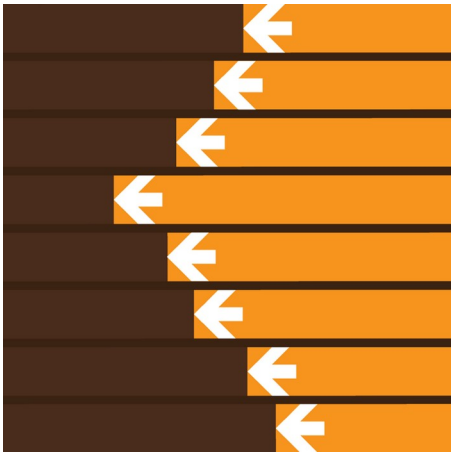
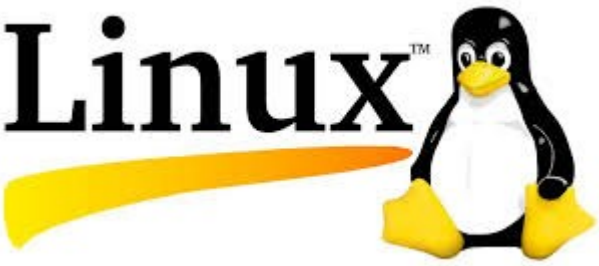
С. І. Гоменюк, А. О. Лісняк,
О. В. Кудін, С. М. Гребенюк, О. В. Чопорова

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ





Міністерство освіти і науки України
Запорізький національний університет

С. І. Гоменюк, А. О. Лісняк,
О. В. Кудін, С. М. Гребенюк, О. В. Чопорова

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Навчальний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»

Затверджено
вченою радою ЗНУ
Протокол № __ від _____ 2025 р.

Запоріжжя
2025

УДК: 004.421(075.8)
П18

Гоменюк С. І., Лісняк А. О., Кудін О. В., Гребенюк С. М., Чопорова О. В. Паралельні та розподілені обчислення : навчальний посібник для здобувачів ступеня вищої освіти магістра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Інженерія програмного забезпечення». Запоріжжя : Запорізький національний університет, 2025. 153 с.

У навчальному посібнику в систематизованому вигляді подано програмний матеріал, присвячений розробці паралельних та розподілених програм, що викладається студентам при вивченні ними відповідних дисциплін з програмування. Основна увага приділяється засвоєнню знань з загальної теорії розробки паралельних програм, а також застосуванню найбільш поширених на сьогодні відповідних низько- та високорівневих бібліотек. До кожного з шести розділів посібника пропонуються необхідні для опанування матеріалу теоретичні відомості, наводяться приклади написання програм та контрольні запитання.

Видання призначено для здобувачів ступеня вищої освіти магістра спеціальності «Інженерія програмного забезпечення», які навчаються за освітньо-професійною програмою «Інженерія програмного забезпечення».

Рецензент

Г. М. Шило, доктор технічних наук, професор, завідувач кафедри комп’ютерних наук Запорізького національного університету

Відповідальний за випуск

А. О. Лісняк, кандидат фізико-математичних наук, доцент, завідувач кафедри програмної інженерії

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ.....6
ПЕРЕДМОВА.....7
ТЕМА 1 ЗАГАЛЬНІ ПОНЯТТЯ ПРО ПАРАЛЕЛІЗМ.....10
1.1 Основні поняття та визначення.....10
1.2 Класифікація багатопроцесорних систем.....19
1.3 Конкурентність і гонитва.....22
Питання для самоперевірки.....28
Вправи.....28
ТЕМА 2 НИЗЬКОРІВНЕВА БІБЛІОТЕКА P-THREAD.....30
2.1 Стандарт POSIX.....30
2.2 Базові поняття P-thread.....31
2.2.1 Створення потоків.....32
2.2.2 Синхронізація потоків.....35
2.2.3 Завершення потоків.....39
2.2.4 Передача параметрів у потоки.....43
2.2.5 Застосування м’ютексів.....49
2.2.6 Від’єднання потоків.....56
Питання для самоперевірки.....62
Вправи.....62
ТЕМА 3 СТАНДАРТНІ ЗАСОБИ РОЗРОБКИ БАГАТОПОТОКОВИХ ПРОГРАМ МОВИ C++.....64
3.1 Клас std::thread.....64
3.1.1 Запуск потоку.....64
3.1.2 Синхронізація потоків.....66
3.1.3 Запуск потоків у класах.....73
3.2 Клас std::mutex.....78

3.3 Клас std::atomic.....82
3.4 Асинхронні виклики.....86
3.5 Синхронізація доступу до стандартного потоку виводу.....92
3.6 Автоматичне розпаралелювання програмного коду із застосуванням стандартних алгоритмів.....98
Питання для самоперевірки.....100
Вправи.....101
ТЕМА 4 ВИСОКОРІВНЕВА БІБЛІОТЕКА BOOST.....102
4.1 Автозбирання програм в Linux.....102
4.2 Створення потоків із застосуванням класу boost::thread.....107
4.3 Синхронізація доступу до спільних ресурсів.....109
4.4 Переривання потоків.....113
4.5 Групова робота з потоками.....117
Питання для самоперевірки.....120
Вправи.....121
ТЕМА 5 ТЕХНОЛОГІЯ OPENMP.....123
5.1 Директиви OpenMP.....123
5.2 Паралельні та послідовні області.....128
5.3 Цикли OpenMP.....133
5.4 Синхронізація потоків.....135
Питання для самоперевірки.....137
Вправи.....138
ТЕМА 6 РЕАЛІЗАЦІЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MPI.....139
6.1 Налаштування та тестування MPI.....140
6.2 Базові функції бібліотеки MPI.....142
6.3 Обмін повідомленнями.....145
Питання для самоперевірки.....151
Вправи.....152

РЕКОМЕНДОВАНА ЛІТЕРАТУРА.....153
ВИКОРИСТАНА ЛІТЕРАТУРА.....154

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ОС – операційна система

ПЗ – програмне забезпечення

ПК – персональний комп’ютер

ЦП – центральний процесор

API – Application Programming Interface

GNU – GNU's Not UNIX

IDE – Integrated Development Environment

IEEE – Institute of Electrical and Electronics Engineers

LSB – Linux Standard Base

MIMD – Multiple Instruction Multiple Data

MISD – Multiple Instruction Single Data

MPI – Message Passing Interface

MPP − Massively Parallel Processing

OpenMP – Open Multi-Processing

POSIX – Portable Operating System Interface

RAII – Resource Acquisition Is Initialization

SIMD – Single Instruction Multiple Data

SISD – Single Instruction Multiple Data

SMP − Symmetric Multiprocessing

STL – Standard Template Library

SUS – Single UNIX Specification

ПЕРЕДМОВА

Паралельним програмуванням називається процес поділу вихідної задачі на певну кількість більш простих та незалежних підзадач, які можуть бути виконані одночасно на різних обчислювальних пристроях (ядрах, процесорах, комп’ютерах). За рахунок такого підходу не тільки зменшується час виконання програми, а й підвищується її загальна продуктивність. Паралельне програмування активно використовується як для обробки великих масивів інформації, так і у випадках, коли обчислення слід виконати у стислі терміни (наприклад, при прогнозуванні погоди). Дисципліна відноситься до циклу професійної підготовки.

Найбільш часто на практиці при розробці паралельних програм використовують наступні підходи:

– багатопотокове програмування, коли обчислювальний процес розбивається на декілька паралельних потоків виконання (threads), які працюють у його спільній пам’яті, що дозволяє збільшити загальну продуктивність програми за рахунок використання декількох ядер процесора;

– розподілене програмування, коли обчислювальна задача розподіляються між кількома комп’ютерами, які працюють у мережі, що дозволяє підвищити загальну продуктивність при обробці великих обсягів даних;

– асинхронне програмування:, коли потоки виконання не блокуються для очікування завершення розрахунків, а продовжують роботу над іншими задачами, що дозволяє збільшити загальну ефективність використання обчислювальних ресурсів та зменшити час очікування.

Очевидно, що широке розповсюдження останнім часом багатопроцесорних комп’ютерів та обчислювальних мереж, робить важливим вміння сучасних програмістів розробляти паралельні та розподілені програми.

Метою дисципліни «Паралельні та розподілені обчислення» є опанування основними техніками та засобами паралельного та розподіленого програмування.

Завданням даного курсу є:

– ознайомлення з основними поняттями паралельного та розподіленого програмування;

– ознайомлення з низькорівневою бібліотекою P-thread;

– набуття умінь використовувати стандартну бібліотеку мови програмування C++ для розробки паралельних та асинхронних програм;

– опанування уміннями використовувати бібліотеку Boost для розробки багатопотокових програм;

– ознайомлення з основами технології OpenMP;

– ознайомлення з основами розробки розподілених програм з використанням технології MPI.

В результаті вивчення курсу студент повинен **знати**:

– основні бібліотеки та стандарти розробки паралельних та розподілених програм;

– технології розробки багатопотокових та асинхронних програм;

– технології розробки розподілених програм.

Крім того він повинен **вміти**:

– використовувати ітеративний паралелізм;

– використовувати критичні секції;

– використовувати бар’єри;

– створювати паралельні програмні засоби у системах зі спільною пам’яттю;

– створювати програмні засоби на базі розподілених обчислень.

Також слід зазначити, що розробка паралельних та розподілених програм є досить складним завданням, оскільки їх налагодження та тестування є

набагато складнішою задачею у порівнянні з написанням класичних послідовних застосунків.

