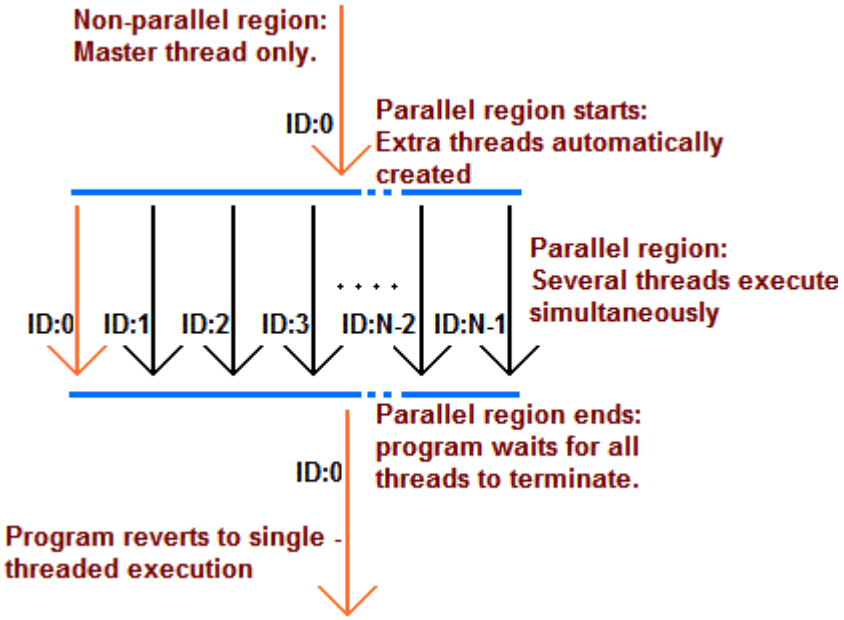

Рис.4.2. Схема розпаралелювання потоків за допомогою директив OpenMP.

Паралельна версія програми створюється після її компіляції за допомогою компілятора, що підтримує технологію OpenMP. Тобто функцію створення паралельної програми (у машинних кодах) частково бере на себе компілятор з підтримкою OpenMP, що у порівнянні з використанням низькорівневих програмних бібліотек для роботи з потоками (наприклад, POSIX Threads) звільняє програміста від необхідності створювати та відлагоджувати програмний код по створенню, організації паралельного виконання та синхронізації потоків.

В основі розпаралелювання з використанням OpenMP лежить модель клонування-приєднання потоків (fork–join model). Програміст вказує директивою OpenMP початок блоку коду, де відбудеться клонування (fork) потоків. Створені потоки паралельно виконують вказаний блок коду. Після його виконання потоки збираються на неявному бар'єрі (implicit barrier) і "приєднуються" (join) до основного потоку (рис.4.2). Кількість створюваних в точці розпаралелювання потоків можна регулювати як в самій програмі за допомогою виклику відповідних бібліотечних функцій OpenMP, так і ззовні, за допомогою поточного значення змінних оточення.

За рахунок ідеї "інкрементного розпаралелювання" OpenMP зручно використовувати для швидкого розпаралелювання послідовних версій програм з великими циклами, в яких присутній "потенційний" паралелізм (повна або часткова незалежність обчислень в окремих ітераціях циклу). Згідно цієї ідеї розробник не створює нову паралельну версію програми з нуля, а просто додає в текст послідовної версії програми потрібні директиви OpenMP. За допомогою директив OpenMP програміст може реалізувати будь яку схему розпаралелювання обчислень в рамках моделі клонування-приєднання потоків (fork–join model) (рис.4.3).

Передбачається, що OpenMP-програма на однопроцесорній платформі може бути виконана як послідовна програма, тобто відсутня необхідність підтримувати окремо послідовну і паралельну версії програми. У випадку компіляції програми на однопроцесорній платформі директиви OpenMP просто ігноруються компілятором, а для виклику бібліотечних функцій OpenMP застосовуються "заглушки" (stubs), текст яких наведений в специфікаціях стандарту OpenMP.

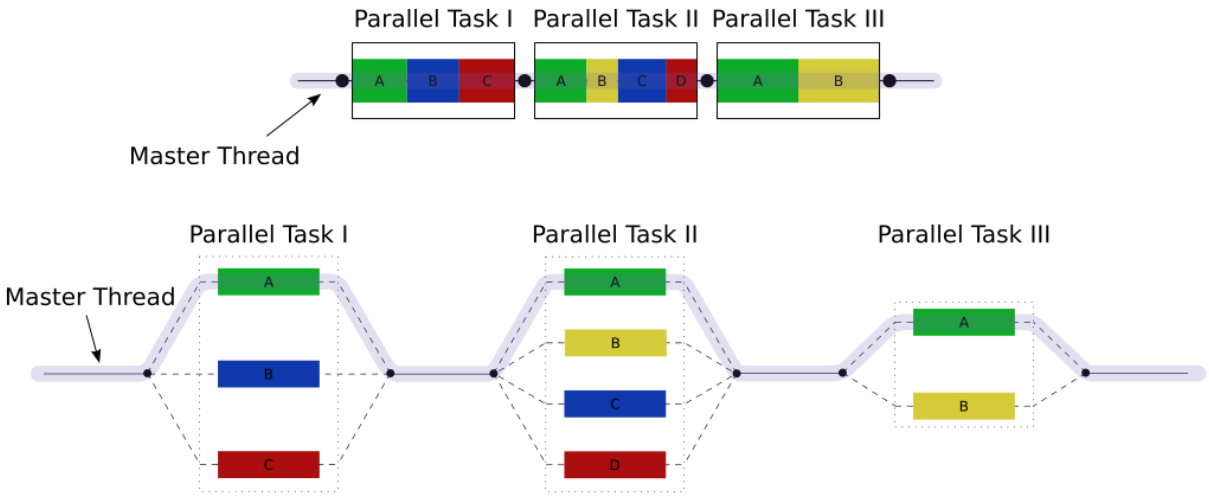

Рис.4.3. Приклад схеми розпаралелювання обчислень на рівні програмних потоків.

4.1.2. Використання OpenMP

Інтерфейс прикладного програмування (API) OpenMP (Open Multi-Processing) забезпечує розпаралелювання на рівні програмних потоків для більшості сучасних апаратних платформ (мультипроцесорних систем зі спільною пам'яттю) на мовах програмування C, C++ та Fortran. OpenMP підтримується в багатьох операційних системах, включаючи Solaris, AIX, HP-UX, Linux, macOS та Windows. OpenMP використовує переносиму, масштабовану модель паралельних обчислень, яка надає програмістам простий і гнучкий інтерфейс для розробки паралельних програм для широкого спектру комп'ютерних засобів, починаючи від стандартного персонального комп'ютера і закінчуючи суперкомп'ютером.

Розробкою стандарту OpenMP займається некомерційний технологічний консорціум OpenMP Architecture Review Board (OpenMP ARB), сформований провідними компаніями в області комп'ютерних систем та програмного забезпечення, включаючи Arm, AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments та Oracle Corporation. На домашній сторінці консорціуму (www.openmp.org) можна ознайомитись зі специфікацією стандарту та скористатись іншими ресурсами для розробки OpenMP-програм. Остання версія стандарту - OpenMP 5.2 (November 9, 2021).

Інтерфейс прикладного програмування OpenMP реалізований у багатьох "комерційних" та вільних (free, opensource) компіляторах. Для прикладу підтримка OpenMP реалізована у лінійці продуктів Microsoft Visual Studio та середовищі паралельного програмування Intel Parallel Studio. OpenMP також підтримується у GNU GCC починаючи з версії 4.2. Треба пам'ятати, що не всі компілятори (та ОС) підтримують повний набір функцій останньої версії стандарту OpenMP.

Для використання технології OpenMP в програмі на мові ANSI C, потрібно включити в текст програми хедерний файл omp.h (#include <omp.h>). У разі використання компілятора GNU GCC, в момент компіляції потрібно вказати ключ -fopenmp (рис.4.4). На рис.4.5 наведено приклад простої OpenMP-програми.

```
$gcc -fopenmp -o hello hello.c
```

Рис.4.4. Приклад компіляції програми з врахуванням директив OpenMP.

```
#include <stdio.h>
#include <omp.h>

void main() {
```

```
  int x;

  printf("start\n");

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    x = id;
    printf("hello world (%d)\n", id);
  }

  printf("finish (%d)\n", x);
}
```

Рис.4.5. Приклад простої OpenMP-програми.

4.1.3. Складові компоненти OpenMP

Інтерфейс прикладного програмування OpenMP складається з наступних компонент (рис.4.6):

1) конструкції для розпаралелювання (клонування та неявного приєднання паралельних програмних потоків), наприклад, директива parallel;

2) конструкції для розподілу роботи (обчислювального навантаження та різних обчислювальних завдань) між програмними потоками, наприклад, директиви do/for або section;

3) конструкції для управління класами змінних (роботою з даними): вирази shared і private;

4) конструкції для синхронізації паралельного виконання програмних потоків, наприклад директиви critical, atomic та barrier;

5) бібліотечні функції часу виконання, наприклад, omp_get_thread_num;

6) внутрішні контрольні змінні (internal control variables, ICVs) та відповідні змінні оточення, наприклад, OMP_NUM_THREADS.

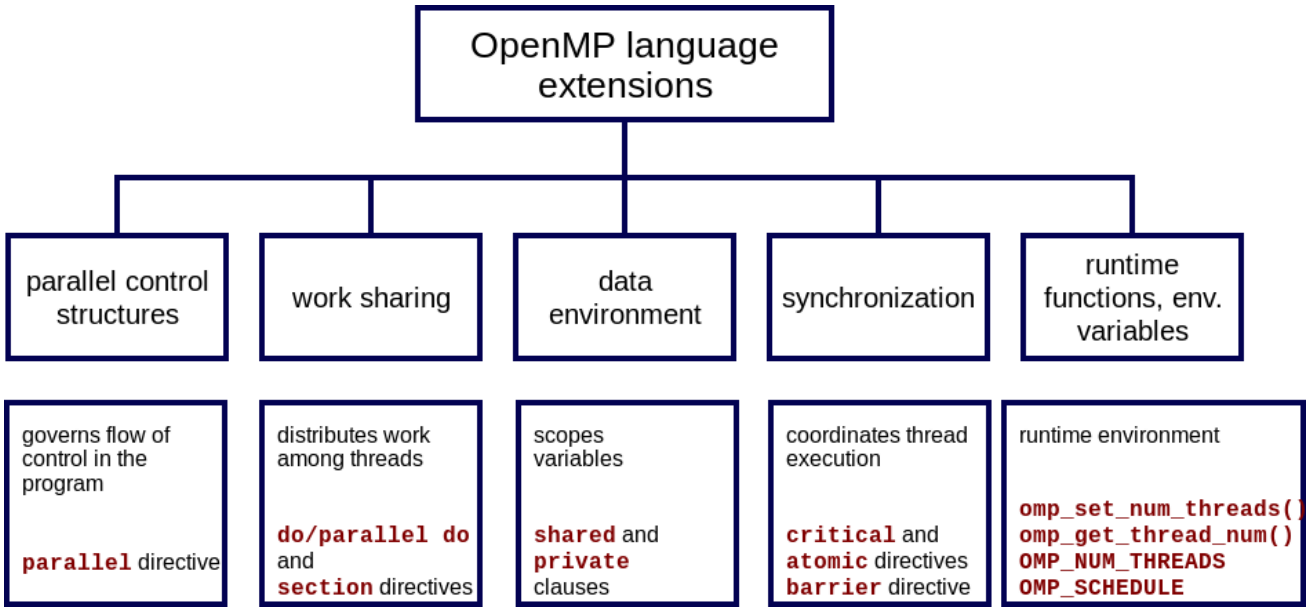

Рис.4.6. Складові компоненти OpenMP.

4.1.4. Опис паралельних частин програми

Для вказання паралельної частини OpenMP-програми використовується директива parallel. В програмах на мовах програмування C та C++ директива має вигляд: #pragma omp parallel (рис.4.5, рис.4.7). Початок та кінець блоку коду позначається фігурними дужками (рис.4.7). В програмах на мові програмування Fortran директива має вигляд: !$OMP PARALLEL (рис.4.8). Початок та кінець блоку коду позначається директивами !$OMP PARALLEL та !$OMP END PARALLEL (рис.4.8).

```
#pragma omp parallel
{
      // блок коду, що виконується паралельно
}
```

Рис.4.7. Використання директиви parallel в програмах на мовах програмування C та C++.

```
!$OMP PARALLEL

        < блок коду, що виконується паралельно >

!$OMP END PARALLEL
```

Рис.4.8. Використання директиви parallel в програмах на мові програмування Fortran.

Для виконання коду, розташованого у вказаному блоку, додатково породжується OMP_NUM_THREADS-1 потоків, де OMP_NUM_THREADS - це змінна оточення, значення якої користувач, взагалі кажучи, може змінювати. Процес, який виконав дану директиву (головний потік, main thread), завжди отримує номер 0. Усі OMP_NUM_THREADS потоків (головний та OMP_NUM_THREADS-1 породжених) паралельно виконують код, який містить блок. Після завершення блоку коду автоматично відбувається неявна бар'єрна синхронізація всіх потоків, і як тільки всі потоки доходять до цієї точки, головний потік продовжує виконання наступної частини програми, а решта потоків знищуються.

Паралельні області коду можуть бути вкладеними одна в одну (рис.4.9). За замовчуванням вкладена паралельна область коду виконується одним потоком. Необхідну стратегію обробки вкладених областей коду визначає змінна оточення OMP_NESTED, значення якої можна змінити в програмі за допомогою функції omp_set_nested(). В стандарті OpenMP v.5.1 пропонується замість цього механізму використовувати механізм "рівнів активності" (active levels) та відповідну функцію omp_set_max_active_levels().

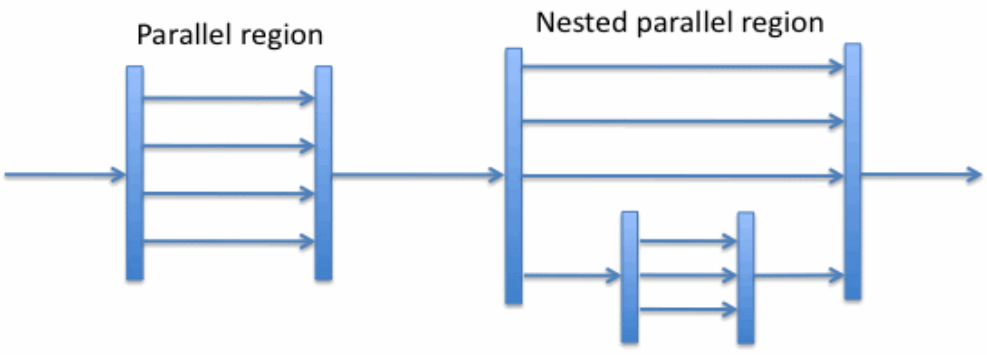

Рис.4.9. Приклад схеми розпаралелювання з вкладеними паралельними областями коду.

Необхідність породження потоків і паралельного виконання коду паралельної області програми програміст може визначати динамічно за допомогою додаткової опції if в директиві

parallel (рис.4.10). Якщо <умова> не виконана, то директива parallel не спрацьовує і виконання програми продовжується в послідовному режимі.

```
#pragma omp parallel if (<умова>)
{
      //блок коду, що виконується паралельно, якщо виконується <умова>
}
```

Рис.4.10. Використання директиви parallel if.

За допомогою функції omp_in_parallel() можна визначити в якій своїй частині – послідовній чи паралельній виконується програма. Функція повертає 0, якщо вона викликана у послідовній частині програми, або 1, якщо вона викликана у паралельній області програми (тобто якщо значення внутрішньої контрольної змінної (ICV) active-levels-var більше нуля).

4.1.5. Управління кількістю програмних потоків

Змінна оточення OMP_NUM_THREADS встановлює максимальну кількість програмних потоків, які будуть виконувати вказану директивами OpenMP паралельну область програми (якщо в програмі не вказано жодне інше значення). Значення змінної оточення OMP_NUM_THREADS встановлює початкове значення контрольної змінної (ICV) nthreads-var. Як правило, за замовчуванням значення OMP_NUM_THREADS дорівнює кількості апаратних одиниць розпаралелювання (процесорів, ядер процесора), які доступні в системі, тобто які "бачить" операційна система. Передбачається, що перед запуском OpenMP-програми користувач може змінити значення змінної оточення OMP_NUM_THREADS на потрібне.