Деякі сучасні мови програмування, наприклад, Go забезпечують вбудовану можливість паралельного програмування. Проте при розробці високопродуктивного програмного забезпечення з багатьох причин частіше за все застосовуються мови програмування C та C++, на яких написано більшість сучасних операційних систем та багато іншого професійного програмного коду. На жаль, на сьогодні вбудованих можливостей для розробки паралельних програм ці мови не мають. Однак, для створення такого програмного забезпечення розроблено значну кількість бібліотек, що реалізують весь необхідних функціонал. Тут можна відмітити низькорівневу бібліотеку POSIX Threads, яка дозволяє створювати багатопотокові програми в середовищі Unix-подібних операційних систем. При використанні об’єктно-орієнтованого підходу у мові C++ (починаючи зі стандарту C++11) можна використовувати відповідні класи бібліотеки STL, які реалізують багатопотокові та асинхронні обчислення. Альтернативою є багатоплатформна бібліотека Boost, яка розширює можливості STL, зокрема і для створення багатопотокових програм. Крім того, широкого поширення набули технології OpenMP та MPI, які дозволяють на мовах програмування C/C++ та Fortran створювати паралельні програми для систем зі спільною та розподіленою пам’яттю.

Таким чином, метою цього навчального посібника є ознайомлення студентів спеціальності 121 «Інженерія програмного забезпечення» з основами розробки паралельних та розподілених програм на мовах програмування C/C++ в Unix-подібних ОС при вивченні дисциплін з програмування. Посібник сприятиме формуванню у студентів таких спеціальних компетентностей: здатність особи розв’язувати складні задачі і проблеми у певній галузі професійної діяльності або у процесі навчання, що передбачає проведення досліджень та/або здійснення інновацій та характеризується невизначеністю умов і вимог; здатність до абстрактного мислення, аналізу та синтезу; здатність

аналізувати предметні області, формувати, класифікувати вимоги до програмного забезпечення; здатність розробляти і реалізовувати наукові та/або прикладні проєкти у сфері інженерії програмного забезпечення; здатність розвивати і реалізовувати нові конкурентоспроможні ідеї в інженерії програмного забезпечення; здатність критично осмислювати проблеми у галузі інформаційних технологій та на межі галузей знань, інтегрувати відповідні знання та розв’язувати складні задачі у широких або мультидисциплінарних контекстах; здатність проектувати, керувати та використовувати хмарну інфраструктуру для отримання обчислювальних потужностей, обробки та збереження даних, застосування хмарних сервісів у власних програмних продуктах; здатність застосовувати та розвивати фундаментальні та міждисциплінарні знання для успішного розв'язання завдань інженерії програмного забезпечення.

ТЕМА 1 ЗАГАЛЬНІ ПОНЯТТЯ ПРО ПАРАЛЕЛІЗМ

1.1 Основні поняття та визначення

Апаратним паралелізмом називають можливість процесора одночасно виконувати декілька команд. Тривалий час персональні комп’ютери (ПК) у більшості випадків обладнувалися одним фізичним центральним процесором (ЦП) з одним обчислювальним модулем (ядром), що виключало можливість застосування апаратного паралелізму. Перші операційні системи (ОС) для ПК були **однозадачними**, тобто одночасно могли виконувати лише одну задачу (програму). У таких ОС програма, що виконується, монопольно має доступ до всіх ресурсів комп’ютера: ЦП, пам’яті, пристроїв вводу-виводу тощо. Якщо ж, наприклад, поточна програма чекає на введення даних користувачем, то її виконання блокується, а ЦП фактично простоює. Це приводить до нераціонального використання ПК. Тому намагання підвищити ефективність застосування обчислювальної техніки підштовхнуло виробників до створення **багатозадачних** ОС (Unix, Linux, Windows тощо), в яких одночасно можуть виконуватися декілька програм. Водночас, якщо ЦП не підтримує апаратний паралелізм, то ОС завдяки наявності спеціальної підсистеми – **планувальника**, може перемикати програми, що виконуються, між ЦП таким чином, щоб у користувача складалося враження їх паралельної роботи. Це називається **квазі-багатозадачністю**. Оскільки у будь-якому сучасному ПК зазвичай одночасно виконується програм набагато більше, ніж у нього є процесорів (ядер), то перемикання (яке називається **контекстним перемиканням**) програм в ОС здійснюється постійно.

Аби програми, що одночасно виконуються, не заважали одна одній, вони працюють у певному віртуальному середовищі, яке прийнято називати процесом.

Процес – це одна з найбільш базових абстракцій будь-якої багатозадачної ОС. Він представляє собою двійковий код, який виконується в даний момент часу. З кожним процесом асоціюються його дані, пам’ять, відкриті файли і т. п. В сучасних ОС з процесом також можуть бути пов’язані і потоки виконання.

Потік[1] (thread) – це окремий елемент (програмний код), який може виконуватися паралельно (одночасно) з іншими потоками всередині процесу (рис. 1.1).

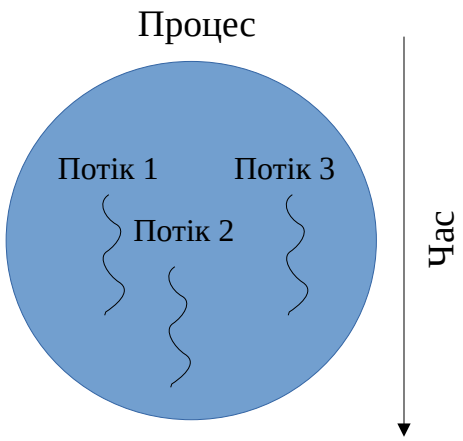


Рис. 1.1 – Приклад процесу,
що містить три потоки

На логічному рівні потік – це найменша виконувана абстракція, що обробляється планувальником ОС. Процес може містити один або декілька потоків. У першому випадку сам процес і є потоком. Такі процеси прийнято називати **однопотоковими**. Ранні версії ОС Unix підтримували тільки однопотокові процеси.

1 Деякими авторами вважається невдалим переклад англійського слова thread («нитка») як «потік», оскільки цей термін конкурує з поняттям потоку (stream) вводу-виводу.

Якщо ж процес містить більше одного потоку, то при його виконанні одночасно може виконується кілька дій. Такі процеси називаються **багатопотоковими**. Прикладом багатопотокового процесу є копіювання файлів в Windows (рис. 1.2). Тут один потік безпосередньо реалізує процедуру копіювання, а інший – анімацію прогресу операції, яка полягає у зображенні та обслуговуванні відповідного графічного віджету.

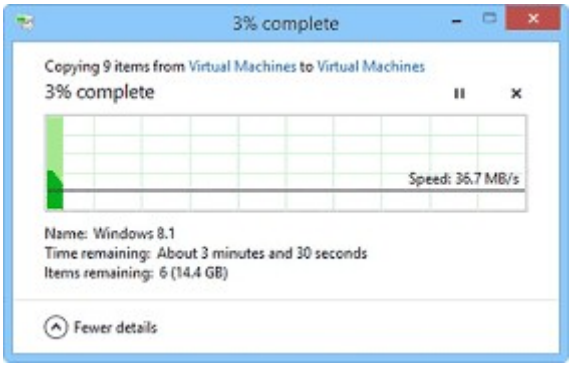


Рис. 1.2 – Багатопотоковий процес копіювання файлів

У сучасних ОС кожному процесу надається певний об’єм оперативної пам’яті (віртуальної), яка є недоступною для інших процесів. Усі потоки процесу мають доступ до його пам’яті (тобто розділяють цю пам’ять як із самим процесом, так і з іншими його потоками). Інакше кажучи, всі потоки певного процесу виконуються в спільному адресному просторі.

В ОС Linux з кожним потоком асоціюється віртуальний процесор. Таким чином, Linux організовує багатозадачність так, щоб на логічному рівні кожен процес і його потоки жодним чином не залежали від інших процесів і потоків. Процес виконується в єдиному адресному просторі пам’яті на одному або декількох віртуальних процесорах абсолютно незалежно від інших процесів, які виконуваних в даний момент часу.

Для кращого розуміння однопотокових та багатопотокових процесів розглянемо наступну аналогію. Нехай деяка IT-фірма для реалізації певного

проєкту найняла двох висококваліфікованих програмістів, кожний з яких працює в окремому приміщенні (рис. 1.3).

У цьому випадку така організація роботи програмістів має наступні переваги:

– вони один одному не заважать;

– у кожного свій персональний набір знаряддя для роботи (документація, кавоварка, ксерокс, кулер для води тощо).

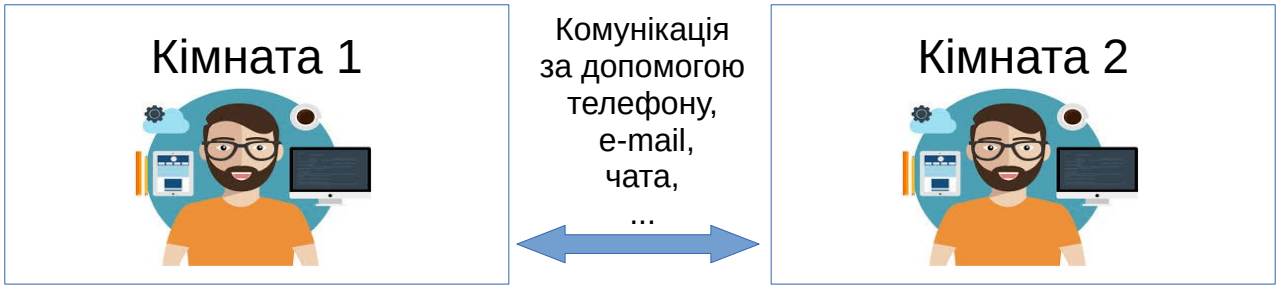


Рис. 1.3 – Модель однопотокового процесу

Але є й певні недоліки:

– ускладнена комунікація між програмістами (для цього потрібно відволікатися від роботи й телефонувати колезі або писати йому листа);

– компанії необхідно утримувати два приміщення, а також придбати два комплекти документації, два ксерокси і т. д.