Змінна оточення OMP_DYNAMIC включає (true, 1) та виключає (false, 0) режим динамічного регулювання кількості потоків, що породжуються для виконання паралельних областей, встановлюючи відповідне початкове значення контрольної змінної (ICV) dyn-var. Якщо значення змінної OMP_DYNAMIC встановлено в 1 (true), то за допомогою функції omp_set_num_threads() програміст може змінити значення контрольної змінної nthreads-var (що відповідає змінній оточення OMP_NUM_THREADS), тобто змінити під час виконання програми кількість потоків, що будуть породжуватись при вході у паралельну область. Значення змінної оточення OMP_DYNAMIC можна дізнатись функцією omp_get_dynamic() та змінити функцією omp_set_dynamic().

Вказати кількість програмних потоків, які будуть паралельно виконувати відповідний блок коду, можна також за допомогою опції num_threads директиви parallel (рис.4.11).

```
#pragma omp parallel num_threads(8)
{
  int id = omp_get_thread_num();
  x = id;
  printf("hello world (%d)\n", id);
}
```

Рис.4.11. Приклад використання директиви parallel num_threads().

Внутрішня контрольна змінна (ICV) thread-limit-var задає максимальну кількість потоків, що можуть виконуватись паралельно в OpenMP-програмі. Значення цієї змінної можна дізнатись функцією omp_get_thread_limit() та встановити за допомогою відповідної змінної оточення OMP_THREAD_LIMIT.

Функція omp_get_max_threads() повертає верхню межу кількості потоків, які можуть бути створенні після "виконання" директиви parallel (якщо не вказано вираз num_threads). Значення, яке повертає omp_get_max_threads(), є значенням першого елемента контрольної

змінної nthreads-var (значенням цієї змінної є список). Значення, яке повертає omp_get_max_threads(), може бути використано для динамічного виділення достатнього обсягу пам'яті для виконання всіх потоків у групі, створеній для виконання відповідної активної паралельної області.

Функція omp_get_num_threads() повертає кількість паралельних потоків, що виконуються у поточній активній паралельній області програми (тобто повертає значення внутрішньої контрольної змінної (ICV) team-size-var). Якщо викликати omp_get_num_threads() у послідовній частині програми, то вона поверне 1.

Функція omp_get_thread_num() повертає номер потоку, що знаходиться в поточній групі, створеній для виконання відповідної активної паралельної області прогами (тобто повертає значення внутрішньої контрольної змінної (ICV) thread-num-var). Іншими словами ця функція повертає номер викликаючого потоку, який разом з іншими потоками виконує паралельну область програми. Номер потоку - це ціле число від 0 до значення на одиницю менше результату, який повертаєт omp_get_num_threads() включно. Номер головного потоку програми дорівнює 0. Функція omp_get_thread_num() повертає 0, якщо вона викликана у послідовній частині програми.

4.1.6. Управління класами змінних в OpenMP

Всі змінні, що використовуються в паралельній області, можуть належати до одного з двох класів - бути або спільними для всіх потоків, або локальними (для кожного потоку створюється своя копія змінної). Приналежність змінної до класу спільних задається в директиві parallel виразом shared (рис.4.12), а приналежність до класу локальних - виразом private (рис.4.12). Кожна спільна змінна існує лише в одному примірнику і доступна для кожного потоку під одним і тим же ім'ям. Для кожної локальної змінної в кожному потоці існує окремий примірник даної змінної, доступний лише для цього потоку.

```
int count;
int a1, a2;
.   .   .

#pragma omp parallel private(a1,a2) shared(counter)
{
  a1 = a2 + 29;
  .   .   .
  counter++;
}
```

Рис.4.12. Приклад опису спільних та локальних змінних в паралельній області програми.

Припустимо, що паралельна область програми містить виклик функції omp_get_thread_num() (рис.4.13). Якщо змінну id в цій паралельній області було описано як локальну (private), то на виході буде отримано весь набір чисел від 0 до OMP_NUM_THREADS-1, що будуть йти у довільному порядку, але кожне з чисел буде зустрічатись лише один раз. Якщо ж змінна id була описана як спільна (shared), то ми отримаємо послідовність з OMP_NUM_THREADS чисел, що лежать в діапазоні від 0 до OMP_NUM_THREADS-1. Які саме числа і у якій кількості повторень будуть в послідовності визначити заздалегідь неможливо. Існує навіть можливість, що це буде OMP_NUM_THREADS однакових чисел X. Це може бути наслідком того, що всі потоки виконали перший рядок коду, але потім їх виконання з якоїсь причини було призупинено. В цей же час потік з номером X присвоює це значення змінній id, і тому, що ця змінна є спільною, відповідне значення буде виведено кожним потоком.

```
  {
    id = omp_get_thread_num();
    printf("%d\n", id);
  }
```

Рис.4.13. Приклад блоку коду, що відповідає паралельній області програми.

Вираз default(shared) в директиві parallel встановить приналежність усіх змінних в паралельній області до спільних (shared). Вираз default(none) змушує програміста в явний спосіб вказати приналежність кожної змінної в паралельній області до того чи іншого класу (якщо програміст забуде це зробити, то про це йому "нагадає" компілятор). Цим виразом можна скористатись для зменшення помилок при роботі зі спільними даними у потоках.

4.1.7. Розподіл обчислювальних завдань між потоками

Створені для виконання паралельної області потоки виконують один і той самий блок коду. Однак в багатьох випадках потрібно якимось чином розподілити між потоками виконання різних обчислювальних завдань. Це можна зробити двома основними способами:

1) з використанням функції omp_get_thread_num(),

2) з використанням виразів sections та section директиви parallel.

Перший спосіб передбачає визначення номеру потоку у групі потоків, створених для виконання відповідної паралельної області програми. Цей номер повертає функція omp_get_thread_num(). Додатково можна дізнатись поточну кількість потоків у групі викликом функції omp_get_num_threads(). В залежності від номеру потоку, йому призначається виконання того чи іншого обчислювального завдання (рис.4.14).

```
  #pragma omp parallel
  {
    id = omp_get_thread_num();
    if (id == 1) {
          < код для потоку з номером 1 >
    }
    else if (id == 2) {
          < код для потоку з номером 2 >
    }
    .   .   .
  }
```

Рис.4.14. Приклад розподілу обчислювальних завдань між потоками з використанням функції omp_get_thread_num().

Другий більш зручний спосіб передбачає використання виразів sections та section директиви parallel (рис.4.15). В даному випадку виразами section визначаються незалежні один від одного фрагменти коду (task1, task2, ...), які можуть бути виконані паралельно та в будь-якій послідовності початку виконання. Кожен з цих фрагментів буде виконаний лише одним потоком з групи створених для виконання паралельної області потоків.

```
...
#pragma omp parallel sections
{
    #pragma omp section
        { task1(); }

    #pragma omp section
        { task2(); }
```

```
    ...

    #pragma omp section
        { taskN(); }
}
// implicit barrier here
...
```

Рис.4.15. Приклад розподілу обчислювальних завдань між потоками з використанням виразів sections та section директиви parallel.

4.2. Розподіл обчислювального навантаження та синхронізація в OpenMP

4.2.1. Розподіл обчислювального навантаження між потоками

Одним з важливих завдань паралельного програмування є організація паралельного виконання ітерацій деякого циклу програмними потоками. Така можливість з'являється тоді, коли обчислення в окремих ітераціях циклу є частково або повністю незалежними. В OpenMP-програмах для розподілу ітерацій циклу між потоками використовуються директиви for та parallel for (рис.4.16, 4.17).

```
#pragma omp for [clause[ [,] clause] ... ] new-line
    loop-nest

#pragma omp parallel for [clause[ [,] clause] ... ] new-line
    loop-nest
```

Рис.4.16. Опис директив for та parallel for.

```
#pragma omp parallel [clauses]
{
    #pragma omp for [clauses]
    for (...)
    {
        // body
    }
}

#pragma omp parallel for [clauses]
for (...)
{
    // body
}
```

Рис.4.17. Схема використання директив for та parallel for.

Якщо в паралельній області знаходиться оператор циклу, то, відповідно до загального правила розпаралелювання в OpenMP, він буде виконаний усіма потоками, тобто кожний потік виконає всі ітерації даного циклу. На відміну від цього, для розподілу ітерацій циклу між різними потоками застосовується директива for, яка відноситься до оператору for, що йде відразу після директиви (рис.4.18). Якщо область паралельного виконання містить лише оператор циклу, то використовується директива parallel for (рис.4.19).

За допомогою виразу schedule директиви for можна визначити спосіб розподілу ітерацій циклу між потоками. Обраний спосіб розподілу ітерацій вказується в дужках як аргумент виразу schedule (рис.4.20). Окрім способу розподілу ітерацій у виразі schedule також можна вказати кількість ітерацій (chunk size), що виділяються окремому потоку (розмір блоку ітерацій, які буде виконувати один потік).

```
#pragma omp parallel default(none) shared(n, a, b, c, d, sum)
{
    #pragma omp for
    for (int i = 0; i < n; ++i)
        a[i] += b[i];

    #pragma omp for
    for (int i = 0; i < n; ++i)
        c[i] += d[i];
```

```
} // End of parallel region
```

Рис.4.18. Приклад використання директиви for в паралельній області коду, що задана директивою parallel.

```
/* Initialize the value of a large array in parallel,
   using each thread to do a portion of the work. */

void main(int argc, char **argv) {
    const int N = 100000;
    int i, a[N];
    .   .   .
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;
}
```

Рис.4.19. Приклад використання директиви parallel for.

```
    #pragma omp parallel for schedule(dynamic,10)
    for (i = 0; i < N; i++)
        a[i] = 2 * i;
```

Рис.4.20. Приклад використання директиви for з виразом schedule: динамічний розподіл блоків ітерацій циклу (по 10 ітерацій у кожному блоці).

Розглянемо способи розподілу ітерацій:

1. static [, m] - блочно-циклічний розподіл ітерацій: перший блок з m ітерацій виконує перший потік, другий блок - другий потік і т.д. до останнього потоку, потім розподіл знову починається з першого потоку (тобто блоки ітерацій розподіляються між потоками за алгоритмом round robin); якщо значення m не вказано, то загальна кількість ітерацій розділяється на приблизно однакові за розміром блоки, з врахуванням кількості потоків (загальна кількість ітерацій поділена на кількість потоків у групі);

2. dynamic [, m] - динамічний розподіл ітерацій з фіксованим розміром блоку: спочатку всі потоки отримують порції з m ітерацій, а потім кожний потік, що завершив роботу, отримує наступну порцію знову ж таки з m ітерацій; тобто кожен потік виконує свій блок ітерацій, після чого отримує інший блок, і так доти, доки не залишиться жодного блоку для розподілу; якщо значення m не вказано, то за замовчуванням воно дорівнює 1;

3. guided [, m] - динамічний розподіл ітерацій блоками, розмір яких поступово зменшується до величини m; аналогічно розподілу dynamic, але розмір виділених блоків весь час зменшується, що в окремих випадках дозволяє більше ефективно збалансувати обчислювальне навантаження між потоками; поточний розмір блоку визначається як кількість неопрацьованих ітерацій, поділена на кількість потоків у групі (якщо ця величина більше m); якщо значення m не вказано, то за замовчуванням воно дорівнює 1;

4. auto - рішення про спосіб розподілу делегується компілятору чи/або системі під час виконання програми (тобто обирається спосіб, встановлений у відповідній реалізації OpenMP за замовчуванням);

5. runtime - спосіб розподілу ітерацій циклу вибирається під час роботи програми в залежності від значення змінної OMP_SCHEDULE (відповідне значення містить внутрішня контрольна змінна (ICV) run-sched-var); це значення можна отримати викликом функції omp_get_schedule() і змінити викликом функції omp_set_schedule(); за замовчуванням OMP_SCHEDULE задає спосіб static.

Якщо в директиві for не вказано вираз schedule, то спосіб розподілу ітерацій між потоками визначається вмістом внутрішньої контрольної змінної (ICV) def-sched-var (рис.4.21), яка за замовчуванням задає спосіб static. Якщо у виразі schedule вказано спосіб

runtime, то спосіб розподілу ітерацій визначається вмістом внутрішньої контрольної змінної (ICV) run-sched-var (рис.4.21), яка за замовчуванням також задає спосіб static.

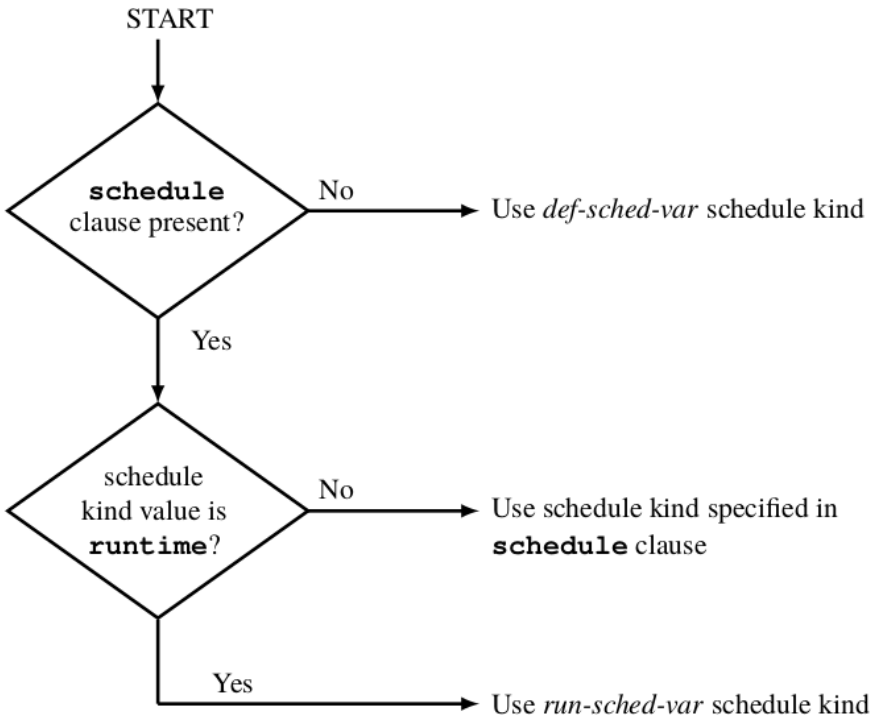

Рис.4.21. Алгоритм визначення способу розподілу ітерацій циклу між потоками.

В кінці паралельного циклу відбувається неявна бар'єрна синхронізація паралельного виконання потоків: їх подальше виконання відбувається тільки тоді, коли всі вони досягнуть цієї точки. Якщо в подібній затримці немає необхідності, то вираз nowait дозволяє потокам, які вже дійшли до кінця циклу продовжити виконання без синхронізації з іншими потоками.

На рис.4.22 наведено приклад, в якому зовнішній цикл оголошений паралельним, причому буде використано блочно-циклічний (static) розподіл ітерацій по дві ітерації в блоці (chunk size дорівнює 2). Щодо внутрішнього циклу ніяких вказівок немає, тому він буде виконуватися послідовно кожним потоком.

```
#include <stdio.h>
#include <omp.h>

void main() {
  const int N = 100000;
  const int M = 100000;
  int i, j, a[N][M], b[N][M];

  #pragma omp parallel for schedule(static,2)
    for (i = 1; i < N; i++)
      for (j = 1; j < M; j++)
        a[i][j] = (b[i][j-1] + b[i-1][j]) / 2.0;
}
```

Рис.4.22. Приклад використання директиви for з виразом schedule.

4.2.2. Виконання блоку коду підмножиною потоків

В деяких випадках є необхідність у тому, щоб частина програмного коду у паралельній області виконувалась не усіма потоками з групи породжених потоків, а лише підмножиною цих потоків. Для цього можна скористатись двома директивами: single (заданий блок коду виконується лише одним потоком) та masked (заданий блок коду виконується лише потоками із вказаними номерами).

Директива single (рис.4.23, рис.4.24) вказує, що заданий блок коду виконується лише одним із потоків у групі (не обов'язково основним потоком) у відповідному контексті. Інші потоки в групі, які не виконують цей блок коду, чекають на неявному бар'єрі в кінці блоку single (якщо в директиві single не вказано вираз nowait). Тобто директивою single вказується блок коду, який виконується лише одним потоком, з синхронізацією усіх потоків на неявному бар'єрі після виконання блоку. Спосіб вибору потоку для виконання заданого директивою single блоку визначається реалізацією OpenMP. Як правило, заданий директивою single блок коду виконується тим потоком, що першим дійшов до даної точки програми.

```
#pragma omp single [clause[ [,] clause] ... ] new-line
    structured-block
```

Рис.4.23. Опис директиви single.

```
void single_example()
{
      #pragma omp parallel
      {
            #pragma omp single
            printf("Beginning work1.\n");

            work1();

            #pragma omp single
            printf("Finishing work1.\n");

            #pragma omp single nowait
            printf("Finished work1 and beginning work2.\n");

            work2();
      }
}
```

Рис.4.24. Приклад використання директиви single.