Якщо ж фірма облаштує робочі місця програмістів в одному приміщенні (рис. 1.4), то такий підхід має низку наступних переваг:

– можливість працівників безпосереднього спілкування один з одним (а оскільки вони працюють над одним проєктом, то час від часу їм потрібно це робити);

– утримувати потрібно тільки одне приміщення;

– закуповувати можна тільки один комплект документації і оргтехніки.

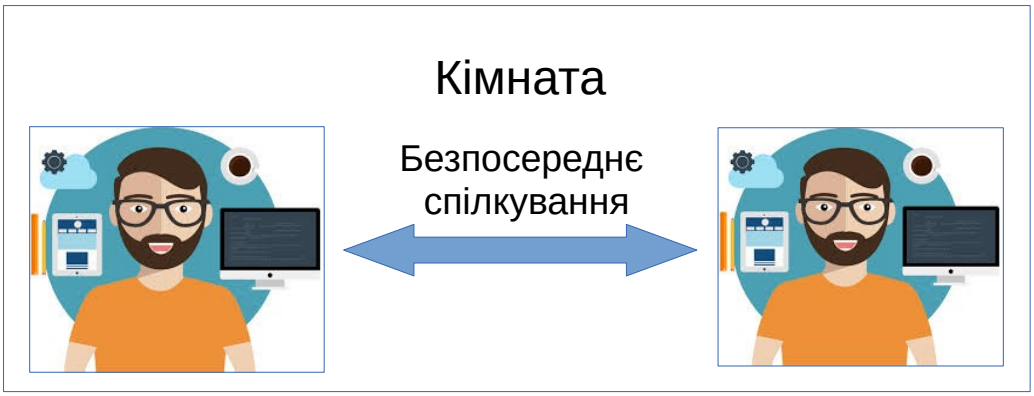


Рис. 1.4 – Модель багатопотокового процесу

Проте у такого підходу є й певні недоліки:

– програмісти можуть заважати один одному (важче сконцентруватися);

– ускладнюється доступ до загальних ресурсів (документацію, наприклад, вже читає інший колега);

– між програмістами можуть виникати конфлікти («хто випив всю каву?») і так далі.

Очевидно, що в наведеній аналогії процесом є кімната, в якій виконується робота над спільним проєктом, а потоком – програміст, який виконує цю роботу. У першому випадку мова йде про декілька однопотокових процесів, а у другому – про один багатопотоковий.

Ясно, що оскільки потоки працюють в спільному адресному середовищі, то вони можуть заважати один одному й конкурувати за спільні ресурси. Проте застосування багатопотокових процесів дозволяє досягнути реального **паралелізму**, під яким розуміється одночасне виконання двох або більше операцій. В контексті комп’ютера це означає, що система виконує кілька незалежних операцій паралельно (одночасно), а не послідовно. Поява і широке поширення комп’ютерів, обладнаних декількома процесорами або декількома

ядрами на одному кристалі – **багатоядерними** процесорами (рис. 1.5), привело до необхідності розробки програм, що підтримують паралелізм.

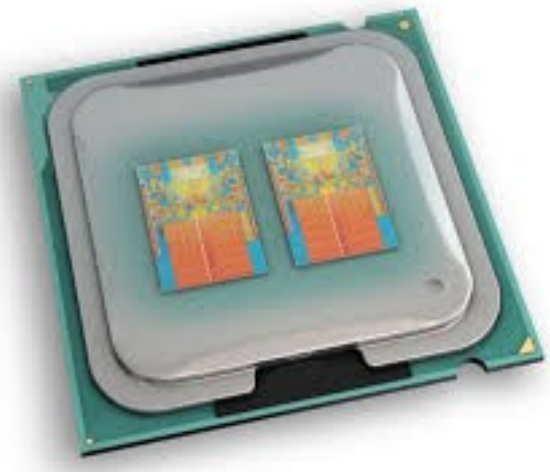


Рис. 1.5 – Двоядерний процесор

Слід зазначити, що виробники процесорів на сьогодні вважають за краще не нарощувати тактову частоту процесора (тобто швидкодію), а збільшувати кількість його ядер. Таким чином, для підвищення швидкості роботи програми потрібно створювати паралельні програми. Слід також враховувати, що навіть сучасні одноядерні процесори за рахунок декількох конвеєрів з обробки команд можуть одночасно виконувати більш ніж одну команду.

Як вже зазначалося, потік – це мінімальна одиниця, якою оперує планувальник ОС. Якщо комп’ютер обладнано одним процесором, то він в один момент часу може виконувати одну команду (без врахування апаратного паралелізму). Тому, якщо на одноядерному процесорі одночасно виконується два потоки, то планувальник ОС організує їх роботу таким чином, щоб вони по черзі використовували процесор (рис. 1.6). Слід також відмітити, що на виконання контекстного перемикання потоків (процесів) ОС вимушена витрачати певний час (тонкі сірі смуги на рисунку). У випадку ж двоядерного

процесора виконання двох потоків може здійснюватися одночасно, тобто паралельно.

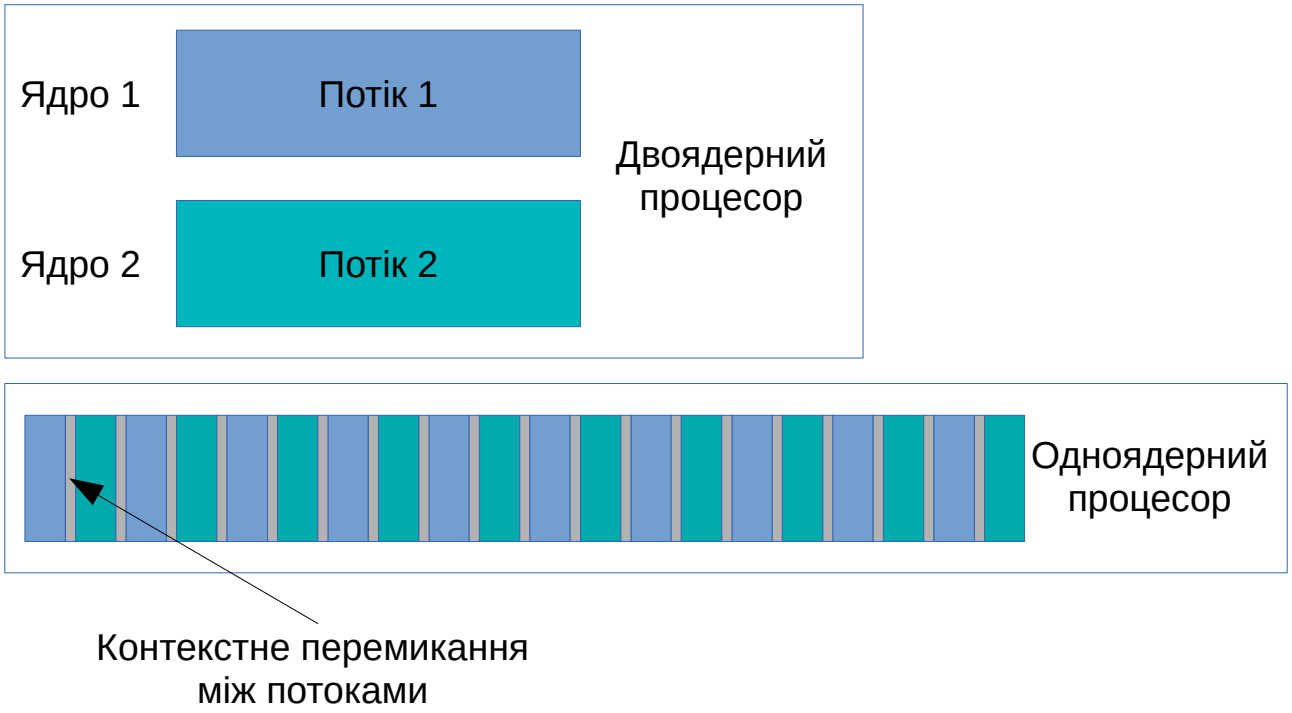


Рис. 1.6 – Організація багатозадачності на одно- та двохядерному процесорах

Якщо ж на двоядерному процесорі буде виконуватися три потоки, то без контекстного перемикання між ними також не обійтися (рис. 1.7). Оскільки кількість потоків, що одночасно виконується в будь-якій сучасній ОС, може сягати декількох тисяч, то становиться очевидним, що планувальник ОС буде виконувати контекстне перемикання процесів та потоків фактично завжди.