Директива masked (рис.4.25) визначає блок коду, який виконується підмножиною потоків з групи потоків, породжених для виконання паралельної області коду. Тільки ті потоки, які відповідають заданому фільтру (вираз filter), беруть участь у виконанні заданого “замаскованого” блоку коду. Інші потоки з групи не виконують заданий блок. Ні на вході, ні на виході з заданого блоку бар’єрна синхронізація не виконується (на відміну від використання директиви single, де на виході з блоку виконується бар’єрна синхронізація усіх потоків з групи).

```
#pragma omp masked [ filter(integer-expression) ] new-line
   structured-block
```

Рис.4.25. Опис директиви masked.

Якщо в директиві masked вказано вираз filter, то його параметр задає номер потоку в групі, який буде виконувати заданий блок коду. Якщо вираз filter відсутній, то номер потоку приймається рівним нулю, і таким чином заданий блок коду виконує лише основний потік. Якщо у виразі filter використовується змінна, то її значення береться з контексту, в якому викликається директива masked. Потрібно пам'ятати, що, якщо параметром filter є якійсь вираз, то в різних потоках він може приймати різні значення.

Якщо декілька потоків з групи виконують заданий "замаскований" блок коду, то цей блок має містити відповідні засоби синхронізації для уникнення некоректної роботи зі спільними даними (data races). Директива master, яка більше не використовується (deprecated), ідентична директиві masked без виразу filter.

4.2.3. Бар'єрна синхронізація потоків

Одним з найбільш розповсюджених способів синхронізації є бар'єрна синхронізація. В OpenMP для бар'єрної синхронізації використовується директива barrier (рис.4.26). Усі потоки, дійшовши до цієї директиви, зупиняються і чекають поки усі інші потоки не дійдуть до цієї точки програми, після чого всі потоки одночасно продовжують виконання. Тобто директива barrier визначає явний бар'єр у точці паралельної області програми, де вказано цю директиву. На рис.4.27 наведено приклад використання бар'єрної синхронізації.

```
#pragma omp barrier new-line
```

Рис.4.26. Опис директиви barrier.

```
#include <omp.h>
#include <stdio.h>

 int main (int argc, char *argv[]) {
   int th_id, nthreads;

   #pragma omp parallel private(th_id)
   {
     th_id = omp_get_thread_num();
     printf("Hello World from thread %d\n", th_id);

     #pragma omp barrier
     if ( th_id == 0 ) {
       nthreads = omp_get_num_threads();
       printf("There are %d threads\n",nthreads);
     }
   }
   return 0;
 }
```

Рис.4.27. Приклад бар'єрної синхронізації.

Більшість директив OpenMP, в яких задається відповідний блок коду, передбачають неявну бар'єрну синхронізацію потоків в момент завершення цього блоку коду. Тобто при використанні більшості директив OpenMP треба враховувати той факт, що за замовчуванням після виконання блоку потоки будуть очікувати один одного на неявному бар'єрі. Змінити цю поведінку можна за допомогою виразу nowait, який вказується після директиви (рис.4.28). Якщо вираз nowait присутній, то неявний бар'єр в кінці блоку опускається. В цьому випадку потоки, які закінчили виконання блоку коду, переходять до виконання операторів, які слідують за блоком, не чекаючи, поки інші потоки у групі закінчать виконання блоку.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (int i = 0; i < n; ++i)
        a[i] += b[i];
}
```

Рис.4.28. Приклад використання виразу nowait.

4.2.4. Критичні секції коду та атомарні операції

Критична секція коду задається за допомогою директиви critical (рис.4.29). В кожний момент часу в критичній секції коду може перебувати лише один програмний потік з групи породжених для виконання паралельної області потоків (рис.4.30). Якщо критична секція вже виконується будь-яким потоком P, то всі інші потоки, які виконали директиву critical з цією ж назвою, будуть заблоковані, поки потік P не завершить виконання даної критичної секції. Як тільки P дійде до кінця заданого директивою critical блоку коду, один з заблокованих на вході потоків увійде в секцію. Якщо на вході в критичну секцію стоїть декілька потоків, то випадковим чином обирається один з них, а інші заблоковані потоки продовжують очікування. Усі неіменовані критичні секції за замовченням асоціюються з одним і тим же ім'ям (рис.4.31). В стандарті OpenMP передбачено використання в директиви critical виразу hint (рис.4.29), параметр якого може бути використаний реалізацією OpenMP для передачі додаткової інформації про параметри критичної секції (без порушення базової логіки її роботи).

```
#pragma omp critical [(name) [[,] hint(hint-expression)] ] new-line
   structured-block
```

Рис.4.29. Опис директиви critical.

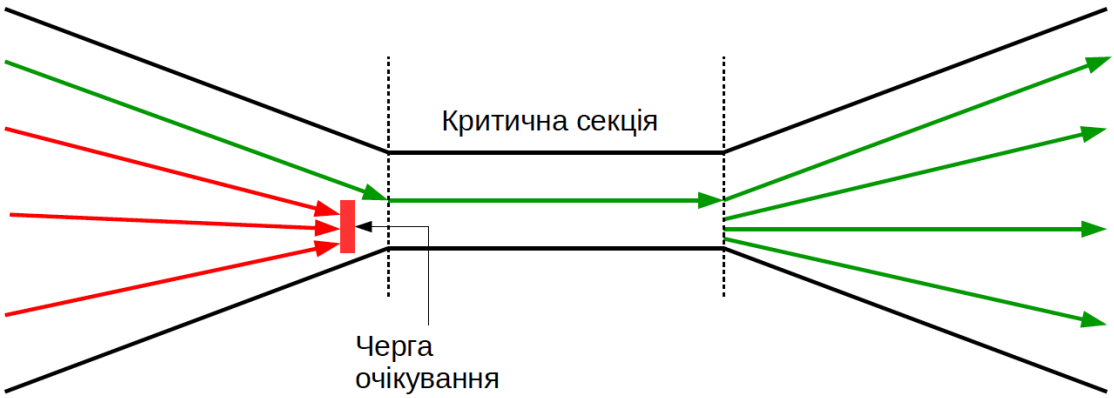

Рис.4.30. Схема використання критичної секції коду.

```
int x, count;

#pragma omp parallel shared(x, count)
{
  .   .   .
  #pragma omp critical
  {
    x = x + a[i];
    count++;
```

```
    }
    .   .   .
  }    /* end of parallel region */
```

Рис.4.31. Приклад використання директиви critical.

Окремим випадком використання критичних секцій на практиці є оновлення значень спільних змінних. Наприклад, якщо змінна sum є спільною і оператор виду sum++ знаходиться в паралельній області програми, то при одночасному виконанні даного оператору декількома потоками можна отримати некоректний результат. Щоб уникнути такої ситуації можна скористатися механізмом критичних секцій або директивою atomic (рис.4.32), яка спеціально передбачена для таких випадків. Директива atomic відноситься до оператору, який йде безпосередньо за нею, гарантуючи коректну роботу зі спільною змінною (рис.4.33).

```
#pragma omp atomic [clause[[,] clause] ... ] new-line
   statement
```

Рис.4.32. Опис директиви atomic.

```
  #pragma omp parallel
  {
    .   .   .
    #pragma omp atomic
    sum++;
    .   .   .
  }
```

Рис.4.33. Приклад використання директиви atomic.

4.3. Бібліотека паралельного програмування oneTBB

4.3.1. Організація паралельних обчислень за допомогою oneTBB

Бібліотека паралельного програмування oneAPI Threading Building Blocks (oneTBB) призначена для організації масштабованих паралельних обчислень з використанням мови програмування C++ [21,22]. Ініціатором розробки і першим розробником oneTBB була корпорація Intel (попередня назва цієї бібліотеки - Intel TBB). Для роботи з oneTBB не потрібні спеціальні розширення мови C++ або спеціальні компілятори. oneTBB призначений для підтримки та спрощення паралельного програмування з гнучким масштабуванням паралельної обробки великих наборів даних. Крім того, oneTBB підтримує вкладений паралелізм, тобто дозволяє створювати паралельні компоненти, що складаються з менших паралельних компонентів. При роботі з бібліотекою oneTBB програміст специфікує окремі обчислювальні завдання (tasks), які oneTBB в автоматичному режимі відображає у паралельні програмні потоки, забезпечуючи ефективне розпаралелювання програми під час її виконання.

Програмна бібліотека oneTBB - це, в першу чергу, бібліотека шаблонів C++, що полегшують розроблення паралельних програм з максимальним використанням переваг багатоядерних процесорів. Бібліотека складається із паралельних структур даних та алгоритмів, які дозволяють програмісту уникнути деяких труднощів, що виникають внаслідок використання низькорівневих бібліотек для роботи з потоками, таких як POSIX threads або Windows threads, в яких потоки створюються, синхронізуються та завершуються “вручну”. Натомість бібліотека oneTBB надає абстракцію доступу до безлічі паралельних процесорів, з представленням обчислень у вигляді незалежних завдань (tasks), які динамічно розподіляються між ядрами процесора під час виконання програми в автоматичному режимі з ефективним використанням кешу.

У програмі з використанням oneTBB передбачено створення, синхронізація та завершення завдань в рамках деякого графу залежностей між ними в рамках парадигм паралельного програмування високого рівня. Під час виконання програми, завдання виконуються окремими потоками відповідно до графу залежностей. Такий підхід забезпечує незалежність паралельного програмування від низькорівневих деталей роботи апаратної платформи паралельної комп'ютерної системи.

В роботі oneTBB реалізовано принцип "викрадення завдань" (task stealing, work stealing) для балансування обчислювального навантаження між ядрами процесора (рис.4.34), забезпечуючи тим самим їх ефективне використання та можливість масштабування паралельних обчислень. Спочатку обчислювальне навантаження рівномірно розподіляється між доступними ядрами процесора. Якщо одне ядро завершує свої завдання, тоді як інші ядра все ще мають значну кількість завдань в черзі, oneTBB перекидає частину завдань від одного із зайнятих ядер вільному ядру. Цей механізм дозволяє програмісту абстрагуватись від можливостей апаратної платформи паралельної комп’ютерної системи, за рахунок масштабування паралельних обчислень з використанням oneTBB на будь яку кількість доступних ядер процесора без змін у тексті програми та її машинному коді.

oneTBB, як і STL (Standard Template Library), побудована на основі шаблонів (templates). Це дає перевагу поліморфізму з низькими витратами, оскільки шаблони, як мовні конструкції часу компіляції, можуть бути ефективно оптимізовані сучасними компіляторами C++, забезпечуючи швидке виконання відповідних програм.

Перша версія oneTBB (на той час Intel TBB) була надана для використання корпорацією Intel 29 серпня 2006, через рік після випуску першого двоядерного процесора Intel Pentium D. Після цього майже кожний рік виходила нова версія або підверсія oneTBB (Intel TBB). На даний час остання версія - oneTBB 2021.5 (22/12/2021) входить до відкритого стандарту oneAPI, яким визначаються різні програмні інтерфейси (бібліотеки) для роботи з апаратним прискоренням паралельних обчислень та використання можливостей такого прискорення.

Комерційна версія oneTBB підтримує ОС Windows (XP або новішої версії), OS X (версія 10.5.8 або новіша) та ОС Linux, а також компілятор Intel C++ (версія 11.1 або новіша) та GNU

Compiler Collection (gcc). Крім того, в рамках версії oneTBB з відкритим кодом окрім наведених вище платформ також підтримуються платформи: Solaris, PowerPC, Xbox 360, QNX Neutrino та FreeBSD.

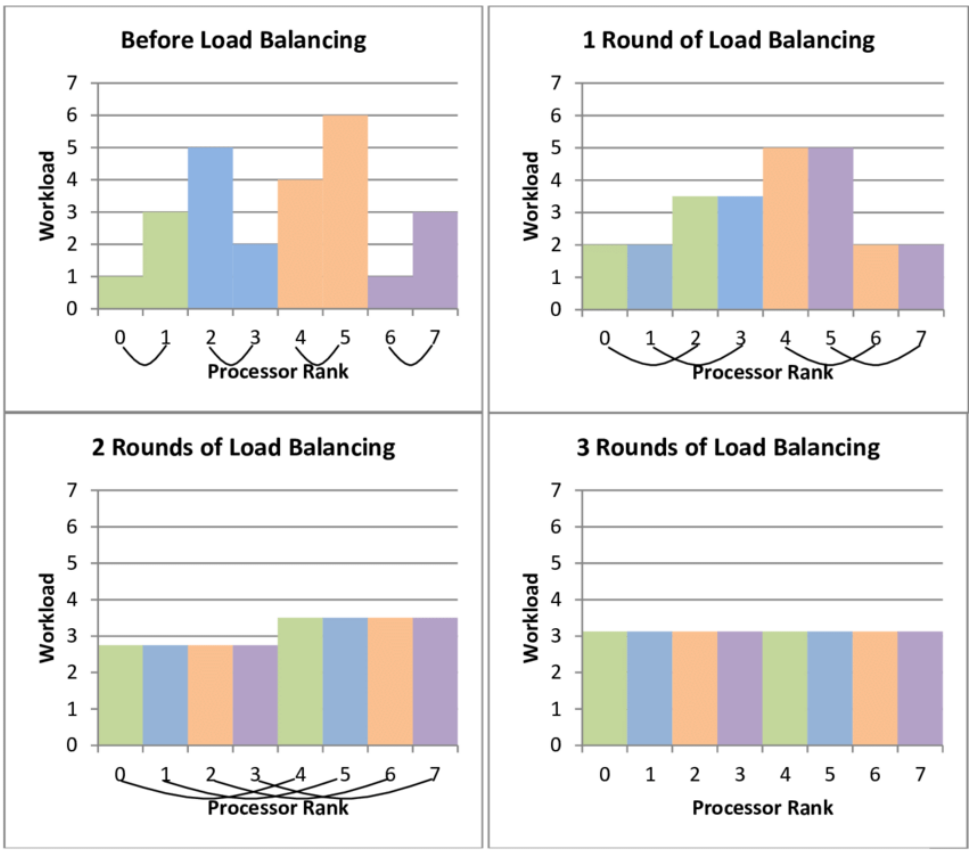

Рис.4.34. Приклад балансування обчислювального навантаження за принципом "викрадення завдань" (task stealing, work stealing).

4.3.2. Структура бібліотеки паралельного програмування oneTBB

Бібліотека паралельного програмування oneTBB складається з набору шаблонів класів та функцій, в яких реалізовано низку паралельних алгоритмів та структур даних. Зокрема oneTBB містить наступні основні компоненти паралельного програмування:

1) Базові паралельні алгоритми (Basic algorithms): parallel_for, parallel_reduce, parallel_scan (parallel_reduce - паралельний обрахунок суми, добутку, тощо; parallel_scan - паралельний обрахунок “префіксної” суми виду y[i]=y[i-1]+x[i]);

2) Складні паралельні алгоритми (Advanced algorithms): parallel_while, parallel_do, parallel_pipeline, parallel_sort;

3) Потокобезпечні контейнери (Containers): concurrent_queue, concurrent_priority_queue, concurrent_vector, concurrent_hash_map;

4) Масштабовані розподілювачі пам'яті (Scalable memory allocation): scalable_malloc, scalable_free, scalable_realloc, scalable_calloc, scalable_allocator, cache_aligned_allocator;

5) Засоби взаємного виключення (Mutual exclusion): mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, recursive_mutex;

6) Атомарні операції (Atomic operations): fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store;

7) Механізм логічного часу (Timing): глобальні часові мітки (time stamps);

8) Планувальник завдань (Task Scheduler): прямий доступ до управління створенням та активацією обчислювальних завдань (tasks).

При розробці паралельної програми з використанням oneTBB програміст починає з аналізу обчислювальної задачі з точки зору можливостей реалізації відповідних паралельних алгоритмів на основі шаблонів “готових рішень”, які надає oneTBB (рис.4.35), і здійснює пошук варіантів розпаралелювання програмного коду засобами oneTBB [21,22].

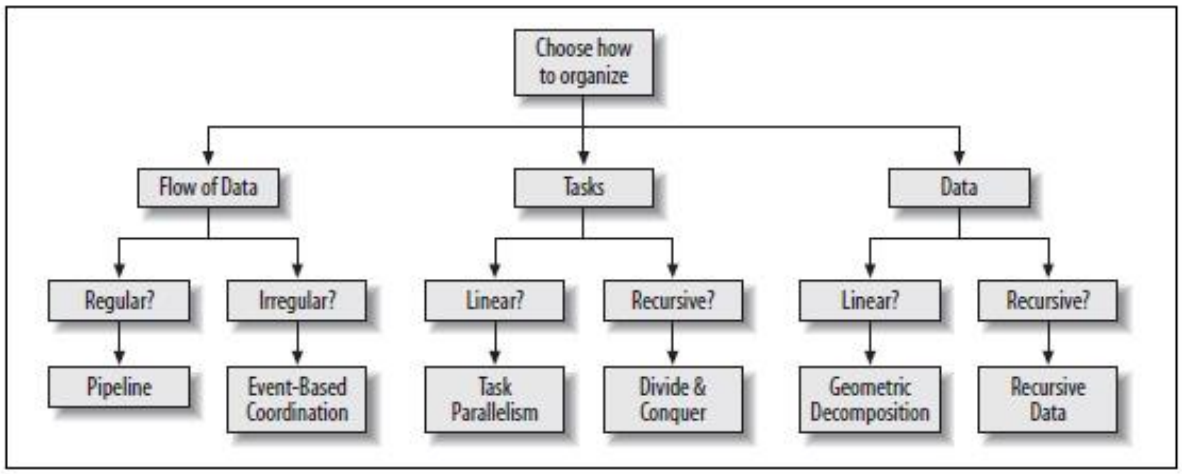

Рис.4.35. Схема послідовності аналізу обчислювальної задачі з точки зору паралельного програмування з використанням бібліотеки oneTBB.

На рис.4.36. наведено приклад програми з використанням oneTBB, в якій створюється два потоки, кожний з яких виводить повідомлення “Hello world!”.