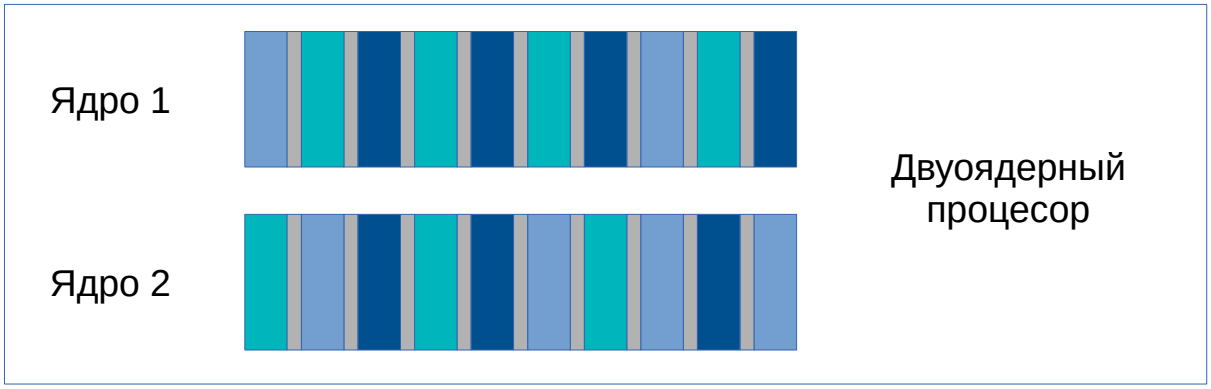


Рис. 1.7 – Перемикання трьох процесів на двоядерному процесорі

Найбільш простий способом розпаралелювання програми є розбиття її на декілька окремих процесів, що одночасно виконуються (можливо на окремих комп’ютерах), і можуть обмінюватися повідомленнями за допомогою стандартних каналів міжпроцесової комунікації (сигналів, сокетів, файлів або конвеєрів) (рис. 1.8). До переваг такої реалізації можна віднести незалежність процесів один від одного, високу безпеку та можливість організації роботи в мережі (тобто розподіленої обробки даних). До недоліків – великі накладні витрати на запуск нового процесу та порівняно низьку швидкість обміну даними (особливо при реалізації міжпроцесової комунікації за допомогою файлів або невисокій пропускній здатності мережі).

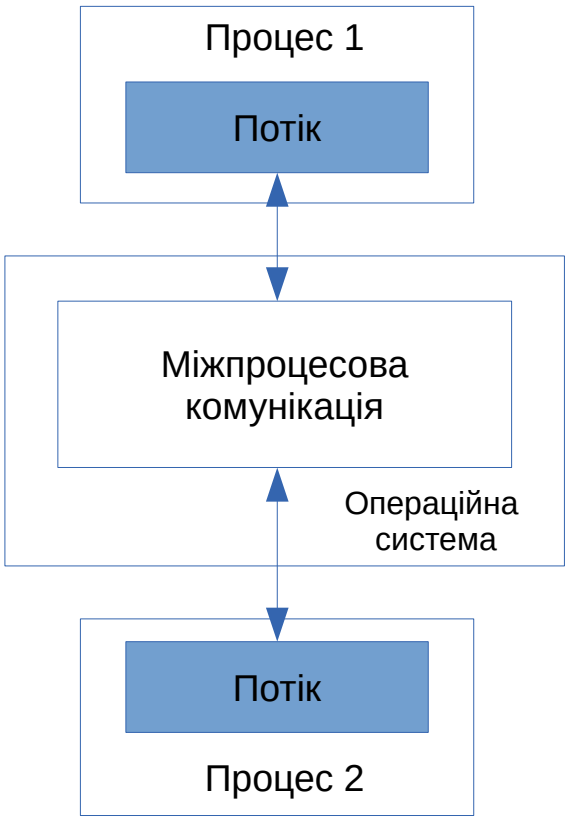


Рис. 1.8 – Комунікація між процесами

Альтернативним способом організації паралелізму є запуск декількох потоків в одному процесі. Кожний потік працює при цьому незалежно від інших, однак, їм доводиться розділяти спільний адресний простір (рис. 1.9).

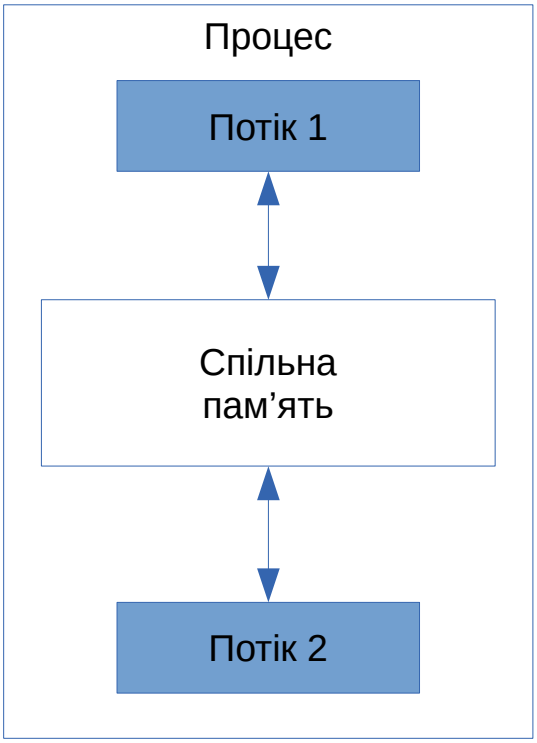


Рис. 1.9 – Комунікація між потоками в одному процесі

Основними перевагами використання потоків, як засобу організації паралелізму, є низькі накладні витрати на запуск і обслуговування потоків, а також – швидка й проста комунікація між ними. В якості недоліків можна зазначити необхідність узгодження даних між потоками та можливу конкуренцію між ними за ресурси.

Таким чином, на сьогодні, враховуючи масове поширення комп’ютерних систем, обладнаних багатоядерними процесорами, розробка паралельних програм є важливою задачею при розробці сучасного програмного забезпечення (ПЗ).

Багатопотокові програми мають наступні важливі переваги:

– покращена структура програми (деякі алгоритми більш ефективно представляються у вигляді декількох незалежних обчислювальних потоків, ніж у вигляді одного процесу);

– наявність реального паралелізму;

– покращена реактивність (використання потоків дозволяє поліпшити реагування процесу на дії користувача навіть при виконанні тривалих обчислювальних процедур);

– можливість уникнути блокування вводу-виводу (блокується тільки потік, який, наприклад, очікує завершення операції вводу даних, у той же час інші – продовжують своє виконання);

– більша швидкість виконання (накладні витрати на перемикання між потоками значно нижчі, ніж у випадку перемикання або обміну інформацією між процесами);

– раціональне використання пам’яті (потоки працюють зі спільною пам’яттю, що дозволяє більш ефективно її використовувати).

Головними недоліками багатопотокових програм є:

– більш висока складність розробки (багатопотокові програми набагато складніші в розробці, особливо це стосується питань їх налагодження та пошуку помилок);

– складність синхронізації доступу до спільної пам’яті.

1.2 Класифікація багатопроцесорних систем

Найбільш відому класифікацію багатопроцесорних комп’ютерних систем запропонував у 1966 році американський вчений М. Флінн (Michael J. Flynn). Згідно з нею всі комп’ютерні системи можна умовно поділити на чотири великих групи:

1) SISD (Single Instruction Single Data − одиночний потік команд, одиночний потік даних) − послідовні комп’ютери в яких виконується лише одна програма;

2) SIMD (Single Instruction Multiple data – одиночний потік команд, множинний потік даних) – паралельний комп’ютер, в якому виконується одна програма, но кожна команда якої обробляє масив даних (векторний паралелізм);

3) MISD (Multiple Instruction Single Data − множинний потік команд, одиночний потік даних) − не реалізований на практиці тип комп’ютера, в якому декілька команд працюють з одним елементом даних;

4) MIMD (Multiple Instruction Multiple Data − множинний потік команд, множинний потік даних) − паралельний комп’ютер, в якому декілька програм одночасно й незалежно одна від одної обробляють власні дані (паралелізм багатопроцесорних систем).

Тут під «потоком команд» розуміється послідовність команд однієї програми, а під «потоком даних» – інформація, що обробляється однією програмою.

На сьогодні найбільш поширеним типом паралелізму є MIMD, оскільки більшість сучасних комп’ютерів обладнано декількома процесорами. Всі такі комп’ютери можна поділити на дві окремі групи: зі спільною (shared) (рис. 1.10) та розподіленою (distributed) пам’яттю (рис. 1.11).

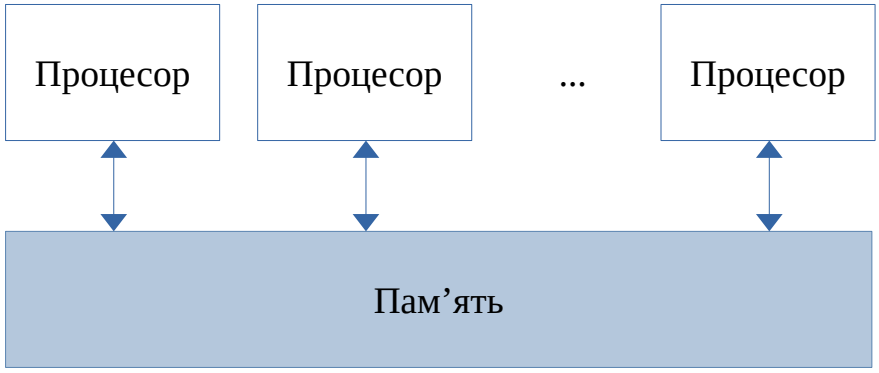


Рис. 1.10 − Система зі спільною пам’яттю

Головною різницею між ними є спосіб доступу до пам’яті. У комп’ютерах зі спільною пам’яттю весь адресний простір є єдиним, що дозволяє декільком процесорам одночасно звертатися до однієї й тієї ж адреси (змінною). Це має як свої переваги, так і недоліки, які будуть розглянуті нижче. Крім того, в таких комп’ютерах не потрібно переміщувати дані між програмами, які паралельно працюють, що спрощує як сам процес програмування, так і швидкість міжпроцесного обміну даними.

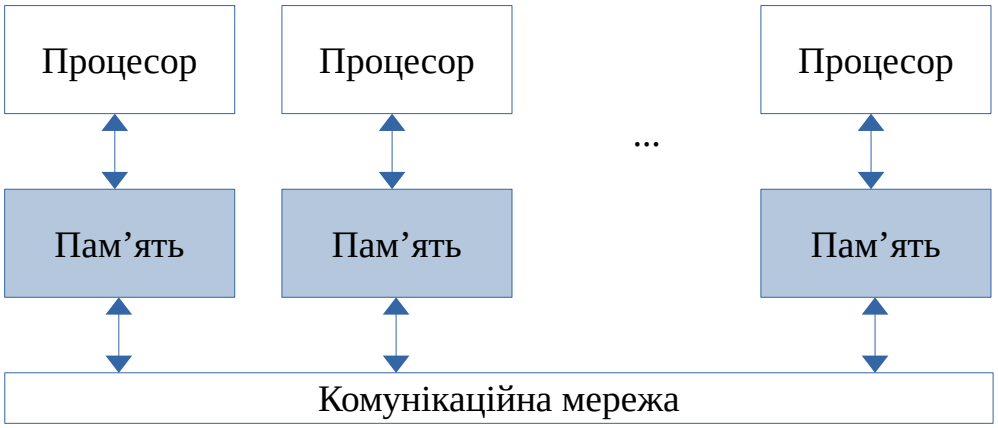


Рис. 1.11 − Система за розподіленою пам’яттю

У системах з розподіленою пам’яттю кожен процесор має окремий фізичний адресний простір пам’яті. Таким чином, для обміну інформацією між процесами, що паралельно працюють, необхідно фізично переміщувати дані між ними. А це, в свою чергу, ускладнює як програмування таких задач, так і збільшує час на міжпроцесний обмін. Проте для розв’язання великих задач такий підхід є виправданим, оскільки можливостей одного фізичного комп’ютера може не вистачити.

MIMD-системи часто розбивають на три великих класи: 1) симетричні мультипроцесори; 2) системи з масовим паралелізмом та 3) кластери.

Симетричні мультипроцесори (SMP − Symmetric Multiprocessing) − це найбільш поширений на сьогодні тип багатопроцесорних комп’ютерів (у тому числі багатоядерних), в яких усі процесори (ядра) підключені до загальної пам’яті.

Системи з масовим паралелізмом (MPP − Massively Parallel Processing) − це спеціальні типи комп’ютерів, обладнані великою кількістю процесорів з індивідуальною пам’яттю. В таких системах є спеціальний керуючий процесор, який розподіляє роботу між всіма іншими пристроями.

Кластери − це група взаємопов’язаних комп’ютерних систем (вузлів), що сумісно функціонують, утворюючи єдину логічну обчислювальну машину. Це більш дешевий, ніж MPP, варіант створення паралельного обчислювального пристрою, оскільки теоретично в кластери можна об’єднувати будь-яку кількість любих комп’ютерів (у тому числі застарілих або малопотужних). Проте загальна продуктивність кластера багато в чому залежить від швидкості та пропускної спроможності комунікаційної мережі.

1.3 Конкурентність і гонитва

Конкурентність – це спроможність декількох потоків виконуватися в періоди часу, що перекриваються. Це може призводити до стану **гонитви** (race) – проблеми синхронізації доступу потоків до деяких загальних ресурсів. В разі, якщо такими ресурсами є дані, то мова йде про **гонитву за даними** (data race). Наприклад, якщо потік *A* ще не завершився, а потік *B* вже намагається оперувати його даними (які ще можливо не готові), то поведінка процесу може стати непередбачуваною. На прикладі застосування банкомату ця ситуація може виглядати таким чином. Нехай процедура отримання грошей в банкоматі реалізується наступним алгоритмом (рис. 1.12).

Якщо, припустимо, клієнт банку знімає всі гроші зі свого рахунку, а в цей момент банк розпочав автоматичну процедуру списання комунальних платежів, то вочевидь виникає гонитва за даними – станом рахунку клієнта. Ясно, що при неправильній обробці цієї ситуації банк може втратити гроші.

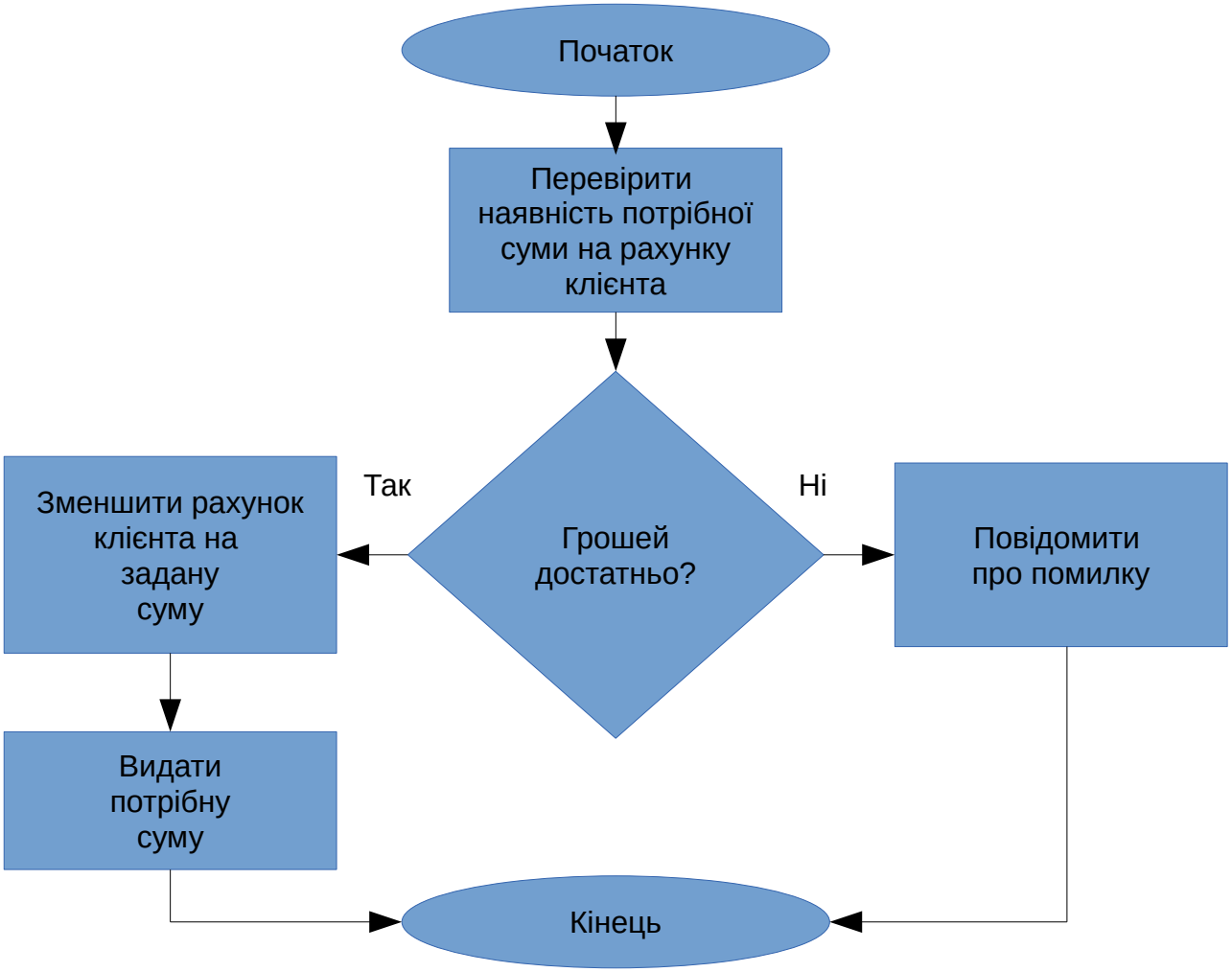


Рис. 1.12 – Блок-схема отримання грошей із застосуванням банкомату

Наприклад, якщо на рахунку було $500 і одночасно від клієнта та від автоматизованої процедури оплати комунальних платежів прийшли два запити: на зняття $500 і $400, то згідно алгоритму, приведеного на рис. 1.12, вони обидва можуть бути задоволені, хоча після цього на рахунку клієнта залишиться від’ємна величина, що призведе до втрати грошей банком.

Щоб запобігти такій ситуації потоки повинні виконати **синхронізацію**. З цією метою вони використовують так звані **м’ютекси** (від англ. mutual exclusion – взаємне виключення) – спеціальні об’єкти (**семафори**), що можуть приймати лише два значення, наприклад, «відкрито» та «закрито». Якщо потік звертається до м’ютексу і він має значення «відкрито», то потік встановлює його в значення «закрито» і може монопольно використовувати **критичну область** коду. Всі інші потоки, які конкурують за ресурс, в цей момент блокуються. Таким чином, в один момент часу тільки один потік може володіти м’ютексом. Після завершення своєї роботи з критичною областю потік, що заблокував м’ютекс, встановлює його в значення у «відкрито» й інші потоки можуть отримати до нього (а, відповідно, і до критичної області) доступ.