```
#include "tbb/tbb.h"
#include “tbb/task_scheduler_init.h”
#include <cstdio>

using namespace tbb;

class say_hello
{
   const char* id;
   public:
      say_hello(const char* s) : id(s) { }
      void operator( ) ( ) const
      {
         printf("Hello world! from task %s\n",id);
      }
};

int main( )
{
   task_scheduler_init init; // Initializing the library

   task_group tg;
   tg.run(say_hello("1")); // Spawn 1st task and return
   tg.run(say_hello("2")); // Spawn 2nd task and return
   tg.wait( );             // Wait for tasks to complete
}
```

Рис.4.36. Приклад програми з використанням oneTBB.

4.3.3. Паралельне виконання ітерацій циклу за допомогою tbb::parallel_for

Для паралельного виконання ітерацій деякого циклу декількома програмними потоками в oneTBB використовується шаблон tbb::parallel_for (за принципом аналогічним використанню директиви parallel for в OpenMP). Припустимо, нам потрібно застосувати функцію Foo() до кожного елемента масиву, і функція Foo() така, що кожен елемент можна обробляти незалежно від інших. Відповідний послідовний програмний код наведено на рис.4.37.

```
void SerialApplyFoo (float a[], size_t n) {
    for(size_t i=0; i!=n; ++i)
        Foo(a[i]);
}
```

Рис.4.37. Послідовна обробка елементів масиву в циклі.

В даному випадку простір ітерацій (iteration space) має тип size_t і знаходиться в діапазоні від 0 до n. Функція-шаблон tbb::parallel_for() розбиває простір ітерацій на фрагменти і запускає кожен фрагмент в окремому потоці. Першим кроком у розпаралелюванні даного циклу (рис.4.37) є перетворення циклу у форму, яка дозволяє обробляти окремий фрагмент (chunk) простору ітерацій. Відтак цикл реалізується у вигляді функціонального об'єкту (function object) в стилі STL, в якому operator() обробляє заданий фрагмент простору ітерацій (рис.4.38).

```
#include "tbb/tbb.h"

using namespace tbb;

class ApplyFoo {
    float *const my_a;
public:
    void operator()(const blocked_range<size_t>& r) const {
        float *a = my_a;
        for(size_t i=r.begin(); i!=r.end(); ++i)
            Foo(a[i]);
    }

    ApplyFoo( float a[] ):
        my_a(a)
    {}
};
```

Рис.4.38. Опис функціонального об'єкту ApplyFoo.

В якості аргументу operator() використовується спеціальний клас-шаблон blocked_range<T>, який надається бібліотекою oneTBB. Він описує одновимірний простір ітерацій над типом даних T. Клас-шаблон parallel_for може працювати також з іншими типами просторів ітерацій. Наприклад, в oneTBB надається клас-шаблон blocked_range2d для двовимірних просторів ітерацій.

Другим кроком є використання описаного функціонального об'єкту ApplyFoo у функції-шаблоні parallel_for() для паралельного виконання блоків ітерацій циклу (фрагментів простору ітерацій) кількома програмними потоками (рис.4.39). Заданий blocked_range простір ітерацій від 0 до n буде поділений на фрагменти, після чого відповідні блоки ітерацій циклу будуть виконуватись паралельно створеними для цього потоками.

```
#include "tbb/tbb.h"

void ParallelApplyFoo(float a[], size_t n) {
    parallel_for(blocked_range<size_t>(0, n), ApplyFoo(a));
}
```

Рис.4.39. Паралельне виконання блоків ітерацій циклу (фрагментів простору ітерацій) за допомогою tbb::parallel_for.

На рис.4.40. наведено приклад використання tbb::parallel_for, в якому елементи масиву myArray обробляються функцією Calculate() паралельно. В цьому прикладі у виклику функції-шаблону parallel_for() в явний спосіб вказано розмір блоків ітерацій (blocked_range<int>(0, SIZE, **500**)), які будуть "роздаватись" програмним потокам для виконання. Якщо не вказувати розмір блоку ітерацій, то його розмір обирається oneTBB автоматично.

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"

using namespace tbb;

// Кількість елементів вектора
const int SIZE = 10000000;

// Клас-обробник
class CalculationTask
{
private:
        double *myArray;
public:
        // Оператор () виконується над діапазоном з простору ітерацій
        void operator()(const blocked_range<int> &r) const
        {
                for (int i = r.begin(); i != r.end(); i++)
                        Calculate(myArray[i]);
        }

        // Конструктор
        CalculationTask (double *a) : myArray(a) { }
};


int main()
{
        double *myArray = new double[SIZE];

        // Ініціалізація бібліотеки TBB
        task_scheduler_init init;

        // Запуск паралельного алгоритму for
        parallel_for(
                blocked_range<int>(0, SIZE, 500),
                CalculationTask(myArray));

        return 0;
}
```

Рис.4.40. Приклад використання tbb::parallel_for.

4.3.4. Розподіл обчислювального навантаження між потоками у tbb::parallel_for

Останнім аргументом функції-шаблону tbb::parallel_for() можна явно вказати спосіб розподілу блоків ітерацій циклу (обчислювального навантаження) між потоками, що будуть створені для їх паралельного виконання (рис.4.41). При цьому третім параметром виразу blocked_range<size_t>(0, n ,**G**) вказано розмір блоку ітерацій (grain size). Під час виконання програми oneTBB буде формувати порції (chunks) обчислювального навантаження для кожного потоку у вигляді певної кількості ітерацій з врахуванням вказаного розміру блоку **G**. Якщо розмір блоку не вказано, то береться відповідне значення за замовчуванням.

```
#include "tbb/tbb.h"

void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0, n ,G),
                 ApplyFoo(a),
                 simple_partitioner());
}
```

Рис.4.41. Приклад виклику parallel_for() з вказанням способу розподілу блоків ітерацій simple_partitioner().

В tbb::parallel_for реалізовані наступні способи розподілу обчислювального навантаження між потоками:

1) auto_partitioner() - автоматичний розподіл блоків ітерацій циклу між потоками; розмір порції (chunk size) визначається автоматично на основі евристик, закладених у oneTBB, в тому числі виходячи з потреб розподілу обчислювального навантаження (load balancing) між ядрами процесора; якщо вказано розмір блоку G, то розмір порції буде складати: chunkSize ≥ [G/2].

2) simple_partitioner() - простий розподіл; відключає автоматичний розподіл блоків ітерацій циклу; розмір порції (chunk size) буде обиратись oneTBB під час виконання програми в діапазоні: G ≥ chunkSize ≥ [G/2]; якщо розмір блоку (grain size) G не вказаний, то береться значення за замовчуванням, рівне 1.

3) affinity_partitioner() - розподіл з прив'язкою до ядер процесора; автоматичний розподіл блоків ітерацій циклу з додатковою оптимізацією з точки зору прив'язки потоків до ядер процесору ("optimizes for cache affinity"); цей варіант розподілу доцільно використовувати, коли кількість ядер процесора n>2.

При виборі розміру блоку ітерацій (grain size) і відповідної порції ітерацій (chunk size) потрібно враховувати два фактори (рис.4.42):

1) Якщо порція занадто мала, то "накладні витрати" на забезпечення розпаралелювання (робота планувальника, переключення контексту, тощо) можуть перекрити виграш від паралельного виконання ітерацій циклу;

2) Якщо порція занадто велика, то це знижує ефект/виграш від розпаралелювання виконання ітерацій циклу.

Відтак розмір блоку ітерацій потрібно обирати таким, щоб він не був ні занадто малим, ні занадто великим (рис.4.43), забезпечуючи мінімальний час виконання паралельної програми.

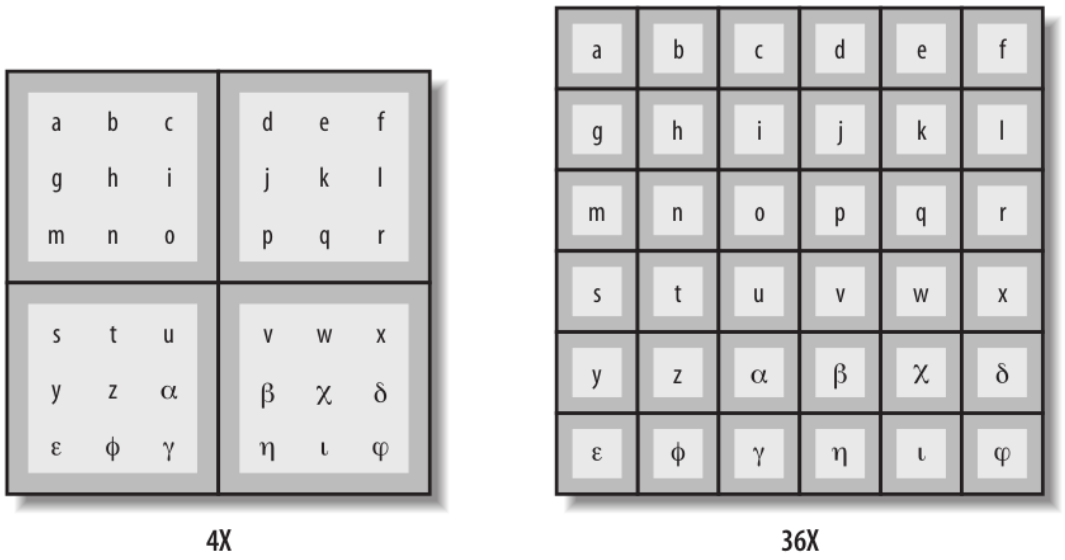

Рис.4.42. Приклад двох варіантів співвідношення паралельних обчислень та витрат на їх забезпечення в залежності від кількості потоків.

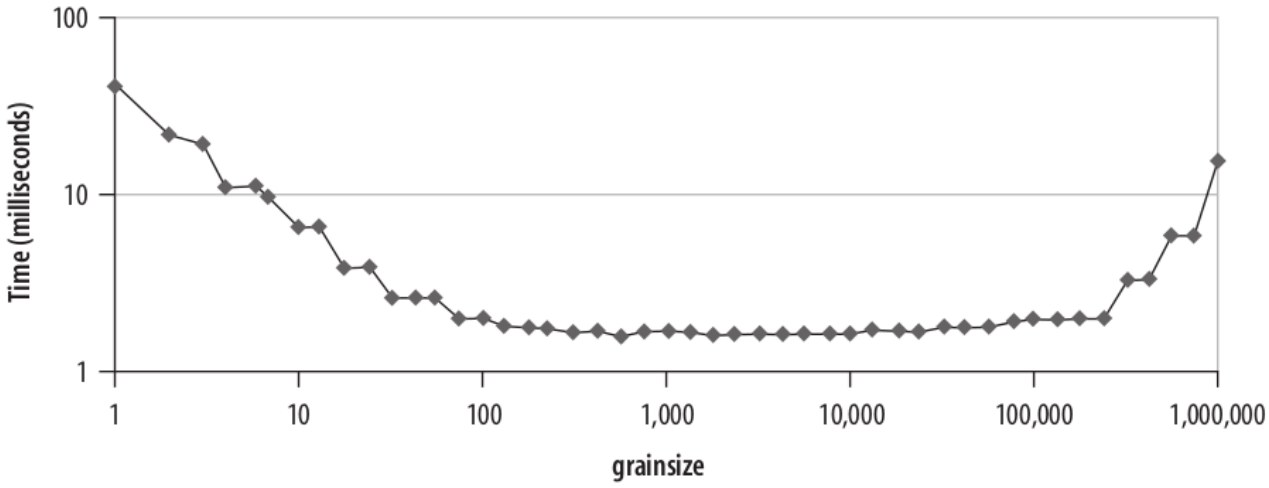

Рис.4.43. Приклад залежності часу виконання parallel_for
від розміру блоку (grainsize, логарифмічна шкала).

Контрольні питання

1. Для чого призначений інтерфейс прикладного програмування OpenMP?
2. Що є одиницею розпаралелювання програми у разі використання інтерфейсу прикладного програмування OpenMP?
3. Який основний елемент інтерфейсу прикладного програмування OpenMP використовується для специфікації схеми розпаралелювання програмного коду?
4. Яка директива OpenMP використовується для вказання паралельної частини програми на мові програмування C?
5. Що відбувається з програмними потоками після завершення паралельної ділянки (блоку коду) OpenMP-програми?
6. За допомогою якої функції можна визначити в якій своїй частині – послідовній чи паралельній виконується OpenMP-програма?
7. Які директиви OpenMP використовуються для розподілу обчислювальних завдань між програмними потоками?
8. За допомогою якої функції можна визначити номер програмного потоку в паралельній частині OpenMP-програми?
9. Як визначити кількість програмних потоків, що виконуються у поточній активній паралельній області OpenMP-програми?
10. Яка змінна оточення визначає максимальну кількість програмних потоків, що будуть виконувати вказану директивами OpenMP паралельну область програми?
11. Який вираз використовується для управління кількістю програмних потоків, що породжуються для виконання паралельної частини OpenMP-програми?
12. Якими виразами задається приналежність змінної до класу спільних для всіх потоків чи до класу локальних у всіх потоках в директиві parallel OpenMP-програми?
13. За допомогою якої директиви здійснюється управління розподілом обчислювального навантаження між програмними потоками в OpenMP?
14. Для чого призначена бібліотека програмування oneTBB?
15. З яких основних компонент складається бібліотека паралельного програмування oneTBB?
16. Які базові паралельні алгоритми реалізовані у бібліотеці паралельного програмування oneTBB?
17. Для чого використовується програмний шаблон tbb::parallel_for бібліотеки паралельного програмування oneTBB?
18. Як змінюється виграш від розпаралелювання виконання ітерацій циклу в залежності від розміру блоку ітерацій (grain size) і відповідної порції ітерацій (chunk size) в програмному шаблоні tbb::parallel_for?

5. Паралельне програмування в розподілених системах

5.1. Програмний інтерфейс MPI

5.1.1. Організація обчислень в розподіленій системі за допомогою MPI

Message Passing Interface (MPI, “Інтерфейс передачі повідомлень”) – це стандартизований програмний інтерфейс передачі повідомлень, призначений для організації паралельних обчислень в розподілених комп’ютерних системах (рис.5.1). Стандарт MPI визначає синтаксис та семантику бібліотечних функцій для використання у паралельних програмах для розподілених системи (в тому числі кластерів) на мовах програмування C, C++ та Fortran [23-29]. З точки зору класифікації типів зв’язку у розподілених комп’ютерних системах, стандартом MPI задається синхронний та асинхронний нерезидентний зв’язок.

Програмний інтерфейс MPI забезпечує побудову віртуальної топології обчислень, синхронізацію та зв’язок в режимі обміну повідомленнями між набором паралельних обчислювальних процесів, які виконуються на вузлах розподіленої системи (рис.5.2). Одиниця розпаралелювання MPI-програми - це обчислювальний процес. В системах високопродуктивних обчислень, як правило, одному обчислювальному елементу (ядру або процесору) ставлять у відповідність один процес MPI-програми, що забезпечує максимальну продуктивність паралельних обчислень. MPI передбачає використання базових комп’ютерних мереж або швидкісних інтерфейсів між вузлами і не передбачає нічого, що нагадує комунікаційні сервери. Крім того, він передбачає, що серйозні збої в системі, такі як аварії процесів або сегментів мережі, фатальні й не можуть бути відновлені автоматично.