Повною аналогією тут є, наприклад, ванна кімната в квартирі. Перший член родини, який до неї зайшов, закриває двері на замок (використовує м’ютекс) і монопольно використовує її ресурс (душову кабіну). Всі інші члени родини повинні чекати, поки він звідти не вийде. І так далі. Тут критичною областю є ванна кімната, ресурсом – душова кабіна, а м’ютексом – замок на двері.

Таким чином, для того, щоб зробити алгоритм видачі грошей банкоматом безпечним, в нього необхідно внести зміни, наприклад так, як це зображено на рис. 1.13.

Окрім, так би мовити, логічних помилок при синхронізації даних можуть виникати й низькорівневі проблеми. Розглянемо ще один такий приклад можливого виникнення проблем, пов’язаних із синхронізацією потоків. Нехай в потоці виконується оператор мови С, який має назву інкремент.

```
// Збільшення значення цілої змінної x на 1 (інкремент)
x++;
```

Тоді байт-код, який реалізує даний оператор, можна, наприклад, описати таким чином.

load x into register
add 1 to register
store register in x

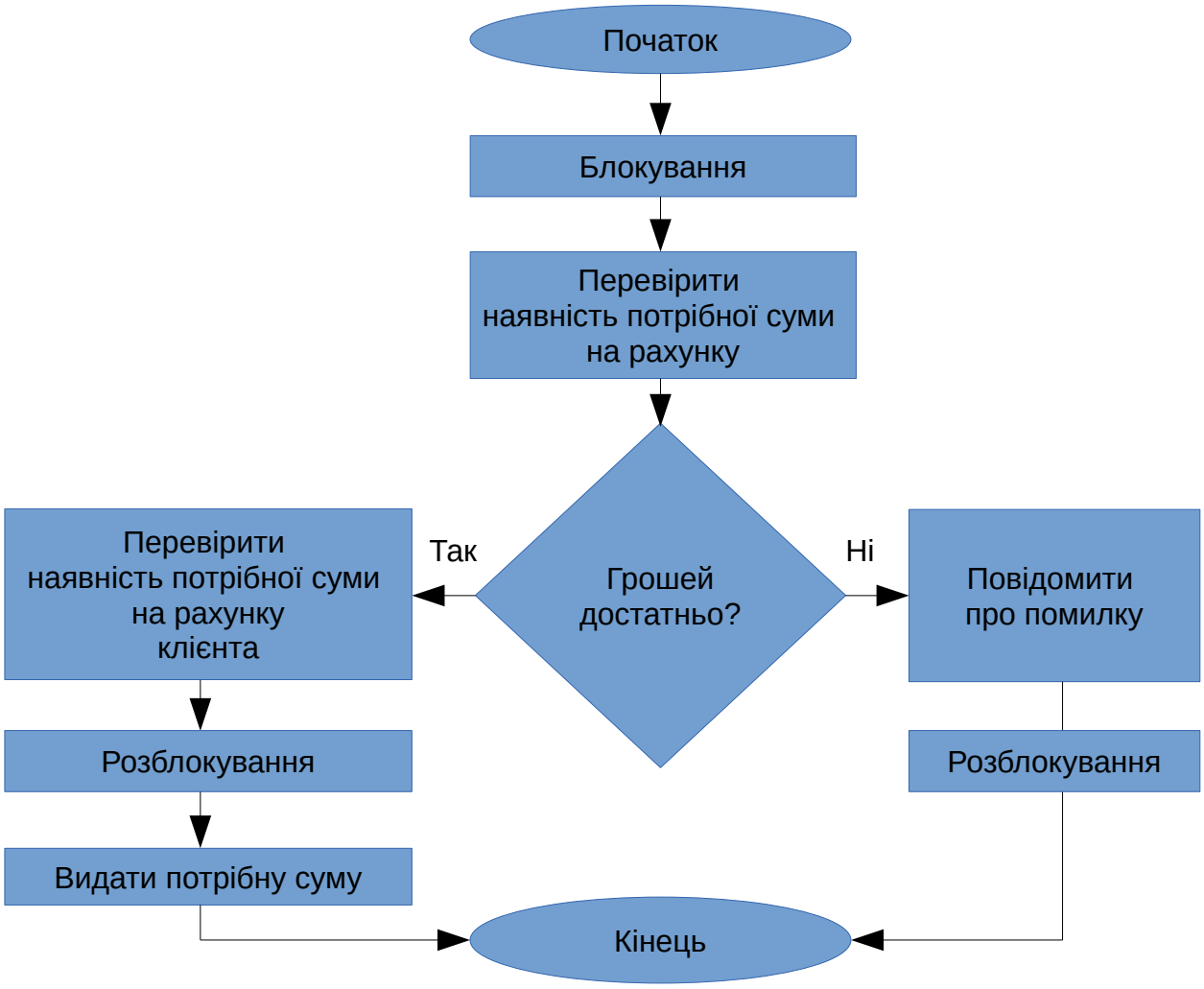


Рис. 1.13 – Приклад використання блокувань в алгоритмі роботи банкомату

При виконанні цих операцій в потоці може виникнути гонитва. Нехай, x = 5. Тоді при виконанні цього коду у двох паралельних потоках можливі наступні варіанти.

Варіант 1

Крок	Потік 1	Потік 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3	store register in x		6	6
4		load x into register	6	6
5		add 1 to register	6	7
6		store register in x	7	7

Варіант 2

Крок	Потік 1	Потік 2	x	register
1		load x into register	5	5
2		add 1 to register	5	6
3		store register in x	6	6
4	load x into register		6	6
5	add 1 to register		6	7
6	store register in x		7	7

Варіант 3

Крок	Потік 1	Потік 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3		load x into register	5	5
4	store register in x		5	5
5		add 1 to register	5	6
6		store register in x	6	6

Очевидно, що лише перший та другий варіант виконання потоків призведуть до отримання вірного результату (x = 7). При цьому вірогідність того, що виконання програми піде по третьому (катастрофічному) варіанту, є далеко не нульовою!

На жаль, використання блокувань не завжди є панацеєю, оскільки на практиці при реалізації багатопотокових програм можуть виникати так звані взаємні блокування (deadlock).

Взаємне блокування (**клінч**) – це ситуація, коли два потоки чекають закінчення роботи один одного і, таким чином, жоден з них не може закінчитися (рис. 1.14). При застосуванні м’ютексів взаємне блокування відбувається, наприклад, тоді, коли потоку *A* потрібен м’ютекс, яким володіє потік *B*, і навпаки.

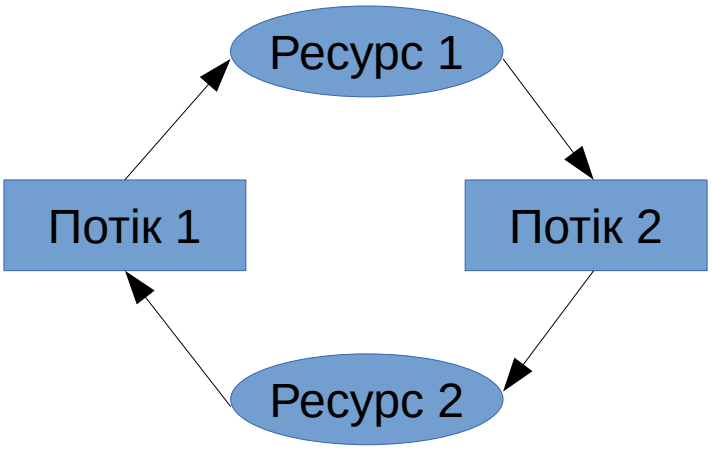


Рис. 1.14 – Взаємне блокування двох потоків, які потребують ресурси один одного

Фактично єдиним способом профілактики клінчів є правильне проєктування послідовності роботи потоків програми.

Питання для самоперевірки

1. Дайте визначення апаратного паралелізму.
2. Яка різниця між однозадачною та багатозадачною операційними системами?
3. Яку функцію виконує планувальник операційної системи?
4. Дайте визначення квазібагатозадачності.

5. Яка різниця між процесом та потоком?

6. Чи підтримували перші версії ОС Linux запуск багатопотокових програм?

7. Перерахуйте переваги та недоліки організації багатозадачності за допомогою однопотокових процесів?

8. Які переваги та недоліки мають багатопотокові програми у порівнянні з однопотоковими?

9. Що називається апаратним паралелізмом?

10. Яким чином організовано виконання трьох потоків на двоядерному процесорі?

11. Що називають конкурентністю?

12. Розкрийте поняття гонитви за даними? Наведіть приклади.

13. Що таке синхронізація та якими засобами вона здійснюється?

14. Дайте визначення deadlock.

15. Які методи попередження виникнення клінчів ви знаєте?

Вправи

1. Визначте характеристики вашого комп’ютеру. Скільки в нього процесорів або ядер? Яка їх робоча частота?

2. Визначте, скільки потоків одночасно виконується на вашому комп’ютері.

3. Складіть блок-схему алгоритму перемноження двох матриць. Спробуйте створити його паралельну версію.

ТЕМА 2 НИЗЬКОРІВНЕВА БІБЛІОТЕКА P-THREAD

2.1 Стандарт POSIX

ОС **Unix** (UNIX є зареєстрованим товарним знаком The Open Group) була розроблена наприкінці 60-х – початку 70-х років минулого сторіччя і на сьогодні є однією з найбільш відомих систем, яка справила великий вплив на всю сучасну комп’ютерну та програмну індустрію. Однією з важливих особливостей Unix є те, що вона цілком написана на мові програмування С, що дозволило її легко переносити на різні типи комп’ютерних платформ. Серед найбільш відомих її нащадків можна виділити Linux, Android, macOS, Solaris та багато інших.

POSIX (Portable Operating System Interface) – це прикладний програмний інтерфейс або **API** (Application Programming Interface) ОС Unix, що регламентує всі аспекти системного програмування в середовищі Unix-подібних ОС: від мови програмування С, яка є стандартною для системного програмування, до переліку низькорівневих функцій та бібліотек, які реалізують весь потрібний користувачу функціонал. Стандарт POSIX було розроблено у середині 80-х років минулого сторіччя міжнародною організацією інженерів у галузі електротехніки, радіоелектроніки та радіоелектронної промисловості – **IEEE** (Institute of Electrical and Electronics Engineers).

Альтернативою POSIX тривалий період часу був стандарт **SUS** (Single UNIX Specification), який розроблявся промисловим консорціумом OSF (Open Software Foundation) та X/Open і описував перелік вимог до ОС, яким вона повинна була відповідати, щоб належати до класу UNIX. Проте на сьогодні SUS де-юре є складовою частиною POSIX.

Слід зазначити, що ОС Linux формально не є Unix, хоча й дуже на неї схожа. Тому всі розробники цієї системи намагаються дотримуватися вимог стандарту POSIX, проте для Linux існує і свій власний стандарт **LSB** (Linux Standard Base), який в цілому відповідає POSIX, хоча й має певні його доповнення. LSB розроблюється некомерційним консорціумом Linux Foundation, який займається просуванням, захистом та стандартизацією цієї ОС.

2.2 Базові поняття P-thread

Як вже зазначалося, POSIX – це стандартизований API ОС Unix, в якому в тому числі регламентується і можливість роботи з потоками, які в цьому стандарті називаються **P-thread** або **Pthread** (POSIX Threads). Вони реалізовані засобами низькорівневої бібліотеки glibc (GNU C Library), яка розширює можливості мови програмування С в тому числі і для розробки багатопотокових програм. Тут слід зазначити, що рекурсивний акронім GNU (GNU's Not UNIX) позначає в загальному вигляді вільне програмне забезпечення.

Функції для роботи з P-thread визначаються в заголовному файлі pthread.h. Вони можуть бути умовно поділені на дві групи:

1) управління потоками (створення, видалення, з’єднання і від’єднання потоків тощо);

2) синхронізація потоків.

У більшості випадків назви функцій P-thread розпочинаються з префіксу «pthread_». В якості параметрів потокові функції можуть приймати змінні спеціальних типів, які також визначені в pthread.h. Найбільш вживаними серед них є такі:

– **pthread_t** – тип даних, що призначений для зберігання дескриптору (унікального ідентифікатора) потоку;

- **pthread_attr_t** – атрибутів потоку;
- **pthread_mutex_t** – м’ютексів.

2.2.1 Створення потоків

Для створення потоку в P-thread використовується спеціальна функція pthread_create(), яка має наступний шаблон.

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

У разі успішного створення потоку функція pthread_create() повертає 0 і в першому аргументі **thread** зберігає ідентифікатор новоствореного потоку. Другий аргумент **attr** містить атрибути потоку (якщо в якості даного аргументу передати значення NULL, то потік буде створений з набором стандартних атрибутів). Аргумент **start_routine** містить покажчик на функцію, яка визначає програмний код (тіло) потоку. Ця функція може приймати один аргумент – **arg**. Її сигнатура може мати, наприклад, такий вигляд:

void *thread_code(void *arg);

Таким чином, після запуску функції pthread_create(), потік починає свою роботу, виконуючи задану функцію (thread_code() в нашому прикладі), яка приймає єдиний аргумент – безтиповий покажчик і повертає значення такого ж типу. Потік виконується до тих пір, поки не закінчиться виконання функції, що утворює його тіло, або поки не закінчить роботу головний потік (якщо не робити спеціальних дій, щоб цьому завадити).

Розглянемо наступний приклад простої програми, в якій створюється потік.

```
#include <stdio.h>
#include <pthread.h>
#define MAX_ITER 10
// Функція потоку
void *thread_code(void *arg)
{
	for (int i = 0; i < MAX_ITER; i++)
		printf("Child thread: %d\n",  i);
	return NULL;
}

int main (void)
{
	pthread_t ptid;

	// Створення дочірнього потоку
	if (pthread_create(&ptid, NULL, thread_code, NULL) != 0)
	{
		perror("pthread_create()");
		return 1;
	}
	// Код головного потоку
	for (int i = 0; i < MAX_ITER; i++)
		printf("Main thread: %d\n",  i);
	return 0;
}
```

Компіляція багатопотокових програм в Linux обов’язково повинна відбуватися з підключенням бібліотеки pthread. В даному випадку це можна здійснити такою командою (при умові, що вихідний текст програми знаходиться у файлі example1.c).

```
cc example1.c -o example1 -l pthread
```

Тут ключ компілятора «-o» визначає ім’я програми, яку він згенерує в разі успіху (за замовченням компілятор автоматично генерує програму, що має назву a.out), а «-l» – необхідні для трансляції додаткові бібліотеки.

Запуск цієї програми (команда «./example1») приведе до наступного результату (рис. 2.1).

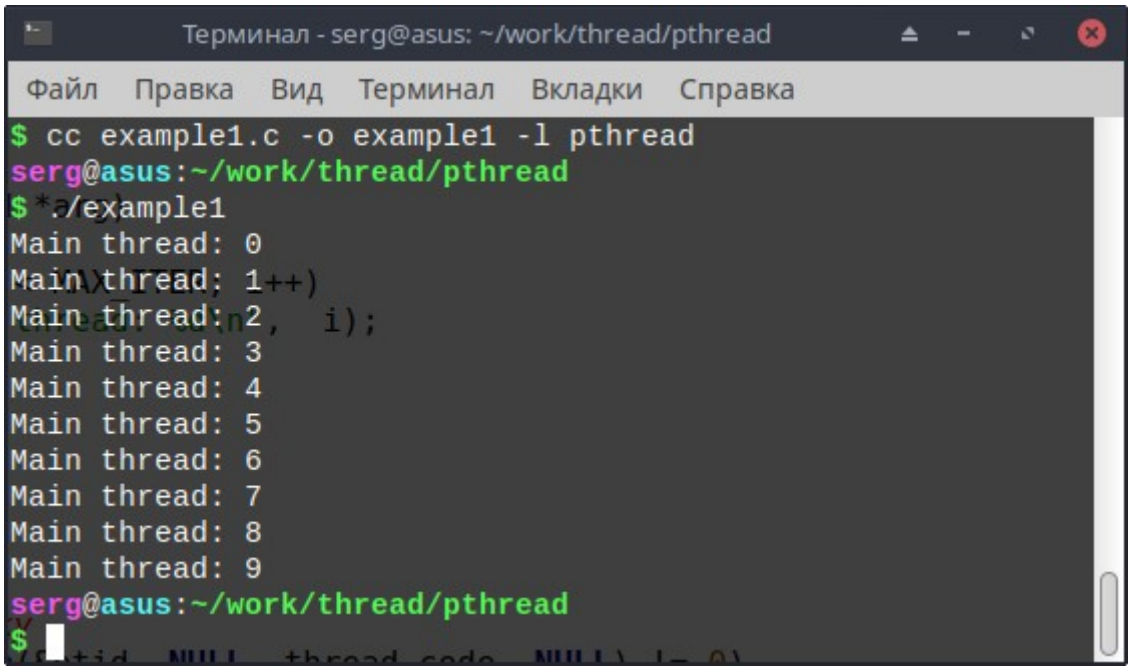


Рис. 2.1 – Результат роботи програми example1

На перший погляд він дещо несподіваний, оскільки з наведеного скріншоту видно, що код дочірнього потоку не виконувався. Проте цей факт має наступне пояснення. Оскільки всі дочірні потоки за замовченням автоматично

завершують свою роботу при завершенні головного потоку програми, якому відповідає тіло функції main(), то в нашому випадку дочірній потік просто не встиг запуститися.

2.2.2 Синхронізація потоків

Для запобігання достроковому завершенню роботи потоків бібліотека P-thread містить спеціальну функцію pthread_join().

int pthread_join(pthread_t thread_id, void **data);

Ця функція виконує очікування завершення потоку, ідентифікатор якого визначається параметром **thread_id**. В разі успіху вона повертає 0 і у параметрі **data** буде збережено дані, які повертаються потоком або за допомогою функції pthread_exit(), або з допомогою оператора return потокової функції.

Таким чином, батьківський потік при виклику функції pthread_join() призупиняє свою роботу до завершення дочірнього потоку, ідентифікатор якого було передано в якості параметру у pthread_join().

Якщо модифікувати вищенаведений приклад так, як це показано нижче, то поведінка програми дещо зміниться (рис. 2.2).

```
    // ...
    // Синхронізація потоку
    pthread_join(ptid, NULL);
    return 0;
}
```

Наведений приклад поки що також не демонструє паралельне виконання потоків. Проте і цьому є пояснення. Кожен з потків закінчив свою роботу так швидко, що планувальнику ОС просто не знадобилося виконувати для них контекстного перемикання.