Процеси об'єднуються в групи, які задають топологію паралельних обчислень. Допускаються вкладені групи та перекриття груп. Усередині групи всі процеси пронумеровані. Номер процесу в межах групи - це унікальне додатне ціле число. З кожною групою асоціюється свій комунікатор (communicator), який можна розглядати як агента, який забезпечує та контролює взаємодію процесів у групі. Тому при здійсненні пересилання повідомлення між процесами необхідно вказувати ідентифікатор групи (ідентифікатор комунікатора), всередині якої проводиться це пересилання. Всі процеси містяться в групі з визначеним ідентифікатором MPI_COMM_WORLD. Кожен окремий процес входить у свою персональну групу MPI_COMM_SELF. В групі з ідентифікатором MPI_COMM_NULL немає жодного процесу.

MPI передбачає, що зв'язок (обмін повідомленнями) відбувається в межах певної групи процесів. Кожна група має свій унікальний ідентифікатор комунікатора, і кожен процес в групі має свій локальний ідентифікатор. Відтак пара ідентифікаторів (комунікатор, процес) однозначно визначає відправника або отримувача повідомлення, і використовується замість адреси транспортного рівня. Топологію паралельних обчислень в MPI можуть формувати одночасно декілька груп процесів, які перекриваються чи/або вкладаються одна в одну.

Атрибутами окремого повідомлення в MPI є: ідентифікатор комунікатора (групи процесів), номер процесу-відправника, номер процесу-отримувача і ідентифікатор (тег) повідомлення. Ідентифікатор повідомлення (message tag) - атрибут повідомлення, що є цілим невід'ємним числом, яке лежить в діапазоні від 0 до 32767. Ідентифікатор (тег) повідомлення використовується для прив’язки точки відправки повідомлення до точки отримання повідомлення в межах реалізованої програмістом схеми обміну повідомленнями та топології паралельних обчислень.

Стандарт MPI містить більше 500 функцій, які входять у наступні основні групи:

1) службові функції,
2) функції відправки-отримання повідомлень між процесами,
3) функції колективної взаємодії та синхронізації процесів,
4) функції для роботи з групами процесів,
5) функції для роботи з типами та структурами даних.

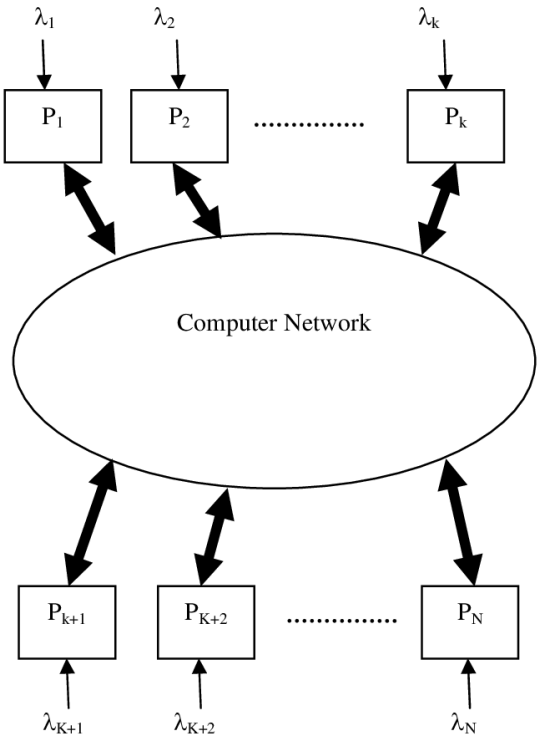

Рис.5.1. Узагальнена схема розподіленої комп'ютерної системи.

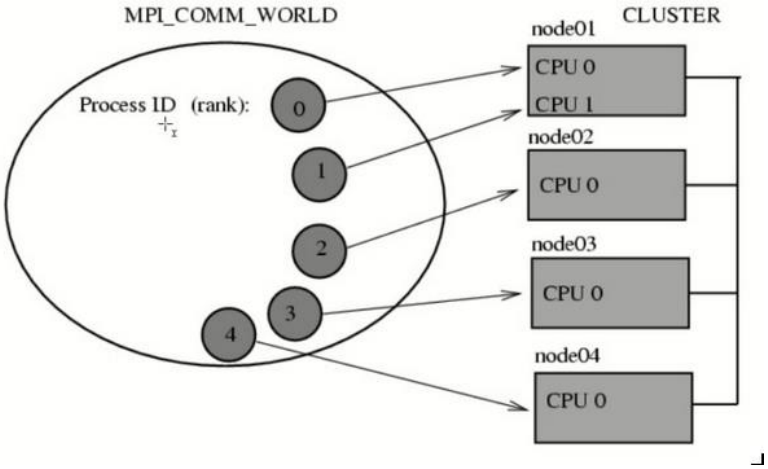

Рис.5.2. Схема організації паралельних обчислень в MPI.

Робота над стандартом MPI розпочалась у 1992 році. Розробкою стандарту займається організація MPI Forum (www.mpi-forum.org). Історія версій стандарту MPI: MPI-1.1 (1995), MPI-1.2 (1996), MPI-2.0 (1997), MPI-1.3 (2008), MPI-2.2 (2009), MPI-3.0 (2012), MPI-3.1 (2015), MPI-4.0 (2021). Крім цього іноді використовуються позначення, які відображають покоління стандарту: MPI-1 (MPI-1.3), MPI-2 (MPI-2.2), MPI-3 (MPI-3.1). В 2020 році розроблено черговий варіант стандарту MPI 4.0, який був затверджений MPI Forum у 2021 р. Стандарт MPI-1.2 підтримується більшістю сучасних реалізацій MPI, в той час як підтримка стандарту MPI-2.1 є більш обмеженою.

MPI часто порівнюють з "Паралельною Віртуальною Машиною" (Parallel Virtual Machine, PVM) - програмним середовищем для розподілених паралельних обчислень з підсистемою передачі повідомлень, розробленою в 1989 році. PVM була одною з систем, яка мотивувала необхідність створення стандарту передавання повідомлень для організації паралельних обчислень. Моделі програмування на рівні програмних потоків з використанням

спільної пам'яті (такі як Pthreads та OpenMP) та паралельне програмування на основі обміну повідомленнями (PVM, MPI) можуть розглядатись, як доповнюючі один одного підходи та використовуватись разом в рамках гібридної моделі паралельної комп'ютерної системи (рис.5.3).

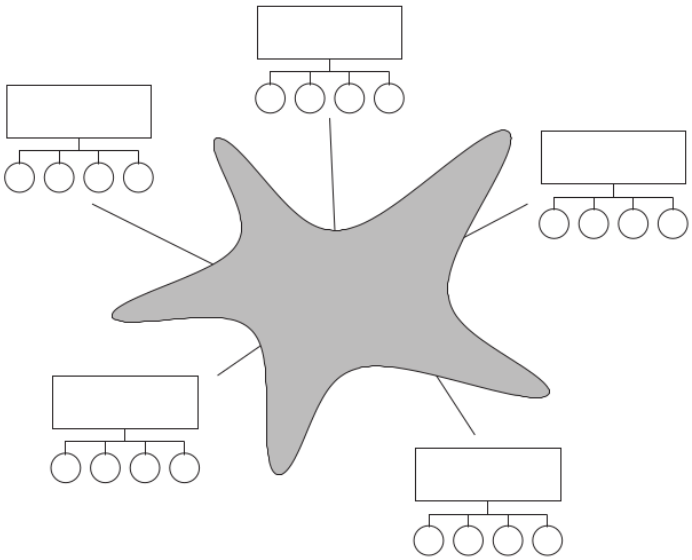

Рис.5.3. Гібридна модель паралельної комп'ютерної системи: в межах одного вузла апаратна пам'ять (прямокутник) є спільною для обчислювальних елементів (кола), і одночасно з цим в масштабі всієї розподіленої системи у кожного вузла - своя локальна пам'ять.

Існує кілька реалізацій стандарту MPI у вигляді програмних бібліотек, які використовуються в режимі проміжного програмного забезпечення (middleware). Основними мовами програмування, для яких розробляється та реалізується стандарт MPI, є C/C++ та Fortran. Крім цього існують програмні інтерфейси реалізацій MPI для мов програмування Java, Python, R, Ruby та ін. Серед відомих реалізацій MPI можна відмітити: MPICH (історично перша реалізація), LAM/MPI, MP_Lite, Open MPI та ін. На даний час найбільш широко використовуються реалізації MPICH (v.4.0.2, 2022) та Open MPI (v.4.1.3, 2022), остання з яких часто використовується на суперкомп'ютерах з рейтингу TOP-500.

5.1.2. Основні службові функції MPI

При використанні MPI, як правило, передбачається, що нові паралельні процеси створюються стандартними засобами відповідної операційної системи за допомогою інтерфейсу системних викликів. В межах окремого процесу задається паралельна частина, на протязі виконання якої процес є підключеним до MPI. По завершенню паралельної частини, процес відключається від MPI. Початок та кінець паралельної частини програми задаються функціями MPI_Init() та MPI_Finalize().

Функція MPI_Init() (рис.5.4) ініціалізує паралельну частину програми. Реальна ініціалізація для кожного процесу (примірнику програми) виконується не більше одного разу, а якщо MPI вже був ініціалізований, то ніякі дії не виконуються і відбувається негайне повернення з функції. Усі інші функції MPI можуть бути викликані тільки після виклику MPI_Init(). На вхід функції MPI_Init(), як правило передаються аргументи функції main() (рис.5.5). Функція повертає: в разі успішного виконання - MPI_SUCCESS, інакше - код помилки. Такі самі значення повертають і майже всі інші функції MPI.

```
int MPI_Init(int* argc, char*** argv)
```

```
int MPI_Finalize(void)
```

Рис.5.4. Опис функцій MPI_Init() та MPI_Finalize().

```
int main(int argc, char** argv)
{
        ...
      // початок паралельної частини програми
      MPI_Init(&argc, &argv);
        ...
      MPI_Finalize();
      // кінець паралельної частини програми
        ...
}
```

Рис.5.5. Приклад визначення паралельної частини програми функціями MPI_Init() та MPI_Finalize().

Функція MPI_Finalize (рис.5.4) завершує паралельну частину програми. Всі подальші звернення до будь-яких функцій MPI, в тому числі до MPI_Init(), заборонені. До моменту виклику MPI_Finalize() даним процесом всі дії, що вимагають його участі в обміні повідомленнями, повинні бути завершені.

За допомогою функції MPI_Comm_size() (рис.5.6) можна визначити загальну кількість паралельних процесів в групі з комунікатором comm. Розмір групи повертається через параметр size. За допомогою функції MPI_Comm_rank() (рис.5.6) можна визначити номер процесу в групі з комунікатором comm. Номер процесу, що повертається за адресою &rank, лежить в діапазоні від 0 до size_of_group-1. На рис.5.7 наведено приклад використання функцій MPI_Init(), MPI_Finalize(), MPI_Comm_size() та MPI_Comm_rank().

```
int MPI_Comm_size(MPI_Comm comm, int* size)

int MPI_Comm_rank(MPI_Comm comm, int* rank)
```

Рис.5.6. Опис функцій MPI_Comm_size() та MPI_Comm_rank().

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char* argv[])
{
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // Print the message
    printf("Hello World! I am %d of %d\n", rank, size);
    // Finalize the MPI environment.
    MPI_Finalize();
    return 0;
}
```

Рис.5.7. Приклад простої програми з використанням MPI.

Визначити статус ініціалізації чи завершення паралельної частини програми можна за допомогою функцій MPI_Initialized() та MPI_Finalized(). Визначити назву обчислювального вузла, на якому виконується процес, можна за допомогою функції MPI_Get_processor_name(). Для негайної зупинки роботи MPI використовується функція MPI_Abort().

5.1.3. Відправка та отримання повідомлень з блокуванням

В MPI реалізовано нерезидентний зв'язок між процесами з обміном повідомленнями в одному з двох основних режимів: синхронному (з блокуванням на викликах функцій відправки та отримання повідомлень) та асинхронному (без блокування на викликах функцій відправки та отримання повідомлень). Для відправки та отримання повідомлень з блокуванням (в синхронному режимі) використовуються функції MPI_Send() та MPI_Recv().

Функція MPI_Send() (рис.5.8) виконує блокуючу відправку повідомлення з ідентифікатором msgtag, що складається з count елементів даних типу datatype, процесу з номером dest у групі з комунікатором comm. Всі елементи повідомлення розташовані неперервно в буфері buf. Значення count може бути нулем. Тип переданих елементів datatype повинен вказуватися за допомогою визначених констант типу (табл.5.1). Дозволяється передавати повідомлення самому собі. Приклад виклику функції MPI_Send() наведено на рис.5.9.

```
int MPI_Send( void*        buf,        // адреса початку буфера повідомлення
              int          count,      // кількість елементів повідомлення
              MPI_Datatype datatype,   // тип даних елементів
              int          dest,       // номер процеса-отримувача
              int          msgtag,     // ідентифікатор (тег) повідомлення
              MPI_Comm     comm )      // ідентифікатор комунікатора (групи)
```

Рис.5.8. Опис функції MPI_Send().

Таблиця 5.1. Відповідність між типами даних, що використовуються для декларування змінної (ANSI C), і тим, як вони кодуються в функціях MPI.

C type	**MPI type**
char	MPI_CHAR
unsigned char	MPI_UNSIGNED_CHAR
char	MPI_SIGNED_CHAR
short	MPI_SHORT
unsigned short	MPI_UNSIGNED_SHORT
int	MPI_INT
unsigned int	MPI_UNSIGNED
long int	MPI_LONG
unsigned long int	MPI_UNSIGNED_LONG
long long int	MPI_LONG_LONG_INT
float	MPI_FLOAT
double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE
unsigned char	MPI_BYTE

Блокування гарантує коректність повторного використання всіх параметрів після повернення з функції MPI_Send(). Вибір способу здійснення цієї гарантії: копіювання в проміжний буфер або безпосередня передача даних процесу dest, залишається за MPI. Слід спеціально зазначити, що повернення з функції MPI_Send() не означає ні того, що

повідомлення вже передано процесу dest, ні того, що повідомлення покинуло обчислювальний вузол, на якому виконується процес, що викликав MPI_Send().

```
long int i;

MPI_Send( &i, 1, MPI_LONG_INT, target, tag, comm);
```

Рис.5.9. Приклад використання функції MPI_Send() для відправки повідомлення з тегом tag процесу з номером target в групі комунікатора comm.

Функція MPI_Recv() (рис.5.10) виконує отримання повідомлення з ідентифікатором msgtag від процесу source з блокуванням. Число елементів в отриманому повідомленні не повинно перевищувати значення count. Якщо число отриманих елементів менше значення count, то гарантується, що в буфері buf зміняться тільки елементи, що відповідають елементам отриманого повідомлення. Якщо потрібно дізнатися точну кількість елементів в повідомленні, то можна скористатися функцією MPI_Probe(). Блокування гарантує, що після повернення з функції MPI_Recv() всі елементи повідомлення вже отримані і розташовані в буфері buf.

```
int MPI_Recv( void*        buf,        // адреса початку буфера повідомлення
              int          count,      // максимальна кількість елементів пов-ня
              MPI_Datatype datatype,   // тип даних елементів
              int          source,     // номер процеса-відправника
              int          msgtag,     // ідентифікатор (тег) повідомлення
              MPI_Comm     comm,       // ідентифікатор комунікатора (групи)
              MPI_Status   *status )   // параметри отриманого повідомлення
```

Рис.5.10. Опис функції MPI_Recv().

В якості номеру процесу-відправника можна вказати константу MPI_ANY_SOURCE - ознаку того, що можна отримати повідомлення від будь-якого процесу. В якості ідентифікатора прийнятого повідомлення можна вказати константу MPI_ANY_TAG - ознаку того, що можна отримати повідомлення з будь-яким ідентифікатором. Якщо процес відправляє два повідомлення іншому процесу і обидва ці повідомлення відповідають одному і тому ж виклику MPI_Recv(), то першим буде прийнято те повідомлення, яке було відправлено раніше.

Перевірити факт надходження повідомлення та визначити його параметри можна за допомогою функції MPI_Probe(), яка також є блокуючою. Для відправки повідомлення з буферизацією використовується фукнція MPI_Bsend(), для відправки повідомлення з синхронізацією - MPI_Ssend(), для відправки повідомлення по готовності - MPI_Rsend(). Відтак в MPI передбачені різні режими та рівні блокування при відправці повідомлень. Потрібно пам’ятати, що використання блокуючих функцій відправки та отримання повідомлень може призвести до виникнення у програмі тупикових ситуацій (deadlock) та відповідних критичних помилок у її роботі (рис.5.11).

5.1.4. Відправка та отримання повідомлень без блокування

Для відправки та отримання повідомлень без блокування (в асинхронному режимі) використовуються функції MPI_Isend() та MPI_Irecv(). Функція MPI_Isend() (рис.5.12) виконує відправку повідомлення, аналогічно MPI_Send(), але повернення з функції відбувається відразу після ініціалізації процесу відправки без очікування обробки всього повідомлення, що знаходиться в буфері buf. Це означає, що не можна повторно використовувати даний буфер для інших цілей без отримання додаткової інформації про завершення даної відправки. Закінчення процесу передачі даних (тобто того моменту, коли

можна знову використати буфер buf без побоювання зіпсувати повідомлення, що відправляється) можна визначити за допомогою параметра request і функцій MPI_Wait() і MPI_Test(). Повідомлення, відправлене будь-якою функцією MPI_Send() і MPI_Isend(), може бути отримано будь-якою функцією MPI_Recv() і MPI_Irecv(). Тобто відправка повідомлення може бути з блокуванням, а отримання без блокування, і навпаки.

```
if(rank == 0) {
        MPI_Recv(…1…)
        MPI_Send(… 1 …)
} else {
        MPI_Recv(…0…)
        MPI_Send(… 0 …)
}
```

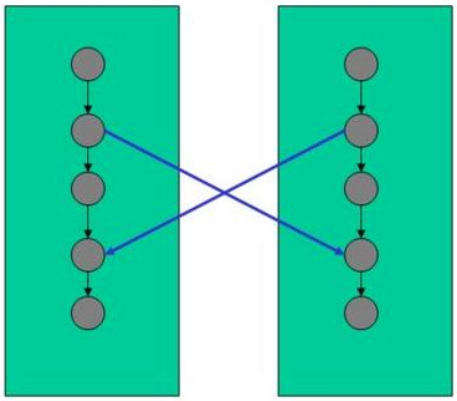

Рис.5.11. Приклад взаємного блокування двох процесів внаслідок помилкової схеми використання функцій MPI_Send() та MPI_Recv() (в параметрах функцій вказано номер отримувача/відправника).

```
int MPI_Isend( void*        buf,      // адреса початку буфера повідомлення
               int          count,    // кількість елементів повідомлення
               MPI_Datatype datatype, // тип даних елементів
               int          dest,     // номер процеса-отримувача
               int          msgtag,   // ідентифікатор (тег) повідомлення
               MPI_Comm     comm,     // ідентифікатор комунікатора (групи)
               MPI_Request  *request ) // ідентифікатор асинхронної відправки
```

Рис.5.12. Опис функції MPI_Isend().

Функція MPI_Irecv() (рис.5.13) виконує отримання повідомлення, аналогічно MPI_Recv(), але повернення з функції відбувається відразу після ініціалізації процесу приймання даних без очікування кінця отримання повідомлення в буфер buf. Закінчення процесу отримання можна визначити за допомогою параметра request і функцій MPI_Wait() та MPI_Test().

```
int MPI_Irecv( void*        buf,     // адреса початку буфера повідомлення
               int          count,   // максимальна кількість елементів пов-ня
               MPI_Datatype datatype,// тип даних елементів
               int          source,  // номер процеса-відправника
               int          msgtag,  // ідентифікатор (тег) повідомлення
               MPI_Comm     comm,    // ідентифікатор комунікатора (групи)
               MPI_Request  *request)// ідентифікатор асинхронного отримання
```

Рис.5.13. Опис функції MPI_Irecv().

Функція MPI_Wait() (рис.5.14) виконує блокуюче очікування завершення асинхронних функцій MPI_Isend() або MPI_Irecv(), асоційованих з ідентифікатором request. Після завершення очікування отримання повідомлення, його атрибути і довжина повертаються у параметрі status. За допомогою функції MPI_Waitall() можна дочекатись декількох ініційованих відправлень/отримань повідомлень.

```
int MPI_Wait( MPI_Request  *request // ідентифікатор асинхронної відправки
                                    // чи асинхронного отримання
              MPI_Status   *status )// параметри отриманого повідомлення
```

Рис.5.14. Опис функції MPI_Wait().

За допомогою функції MPI_Test() можна виконати неблокуючу перевірку завершення асинхронної відправки чи асинхронного отримання повідомлення. За допомогою функції MPI_Testall() можна одночасно перевірити статус завершення декількох ініційованих відправлень/отримань повідомлень.

5.2. Комунікатори та колективна взаємодія процесів в MPI

5.2.1. Використання комунікаторів для роботи з групами процесів

Комунікатор в MPI - це абстрактне поняття, яке об’єднує окрему групу процесів і відповідний контекст виконання операцій обміну повідомленнями і колективної взаємодії між процесами. Комунікатор можна розглядати як окрему область взаємодії групи процесів або як агента-посередника, який забезпечує та контролює взаємодію процесів у групі. За замовчуванням в MPI існують наступні комунікатори:

1) MPI_COMM_WORLD асоційований з групою, в яку входять усі процеси;

2) MPI_COMM_SELF асоційований з групою, в яку входять лише один даний процес;

3) MPI_COMM_NULL асоційований з групою, в яку не входить жодний процес.

Ідентифікатор комунікатора (comm) вказується серед параметрів усіх функцій обміну повідомленнями між процесами та функцій колективної взаємодії процесів для однозначного визначення контексту виконання цих функцій та унікальної ідентифікації процесів у вигляді пари ідентифікаторів (comm, rank), де rank - це номер процеса у групі з комунікатором comm. Окрім “звичайних” комунікаторів (intra-communicators), в стандарті MPI також визначені інтер-комунікатори (inter-communicators), які забезпечують зв’язок між групами процесів.

Для створення нового комунікатора потрібно спочатку створити нову групу процесів, наприклад, за допомогою функції MPI_Group_incl() - додати вказані процеси (ranks[]) до групи group або функції MPI_Group_excl() - виключити вказані процеси (ranks[]) з групи group (рис.5.15). Після цього для створеної групи призначається новий комунікатор за допомогою функції MPI_Comm_create() (рис.5.15). Ще один спосіб - скористатись функцією MPI_Comm_split().

```
int MPI_Group_incl(MPI_Group group,
                   int rank_count,
                   const int ranks[],
                   MPI_Group* new_group);

int MPI_Group_excl(MPI_Group group,
                   int rank_count,
                   const int ranks[],
                   MPI_Group* new_group)

int MPI_Comm_create(MPI_Comm old_communicator,
                    MPI_Group group,
                    MPI_Comm* new_communicator);
```

Рис.5.15. Опис функцій MPI_Group_incl(), MPI_Group_excl(), MPI_Comm_create().

Функція MPI_Comm_split() (рис.5.16) розбиває множину процесів, що входять в групу з комунікатором comm, на окремі підгрупи - по одній підгрупі на кожне значення параметра color (невід'ємне ціле число). Кожна нова підгрупа містить всі процеси одного “кольору”, який задається значенням параметру color (рис.5.17, рис.5.18). Ідентифікатор нового комунікатора повертається в параметрі newcomm. Якщо в якості color вказано значення MPI_UNDEFINED, то в newcomm буде повернуто значення MPI_COMM_NULL. Параметр key задає порядковий номер, під яким викликаючий процес буде входити в нову підгрупу. Якщо всі процеси вказують у key свій поточний порядковий номер в групі з комунікатором comm, то порядок їх входження зберігається в нових підгрупах. На рис.5.19 наведено приклад використання функції MPI_Comm_split() для створення двох підгруп процесів A і B по два процеси в кожній з групи чотирьох процесів.

```
int MPI_Comm_split(
                MPI_Comm comm, // ідентифікатор комунікатора
                int color, // ознака поділу на групи (“колір”)
                int key, // параметр, що визначає нумерацію в нових групах
                MPI_Comm *newcomm ) // ідентифікатор нового комунікатора
```

Рис.5.16. Опис функції MPI_Comm_split().

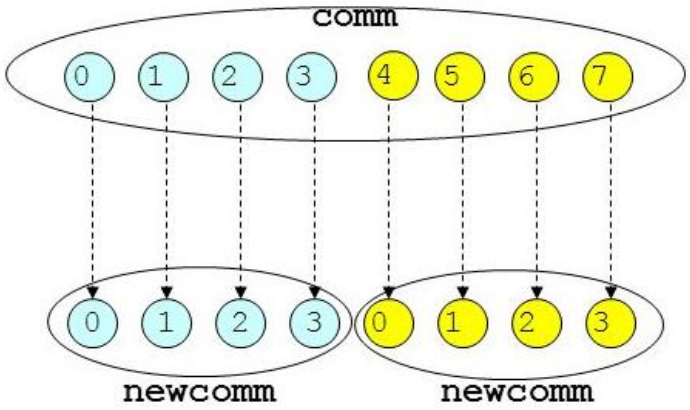

Рис.5.17. Приклад розбиття групи з комунікатором comm
на дві підгрупи з відповідними комунікаторами.

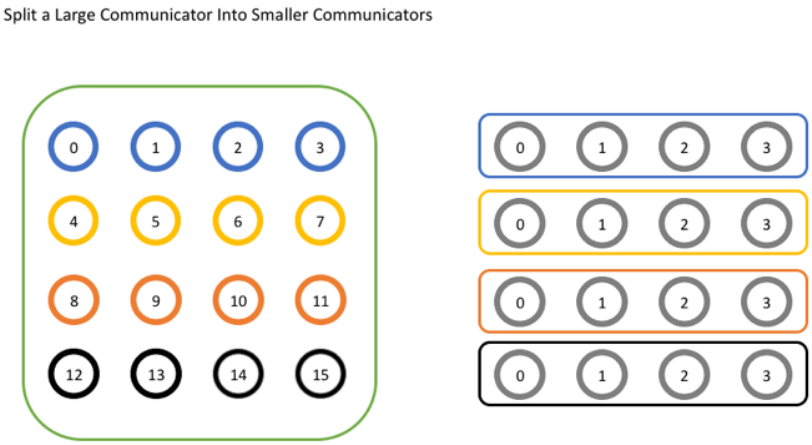

Рис.5.18. Приклад розбиття групи з комунікатором comm
на чотири підгрупи з відповідними комунікаторами.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/**
 * MPI processes split into two groups depending on whether their rank
 * is even.
 * +----------------+---+---+---+---+
 * | MPI processes  | 0 | 1 | 2 | 3 |
 * +----------------+---+---+---+---+
 * | MPI_COMM_WORLD | X | X | X | X |
 * | Subgroup A     | X |   | X |   |
 * | Subgroup B     |   | X |   | X |
 * +----------------+---+---+---+---+
 *
 * In subcommunicator A, MPI processes are assigned ranks as
 * in the same order their rank in the global communicator.
```

```
     * In subcommunicator B, MPI processes are assigned ranks
     * in the opposite order as their rank in the global communicator.
     **/

    int main(int argc, char* argv[])
    {
        MPI_Init(&argc, &argv);

        // Get my rank in the global communicator
        int my_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        // Determine the colour and key based on whether my rank is even.
        char subcommunicator;
        int colour;
        int key;
        if(my_rank % 2 == 0)
        {
            subcommunicator = 'A';
            colour = 0;
            key = my_rank;
        }
        else
        {
            subcommunicator = 'B';
            colour = 1;
            key = comm_size - my_rank;
        }

        // Split the global communicator
        MPI_Comm new_comm;
        MPI_Comm_split(MPI_COMM_WORLD, colour, key, &new_comm);

        // Get my rank in the new communicator
        int my_new_comm_rank;
        MPI_Comm_rank(new_comm, &my_new_comm_rank);

        // Print my new rank and new communicator
        printf("[MPI process %d] I am now MPI process %d in subcommunicator %c.
                \n", my_rank, my_new_comm_rank, subcommunicator);

        MPI_Finalize();

        return EXIT_SUCCESS;
    }
```

Рис.5.19. Приклад використання функції MPI_Comm_split(): група з комунікатором MPI_COMM_WORLD розбивається на дві підгрупи.

Функція MPI_Comm_free() (рис.5.20) знищує групу процесів, асоційовану з комунікатором comm, який після повернення з функції встановлюється в MPI_COMM_NULL.

```
int MPI_Comm_free(MPI_Comm comm)
```

Рис.5.20. Опис функції MPI_Comm_split().

5.2.2. Колективна взаємодія процесів в MPI

В операціях колективної взаємодії процесів беруть участь усі процеси у групі з комунікатором comm. Відповідна функція повинна бути викликана кожним процесом, можливо, зі своїм набором параметрів. Повернення з функції колективної взаємодії може відбутися в той момент, коли участь процесу в даній операції вже завершено. Як і для інших блокуючих функцій, повернення з функції означає лише можливість повторного використання буферу відправки чи отримання даних, і не означає ні того, що операція завершена іншими процесами, ні того, що вони розпочали її виконання (якщо це можливо за змістом операції колективної взаємодії).

Функції колективної взаємодії можна поділити на три основні групи:

1) Функції обміну даними між процесами у групі: MPI_Bcast(), MPI_Gather(), MPI_Scatter(), MPI_Allgather(), MPI_Alltoall() та їх неблокуючі версії;

2) Виконання глобальних операцій над даними, що надаються процесами з групи: MPI_Allreduce(), MPI_Reduce(), MPI_Reduce_scatter_block(), MPI_Ireduce_scatter(), MPI_Scan() та їх неблокуючі версії;

3) Бар'єрна синхронізація процесів у групі: MPI_Barrier(), MPI_Ibarrier().

Функція MPI_Bcast() (рис.5.21) виконує розсилку повідомлення від процесу source усім процесам в групі, включаючи сам процес source (рис.5.22). При поверненні з функції MPI_Bcast() вміст буфера buf процесу source буде скопійовано в локальний буфер кожного процесу в групі. Значення параметрів count, datatype і source повинні бути однаковими у всіх процесів, що викликають фукнцію MPI_Bcast().

```
int MPI_Bcast(
                void *buf, // адреса початку буфера з повідомленням
                int count, // кількість елементів повідомлення
                MPI_Datatype datatype, // тип даних елементів
                int source, // номер процеса-відправника
                MPI_Comm comm ) // ідентифікатор комунікатора
```

Рис.5.21. Опис функції MPI_Bcast().

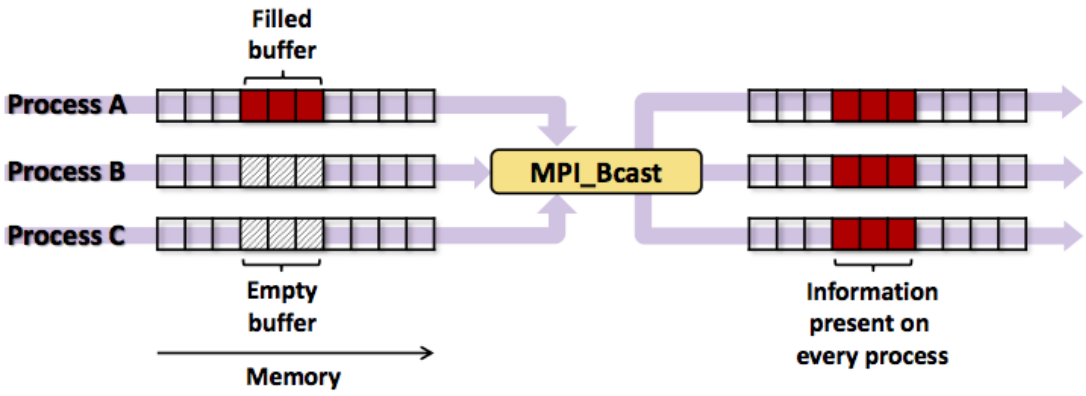

Рис.5.22. Схема виконання функції MPI_Bcast() для трьох процесів.

Функція MPI_Gather() (рис.5.23) виконує збирання даних з усіх процесів у групі в буфер rbuf процесу dest. Кожен процес, включаючи dest, відправляє вміст свого буфера sbuf процесу dest (рис.5.24). Збираючий процес зберігає дані в буфері rbuf, розташовуючи їх у порядку зростання номерів процесів-відправників даних. Параметр rbuf має значення тільки у збираючому процесі, а в інших процесах ігнорується. Значення параметрів count, datatype і dest повинні бути однаковими у всіх процесах, що викликають функцію MPI_Gather(). Зворотна операція - роздача даних від одного процесу усім іншим процесам в групі виконується за допомогою функції MPI_Scatter() (рис.5.25).

```
int MPI_Gather(
                void *sbuf, // адреса початку буфера з даними, що відправляються
                int scount, // кількість елементів даних, що відправляються
                MPI_Datatype stype, // тип елементів даних, що відправляються
                void *rbuf, // адреса початку буфера зібраних даних
                int rcount, // кількість елементів зібраних даних
                MPI_Datatype rtype, // тип елементів зібраних даних
                int dest, // номер процеса, в якому збираються дані
                MPI_Comm comm ) // ідентифікатор комунікатора
```

Рис.5.23. Опис функції MPI_Gather().

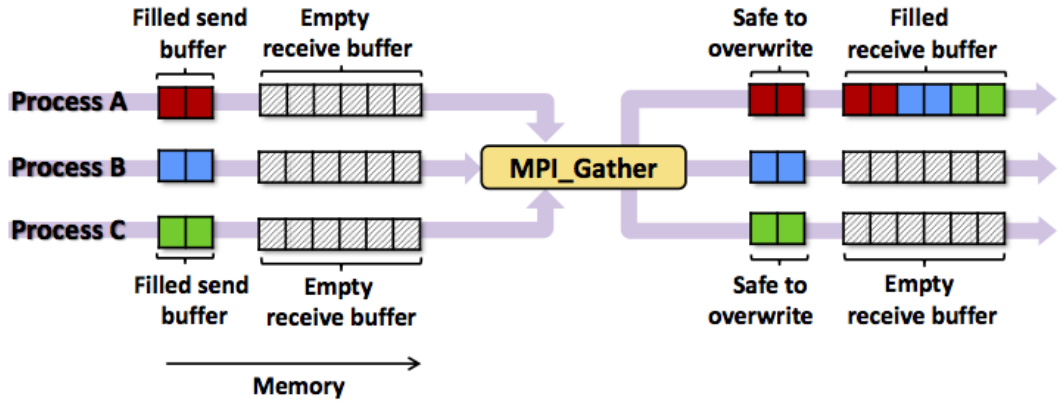

Рис.5.24. Схема виконання функції MPI_Gather() для трьох процесів.

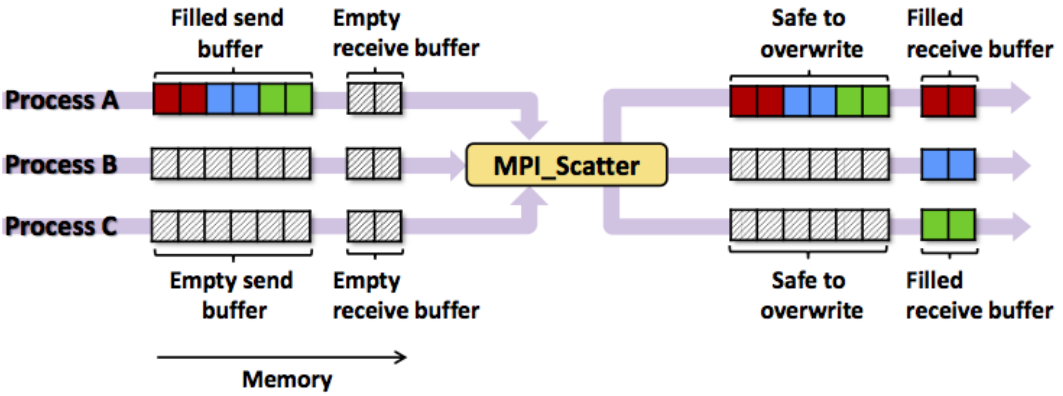

Рис.5.25. Схема виконання функції MPI_Scatter() для трьох процесів.

Функція MPI_Allreduce() (рис.5.26) виконує count глобальних операцій *op* з поверненням count результатів у всіх процесах в буфер rbuf (рис.5.27). Операція *op* виконується незалежно над відповідними аргументами всіх процесів. Значення параметрів count та datatype у всіх процесах повинні бути однаковими. З міркувань ефективності реалізації передбачається, що операція *op* має властивості асоціативності і комутативності. В табл.5.2 наведені типи операцій *op*, що визначені у стандарті MPI.

```
int MPI_Allreduce(
                void *sbuf, // адреса початку буфера з аргументами
                void *rbuf, // адреса початку буфера з результатом
                int count, // кількість аргументів в кожному процесі
                MPI_Datatype datatype, // тип аргументів
                MPI_Op op, // ідентифікатор глобальної операції
                MPI_Comm comm ) // ідентифікатор комунікатора
```

Рис.5.26. Опис функції MPI_Allreduce().

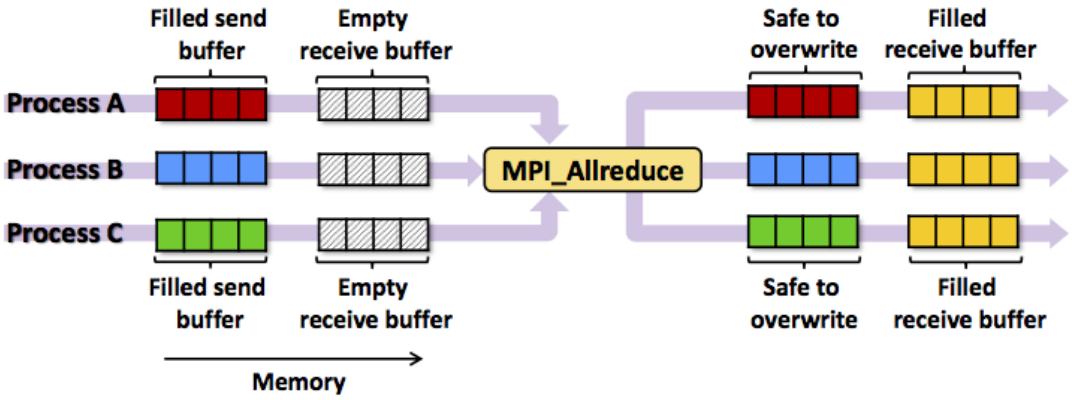

Рис.5.27. Схема виконання функції MPI_Allreduce() для трьох процесів.

Таблиця 5.2. Типи операцій колективної взаємодії, визначені у стандарті MPI.

MPI Reduction Operation		C Data Types	Fortran Data Type
MPI_MAX	maximum	integer, float	integer, real, complex
MPI_MIN	minimum	integer, float	integer, real, complex
MPI_SUM	sum	integer, float	integer, real, complex
MPI_PROD	product	integer, float	integer, real, complex
MPI_LAND	logical AND	integer	logical
MPI_BAND	bit-wise AND	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	logical OR	integer	logical
MPI_BOR	bit-wise OR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer	logical
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double	real, complex,double precision
MPI_MINLOC	min value and location	float, double and long double	real, complex, double precision

Функція MPI_Reduce() (рис.5.28) аналогічна функції MPI_Allreduce(), але результат буде записаний в буфер rbuf тільки у процесі з номером root (рис.5.29).

```
int MPI_Reduce(
                void *sbuf, // адреса початку буфера з аргументами
                void *rbuf, // адреса початку буфера з результатом
                int count, // кількість аргументів в кожному процесі
                MPI_Datatype datatype, // тип аргументів
                MPI_Op op, // ідентифікатор глобальної операції
                int root, // номер процеса-отримувача результату
                MPI_Comm comm ) // ідентифікатор комунікатора
```

Рис.5.28. Опис функції MPI_Reduce().

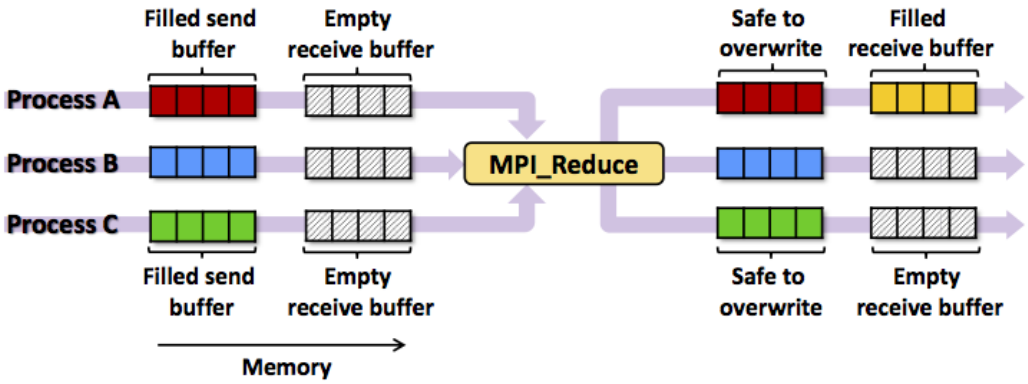

Рис.5.29. Схема виконання функції MPI_Reduce() для трьох процесів.

5.2.3. Бар'єрна синхронізація процесів в MPI

Бар'єрна синхронізація процесів в MPI виконується за допомогою функції MPI_Barrier() (рис.5.30). Ця функція блокує роботу викликаючого процесу до тих пір, поки всі інші процеси у групі з комунікатором comm також не викличуть цю функцію (рис.5.31, рис.5.32). На рис.5.33 наведено приклад використання функції MPI_Barrier().

```
int MPI_Barrier(MPI_Comm comm)

int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

Рис.5.30. Опис функцій MPI_Barrier() та MPI_Ibarrier().

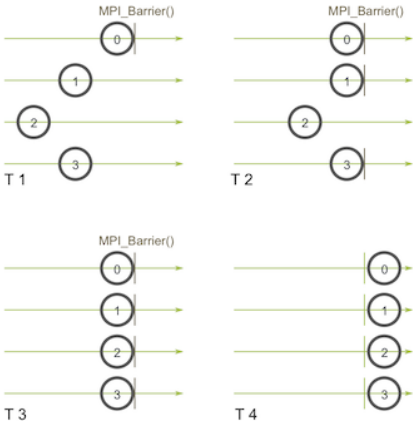

Рис.5.31. Схема бар'єрної синхронізації в MPI.

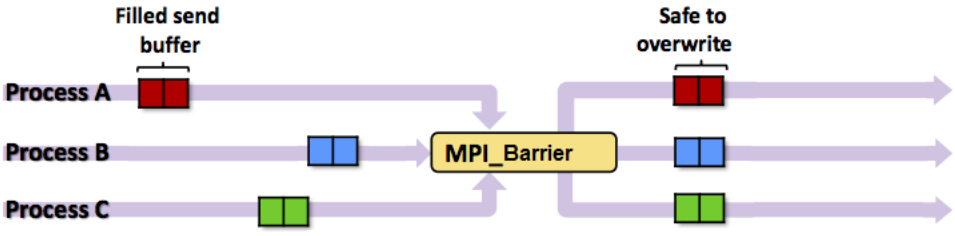

Рис.5.32. Схема виконання функції MPI_Barrier() для трьох процесів.

Функція MPI_Ibarrier() (рис.5.30) є неблокуючою версією функції MPI_Barrier(). Викликом функції MPI_Ibarrier() процес нотифікує інші процеси у групі з комунікатором comm про те, що він досягнув бар'єру, після чого продовжує своє виконання. В даному випадку перевірити, чи усі процеси у групі з комунікатором comm досягнули бар’єру можна за допомогою функції MPI_Wait() або MPI_Test(), вказавши параметром ідентифікатор асинхронної нотифікації request, якій повертає функція MPI_Ibarrier(). Функцію MPI_Ibarrier() можна використати для того, щоб зменшити час “простою” процеса на бар’єрі, заповнивши цей час обчисленнями до моменту виклику MPI_Wait() або MPI_Test().

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("[MPI process %d] I start waiting on the barrier.\n", my_rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("[MPI process %d] I know all MPI processes have waited
            on the barrier.\n", my_rank);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Рис.5.33. Приклад використання функції MPI_Barrier().

Контрольні питання

1. В чому полягає специфіка організації паралельних обчислень в розподілених системах?
2. Який базовий спосіб організації паралельних обчислень використовується в розподілених системах?
3. Для чого призначений програмний інтерфейс MPI?
4. На якому рівні розпаралелюється програма в разі використання програмного інтерфейсу MPI?
5. Як організовано виконання паралельних компонент програми з використанням програмного інтерфейсу MPI?
6. Які реалізації стандарту MPI у вигляді програмних бібліотек є найбільш популярними?
7. Які основні групи функцій входять до програмного інтерфейсу MPI?
8. Якими функціями задаються початок та кінець паралельної частини MPI-програми?
9. За допомогою якої функції в MPI можна визначити загальну кількість паралельних процесів в групі із заданим комунікатором?
10. За допомогою якої функції в MPI можна визначити номер процесу в групі із заданим комунікатором?
11. Яка функція використовується для негайної зупинки роботи MPI?
12. Які функції MPI використовуються для відправки та отримання повідомлень з блокуванням?
13. Які функції MPI використовуються для відправки та отримання повідомлень без блокування?
14. Як організовано прив’язку точки відправки повідомлення до точки його отримання в MPI?
15. За допомогою якої функції в MPI можна виконати блокуюче очікування завершення асинхронної відправки чи асинхронного отримання повідомлення?
16. За допомогою якої функції в MPI можна виконати неблокуючу перевірку завершення асинхронної відправки чи асинхронного отримання повідомлення?
17. Як в MPI організовано створення нових підгруп процесів і відповідних комунікаторів?
18. Яка функція в MPI виконує розсилку повідомлення від процесу-відправника усім процесам в групі, включаючи сам цей процес?
19. Яка функція в MPI виконує збирання даних з усіх процесів у групі в буфер даних процесу-отримувача?
20. Яка функція в MPI виконує задану глобальну операцію над даними, що надаються процесами з групи, з поверненням результату в буфер даних всіх процесів?
21. Яка функція в MPI виконує задану глобальну операцію над даними, що надаються процесами з групи, з поверненням результату в буфер даних тільки процесу з заданим номером?

Глосарій

Паралельна комп'ютерна система – комп'ютерна система, яка складається з деякої кількості обчислювальних вузлів (ядер процесора, процесорів, комп'ютерів, тощо), об'єднаних каналами передачі даних з високою пропускною здатністю та/або використовуючих спільну апаратну пам'ять. Структура паралельної комп'ютерної системи дозволяє виконувати обчислення з явним (апаратним) розпаралелюванням.

Розподілена система (distributed system) – комп'ютерна система, в якій розв'язуються складні обчислювальні завдання з використанням двох і більше комп'ютерів, об'єднаних в мережу. Розподілену обчислювальну систему можна розглядати як слабо-зв'язний набір окремих обчислювачів (вузлів), об'єднаних в рамках єдиної комп'ютерної мережі. В ході роботи розподіленої обчислювальної системи забезпечується доступ процесів даного вузла до віддалених ресурсів інших вузлів та надання локальних ресурсів для використання процесами віддалених вузлів.

Функціональна декомпозиція – підхід до створення паралельного алгоритму, коли його загальна функція розбивається на окремі під-функції, які можуть виконуватись паралельно.

Декомпозиція даних – підхід до створення паралельного алгоритму, коли його вхідні дані розбиваються на окремі частини, які можуть оброблятись паралельно.

SMP (Symmetric MultiProcessing) – симетрична мультипроцесорність – архітектура багатопроцесорних комп'ютерів, в якій два або більше однакових процесорів підключаються до загальної пам'яті. Більшість багатопроцесорних систем сьогодні використовують архітектуру SMP.

Хмарні обчислення (cloud computing) – модель забезпечення повсюдного та зручного доступу на вимогу через мережу до спільного пулу обчислювальних ресурсів, що підлягають налаштуванню (наприклад, до комунікаційних мереж, серверів, засобів збереження даних, прикладних програм та сервісів), і які можуть бути оперативно надані та звільнені з мінімальними управлінськими затратами та зверненнями до провайдера.

UNIX – сімейство багатозадачних багатокористувацьких ОС, які походять від оригінальної ОС AT&T Unix, розробленої у 1970-х роках в дослідницькому центрі Bell Labs Кеном Томпсоном (Ken Thompson), Деннісом Річі (Dennis Ritchie) та іншими. Операційні системи сімейства Unix характеризуються модульним дизайном, в якому кожна задача виконується окремою утилітою, взаємодія здійснюється через єдину файлову систему, а для роботи з утилітами використовується командна оболонка. Ідеї, закладені в основу Unix, мали величезний вплив на розвиток операційних систем. На даний час Unix-системи визнані одними з найбільш історично важливих ОС.

Linux – UNIX-подібна ОС з відкритим програмним кодом на основі ядра Linux (Linux kernel), яке має монолітну архітектуру. Це один із найвидатніших прикладів розробки вільного (free) та відкритого (з відкритим кодом, open source) програмного забезпечення. Значна кількість спеціалізованих дистрибутивів Linux, які розробляють та підтримують різні спільноти, надає широкі можливості вибору програмного забезпечення.

Багатозадачність (multitasking) – властивість операційної системи або середовища програмування забезпечувати можливість паралельного (або псевдопаралельного) виконання декількох обчислювальних процесів.

Time-sharing – розподіл одного обчислювального ресурсу (комп'ютера, процесора, ядра процесора, тощо) між декількома користувачами та/або обчислювальними процесами, які використовують його одночасно в режимі багатозадачності на основі розподілу між ними часу доступу до ресурсу.

Мобільні обчислення (mobile computing) – це взаємодія людини з комп'ютером, що переміщується у просторі під час його використання, з можливістю передавання даних, аудіо та відео. Мобільні обчислення включають мобільний зв'язок, мобільні обчислювальні пристрої та мобільне програмне забезпечення.

Processor affinity – «прив'язка» обчислювального процесу до певного процесора або підмножини процесорів в мульти-процесорних системах, коли згідно вказівки користувача (програміста) процес виконується лише на певному процесорі або певній підмножині процесорів.

Взаємодія обчислювальних процесів (Inter-Process Communication, IPC) – набір механізмів, які надає операційна система для забезпечення обміну даними між процесами, роботи процесів зі спільними даними та синхронізації процесів. Потреба у механізмах IPC обумовлена ізоляцією процесів один від одного в сучасних ОС задля надійності та безпеки обчислень на системному рівні (в тому числі з використанням механізму віртуальної пам'яті).

М'ютекс (lock, mutex) – (від англ. mutual exclusion – взаємне виключення) об'єкт синхронізації паралельних задач (процесів, потоків) для організації взаємного виключення паралельних задач (процесів, потоків) при здійсненні операцій доступу до даних (запис, читання).

Критична секція програмного коду (critical section) – об'єкт синхронізації паралельних задач (процесів, потоків), що дозволяє запобігти одночасному виконанню деякого критичного набору операцій (зазвичай пов'язаних з доступом до даних) кількома паралельними задачами.

Взаємне блокування (deadlock) – ситуація, коли кожен із групи процесів очікує на подію, яку може викликати лише інший процес з цієї ж групи.

Ресурсний голод (resource starvation) – проблема в інформатиці, що унеможливлює виконання процесом задачі, спричинена постійною відмовою в необхідних ресурсах. Причиною відмови в ресурсах може бути: помилка в алгоритмі розподілу ресурсів, витік ресурсів, DoS-атака.

Message passing (обмін повідомленнями) – спосіб взаємодії обчислювальних процесів, в рамках якого вони обмінюються даними шляхом відправки та прийому повідомлень.

Обчислювальний процес – примірник деякої програми під час її виконання. Окремий обчислювальний процес складається з послідовності машинних інструкцій, які виконує CPU.

Process control block (PCB) – блок управління процесом, дескриптор процесу - це структура даних, що використовується операційною системою для зберігання всієї необхідної інформації про процес.

CPU bound process – обчислювальний процес, який більше часу витрачає на обчислення, а операції вводу/виводу виконує рідко.

I/O bound process – обчислювальний процес, який більше часу витрачає на очікування та виконання операцій вводу/виводу ніж на обчислення.

CPU burst – час, який було витрачено на виконання обчислювального процесу у процесорі, до моменту “захоплення” процесору іншим процесом.

I/O burst – час, який було витрачено обчислювальним процесом на очікування операції вводу/виводу, до моменту її здійснення і повернення процесу з черги очікування (wait queue) до черги готовності до виконання (ready queue).

Контекст процесу (process context) – стан регістрів процесора та стан операційного середовища поточного процесу. Зокрема в ОС UNIX розрізняють: користувацький контекст, регістровий контекст та контекст системного рівня.

Переключення контексту (context switch) – процес збереження стану процесу або потоку, щоб забезпечити можливість відновити його виконання пізніше, та завантаження стану іншого процесу або потоку для продовження його виконання. Це дозволяє кільком процесам спільно використовувати один центральний процесор (CPU) в режимі з розподілом у часі (time-sharing) і є важливою особливістю багатозадачної операційної системи.

Програмний потік (thread) – одиниця розпаралелювання обчислень в межах одного обчислювального процесу. В окремого програмного потоку є лише власний регістровий контекст. Сегменти коду та даних обчислювального процесу та його ресурси спільні для всіх потоків в межах цього процесу.

User level threads – програмні потоки, які виконуються на рівні користувача без участі ядра ОС.

Kernel level threads – програмні потоки, які підтримуються та управляються безпосередньо ядром ОС.

Пул потоків (thread pool) – обмежений набір заздалегідь створених (можливо однакових) потоків, яким передаються нові завдання у міру їх надходження з метою зменшення витрат на породження нових потоків під час роботи програми.

System Call Interface (SCI) – інтерфейс системних викликів, це різновид програмного інтерфейсу за допомогою якого прикладна програма звертається до сервісів, які надає ядро операційної системи.

Системний виклик (system call) – запит на виконання окремої функції в межах інтерфейсу системних викликів, за допомогою якого прикладна програма звертається до сервісів, які надає ядро операційної системи.

Планування паралельного виконання процесів – вирішення задачі організації розподілу процесорного часу між обчислювальними процесами з точки зору забезпечення багатозадачності операційного середовища. Планування паралельного виконання процесів передбачає 1) визначення моменту часу заміни виконуваного процесу іншим; 2) вибір процесу на виконання з черги готових процесів.

Синхронізація обчислювальних процесів – координація спільного паралельного виконання обчислювальних процесів за метою 1) запобігання помилкового виконання (порушення логіки роботи паралельної програми); 2) забезпечення цілісності (consistency) спільних даних (впорядкування операцій читання/запису в часі).

Адаптивний паралелізм (adaptive parallelism) – механізм само-адаптивного програмного забезпечення, в рамках якого паралельній програмі делегують повноваження по прийняттю

рішення: 1) яку кількість паралельних процесів (потоків) створити в даний момент (з врахуванням різних «зовнішніх» обставин), 2) де, коли і в якому режимі запустити ці процеси (потоки) на виконання.

Спулінг (spooling) – механізм організації одночасного доступу декількох паралельних процесів до одного пристрою вводу/виводу за допомогою черги (каталогу спулінгу) звертань процесів до пристрою та деякого алгоритму розташування звертань у цій черзі.

Локальність звертань до пам'яті (locality of reference) – явище в сучасних обчислювальних машинах, яке полягає в тому, що набори даних, до яких відбувається звертання в порівняно короткий проміжок часу, розміщуються в пам'яті достатньо добре прогнозованими кластерами.

Паралельне програмування – реалізація паралельних алгоритмів з використанням можливостей паралельних комп'ютерних систем, в тому числі забезпечення паралельного виконання системних та користувацьких обчислювальних процесів.

Технологія паралельного програмування – набір програмних засобів та інструментів для реалізації паралельних програм.

Неіменований канал (anonymous pipe) – механізм IPC призначений для обміну даними між процесами за принципом FIFO. Неіменований канал може бути використаний для обміну даними лише між процесами, які пов'язані один з одним в рамках відношення «батько-син».

Іменований канал (named pipe, FIFO) – механізм IPC призначений для обміну даними між процесами за принципом FIFO. Іменований канал відрізняється від неіменованого тим, що має ім'я. Внаслідок цього іменований канал може бути використаний для обміну даними між процесами, які не пов'язані один з одним в рамках відношення «батько-син». Іменування каналу забезпечується тим, що він відображається у файловій системі у вигляді спеціального файлу.

Черга повідомлень (message queue) – механізм IPC, який дозволяє двом або більше процесам обмінюватись повідомленнями, кожному з яких призначається деяке ціле число ("тип", "ідентифікатор", "пріоритет" повідомлення).

Спільна пам'ять (shared memory) – механізм IPC, який дозволяє двом або більше процесам мати спільні області віртуальної пам'яті і, як наслідок, обмінюватись даними, що містяться в них.

Семафор – засіб синхронізації паралельних процесів або програмних потоків, в основі якого лежить лічильник, над яким можна виконувати дві атомарні операції: збільшення і зменшення значення лічильнику на одиницю. При цьому операція зменшення над лічильником з нульовим значенням є блокуючою. В ОС Linux значення семафора - це ціле число в діапазоні від 0 до 32767. Оскільки в багатьох програмах потрібно більше одного семафора, ОС Linux надає доступ до цього механізму синхронізації (об'єкту System V IPC) у вигляді множини семафорів (semaphore set).

Сигнал (signal) в ОС Linux – повідомлення обчислювального процесу про те, що сталася деяка подія.

Сокети Берклі (Berkeley sockets) – механізм взаємодії процесів (IPC), для обміну даними між процесами, що виконуються на різних обчислювальних вузлах комп'ютерної мережі.

POSIX Threads (Pthreads) – стандарт POSIX, який визначає програмний інтерфейс (API) для створення та використання програмних потоків.

Потоко-безпечна функція (thread-safe function) – функція, яку безпечно викликати у декількох потоках одночасно (тобто вона буде давати однакові результати незалежно від того, в якому потоці вона була викликана).

Умовна змінна (condition variable) – об'єкт синхронізації, за допомогою якого один потік сигналізує іншому про подію, що відбулася. Цього сигналу може очікувати один або декілька потоків (в заблокованому стані) від деякого іншого потоку. Тобто умовна змінна дозволяє одному потоку інформувати інші потоки про зміну стану деякої спільної змінної (або іншого спільного ресурсу), а також дозволяє іншим потокам очікувати (з блокуванням) на отримання такого повідомлення. Основне використання умовних змінних - це узгодження деякої послідовності операцій, що виконуються різними потоками.

Блокування читання-запису (readers–writer lock) – механізм синхронізації, який дозволяє одночасне читання та ексклюзивний запис захищених спільних даних. Блокування читання-запису реалізується у вигляді відповідного об'єкту синхронізації rwlock, який можна захопити або в режимі читання (read lock), або в режимі запису (write lock). Щоб змінити спільні дані, потік повинен спочатку отримати ексклюзивний доступ для запису (захопити rwlock в режимі запису). Ексклюзивне захоплення для запису не дозволяється, поки не будуть звільнені всі захоплення даного rwlock в режимі читання.

Спінлок (spinlock) – механізм синхронізації, який використовується в тих самих ситуаціях, де потрібен м'ютекс, але блокування (призупинка) потоку на виклику операції lock неприпустиме. Спінлок - це механізм синхронізації низького рівня, який, головним чином, використовується у мультипроцесорних системах зі спільною пам'яттю (shared memory multiprocessors). Коли викликаючий потік намагається захопити спінлок, який вже захоплений іншим потоком, він не призупиняється, а виконує спеціальний цикл, в якому весь час перевіряє, чи спінлок не звільнився.

Бар'єрна синхронізація – механізм синхронізації, який використовується у випадках, коли потрібно дочекатися завершення виконання обчислень у декількох програмних потоках, перш ніж продовжити виконання програми одночасно у всіх цих потоках.

OpenMP (Open Multi-Processing) – прикладний програмний інтерфейс паралельного програмування на рівні потоків (базові мови програмування: C, C++, Fortran), який призначений для спрощення створення паралельних програм для апаратних платформ типу "мультипроцесорна система зі спільною пам'яттю" (shared memory multiprocessing).

oneTBB (oneAPI Threading Building Blocks, попередня назва - Intel TBB) – бібліотека, яка підтримує організацію масштабованих паралельних обчислень з використанням мови програмування C++. Для роботи з oneTBB не потрібні спеціальні надбудови чи модифікації мови C++ або спеціальні компілятори. oneTBB призначений для підтримки та спрощення паралельного програмування з гнучким масштабуванням паралельної обробки великих наборів даних.

Parallel Virtual Machine (PVM, "Паралельна віртуальна машина") – програмне середовищем для розподілених паралельних обчислень з підсистемою передачі повідомлень, розроблене в 1989 році. PVM була одною з систем, яка мотивувала необхідність створення стандарту передавання повідомлень для організації паралельних обчислень в розподілених системах, зокрема стандарту MPI.

Message Passing Interface (MPI, “Інтерфейс передачі повідомлень”) – стандартизований програмний інтерфейс передачі повідомлень, призначений для організації паралельних обчислень в розподілених комп’ютерних системах. Стандарт MPI визначає синтаксис та семантику бібліотечних функцій для використання у паралельних програмах для розподілених системи (в тому числі кластерів) на мовах програмування C, C++ та Fortran. З точки зору класифікації типів зв’язку у розподілених комп’ютерних системах, стандартом MPI задається синхронний та асинхронний нерезидентний зв’язок.

Література

1. Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear, The Art of Multiprocessor Programming, 2nd Edition, Morgan Kaufmann, 2020. – 576 p.
2. Peter Pacheco, Matthew Malensek, An Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann, 2021. – 450 p.
3. Bertil Schmidt Jorge Gonzalez-Dominguez Christian Hundt Moritz Schlarb, Parallel Programming: Concepts and Practice, Morgan Kaufmann, 2017. – 416 p.
4. Programming Models for Parallel Computing, ed. by Pavan Balaji, The MIT Press, 2015. - 488 p.
5. Michael McCool, James Reinders, Arch Robison, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012. – 432 p.
6. Thomas Rauber, Gudula Rünger, Parallel Programming For Multicore and Cluster Systems, Springer, 2010. – 455 p.
7. Fayez Gebali, Algorithms and Parallel Computing, John Wiley & Sons, 2011. – 365 p.
8. Victor Alessandrini, Shared memory application programming, Morgan Kaufmann, 2016. – 528 p.
9. W. Richard Stevens, UNIX Network Programming, Volume 2: Interprocess Communications, 2nd ed., Prentice Hall, 1998. – 558 p.
10. W. Richard Stevens, UNIX Network Programming (Volume 1): Networking APIs: Sockets and XTI, Prentice Hall, 1998. – 1009 p.
11. Michael Kerrisk, The Linux Programming Interface: A Linux and UNIX System Programming Handbook, 1st Edition, No Starch Press, 2010. – 1553 p.
12. W. Stevens, Stephen Rago, Advanced Programming in the UNIX Environment, 3rd Edition, Addison-Wesley Professional, 2013. – 1032 p.
13. Robert Love, Linux System Programming, 2nd ed., O'Reilly Media, 2013. – 456 p.
14. Kaiwan N. Billimoria, Hands-On System Programming with Linux, Packt Publishing, 2018. - 794 p.
15. Ulrich Drepper, Ingo Molnar, The Native POSIX Thread Library for Linux, White paper, Red Hat, Inc., 2005.
16. Rohit Chandra et al. Parallel Programming in OpenMP, Academic Press, Morgan Kaufmann Publishers, 2001. – 249 p.
17. Barbara Chapman, Gabriele Jost, Ruud van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, 2008. – 353 p.
18. Ruud Van Der Pas, Eric Stotzer, Christian Terboven, Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD, The MIT Press, 2017. – 392 p.
19. Timothy G. Mattson, Yun (Helen) He, Alice E. Koniges, The OpenMP Common Core, The MIT Press, 2019. – 320 p.
20. OpenMP Application Programming Interface Specification Version 5.2, November 2021. – 649 p.
21. James Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core processor Parallelism, O'Reilly Media, 2007. – 336 p.
22. Michael Voss, Rafael Asenjo, James Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, Apress, 2019. – 820 p.
23. Hagit Attiya, Jennifer Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2 ed., John Wiley & Sons, 2004. – 414 p.
24. George Em Karniadakis, Robert M. Kirby II, Parallel Scientific Computing in C++ and MPI, Cambridge University Press, 2003. - 696 p.

25. Victor Eijkhout, Parallel Programming for Science Engineering: Using MPI, OpenMP, and the PETSc library, 2nd ed., Texas Advanced Computing Center, 2020. – 769 p.
26. William Gropp, Ewing Lusk, Anthony Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd ed., The MIT Press, 2014. - 336 p.
27. William Gropp, Torsten Hoefler, Rajeev Thakur, Ewing Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface, The MIT Press, 2014. - 392 p.
28. Frank Nielsen, Introduction to HPC with MPI for Data Science, Springer International Publishing, 2016. - 304 p.
29. MPI: A Message-Passing Interface Standard, Version 3.1, MPI Forum, University of Tennesse, June 4, 2015. - 836 p.

НАВЧАЛЬНЕ ВИДАННЯ

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ В ОС LINUX

Навчальний посібник

Редактор
Технічний редактор
Комп’ютерне верстання
Художник-дизайнер

Режим доступу:
http://eom.lp.edu.ua/textbooks/np-pposlinux.pdf

Видавець і виготівник: Видавництво Львівської політехніки
Свідоцтво суб’єкта видавничої справи ДК №4459 від 27.12.2012 р.

вул. Ф. Колесси, 4, Львів, 79013
тел. +380 32 2584103, факс +380 32 2584101
vlp.com.ua, ел. пошта: vmr@vlp.com.ua