Таким чином, щоб нарешті побачити паралелізм, потоки повинні виконуватися досить тривалий час. Для штучного уповільнення роботи потоків використаємо стандарту функцію sleep(), яка описана в заголовному файлі unistd.h і припиняє роботу програми на кількість секунд, задану як параметр.

Перепишемо наш приклад таким чином.

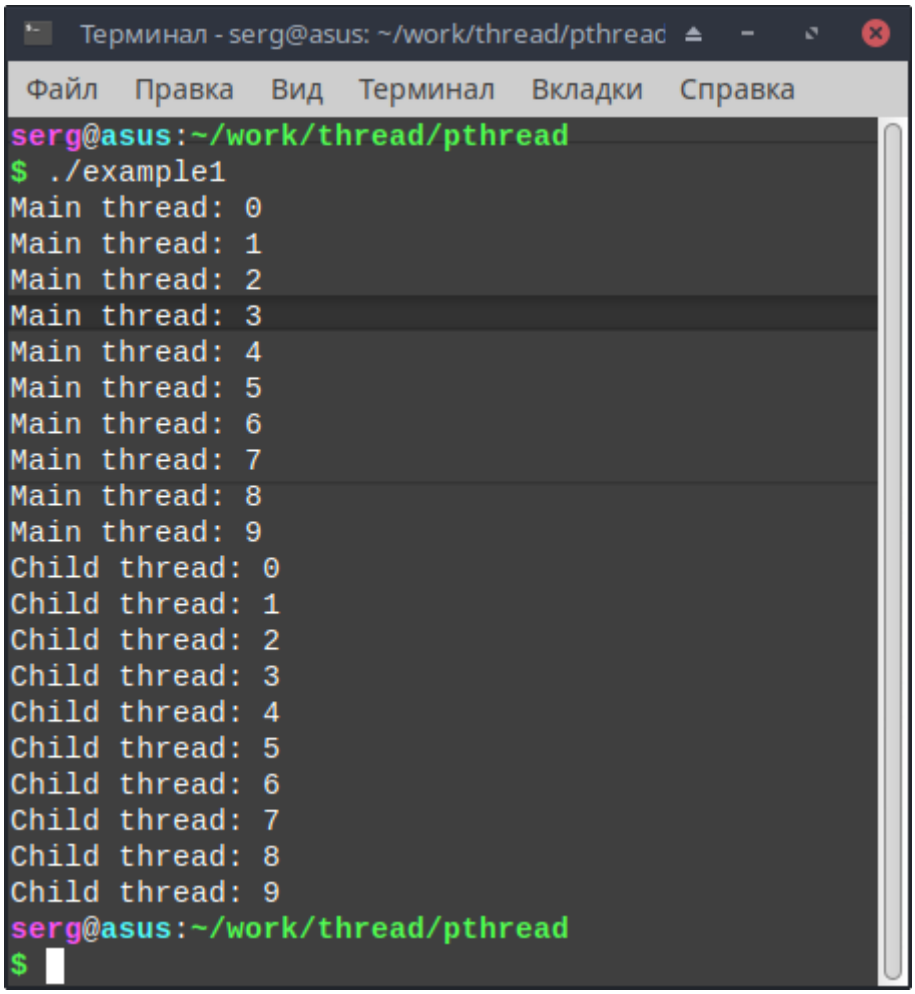


Рис. 2.2 – Результат роботи програми example1 при використанні блокування батьківського потоку

```
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

#define MAX_ITER 10

// Функція потоку
void *thread_code(void *arg)
{
	for (int i = 0; i < MAX_ITER; i++)
	{
		printf("Child thread: %d\n",  i);
		sleep(1);
	}
	return NULL;
}

int main (void)
{
	pthread_t ptid;

	// Створення потоку
	if (pthread_create(&ptid, NULL, thread_code, NULL) != 0)
	{
		perror("pthread_create()");
		return 1;
	}
	for (int i = 0; i < MAX_ITER; i++)
	{
		printf("Main thread: %d\n",  i);
		sleep(1);
```

```
    }
    // Синхронізація потоку
    pthread_join(ptid, NULL);
    return 0;
}
```

Компіляція і запуск цієї програми дасть наступний очікуваний результат (рис. 2.3). З цього скриншоту добре видно, що головний і дочірній потоки програми example1 виконуються паралельно.

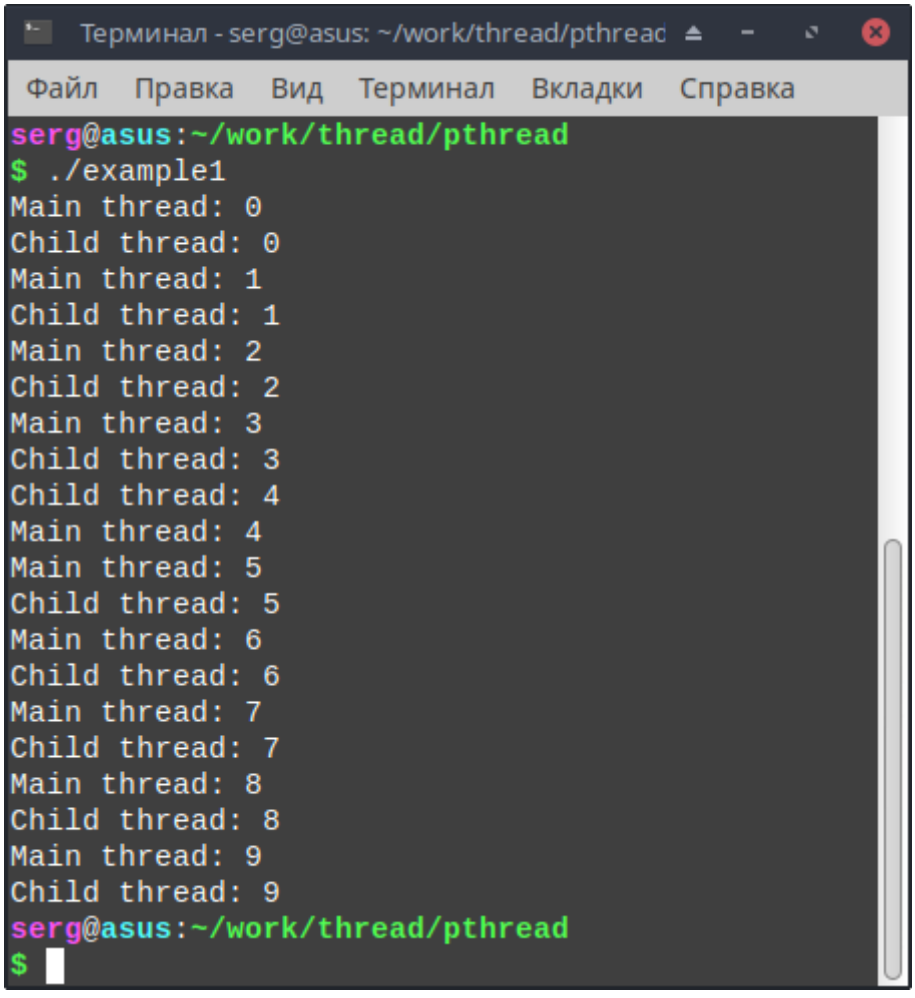


Рис. 2.3 – Приклад паралельного виконання потоків в програмі example1

2.2.3 Завершення потоків

Завершити потік можна декількома способами. Як вже зазначалося, найпростішим і природним є завершення потокової функції (шляхом досягнення кінця функції або викликом оператору «return»). Також потік можна завершити за допомогою спеціальної функції pthread_exit(), яка має наступну сигнатуру.

void pthread_exit(void *retval);

Параметр **retval** призначений для отримання значення, яке повертає дочірній потік батьківському (для аналізу, наприклад, успішності роботи потокової функції).

Крім того, потік можна примусово завершити ззовні за допомогою функції pthread_cancel().

int pthread_cancel(pthread_t thread);

В разі успіху ця функція повертає 0. Параметр **thread** визначає ідентифікатор потоку, який потрібно завершити.

Розглянемо приклад реалізації примусового завершення потоку (вихідний файл example2.c).

```
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
```

```
#define MAX_ITER 10
// Функція потоку
void *thread_code(void *arg)
{
	int i = 0;

	while (1)
	{
		printf("Child thread: %d\n",  i++);
		sleep(1);
	}
	return NULL;
}
int main (void)
{
	pthread_t ptid;

	// Створення дочірнього потоку
	if (pthread_create(&ptid, NULL, thread_code, NULL) != 0)
	{
		perror("pthread_create()");
		return 1;
	}
	// Функціонал головного потоку
	for (int i = 0; i < MAX_ITER; i++)
	{
		if (i == 4)
			pthread_cancel(ptid); // Примусове завершення потоку
		printf("Main thread: %d\n",  i);
```

```
        sleep(1);
        }
        // Блокування головного потоку до завершення дочірнього
        pthread_join(ptid, NULL);
        return 0;
}
```

В даній програмі потокова функція спроєктована таким чином, щоб вона виконувалася довічно. Отже, якщо не передбачити її примусове завершення, то програма фактично зациклиться. Завдяки застосування функції pthread_cancel() цього не відбувається. Результат виконання даної програми наведено на рис. 2.4.

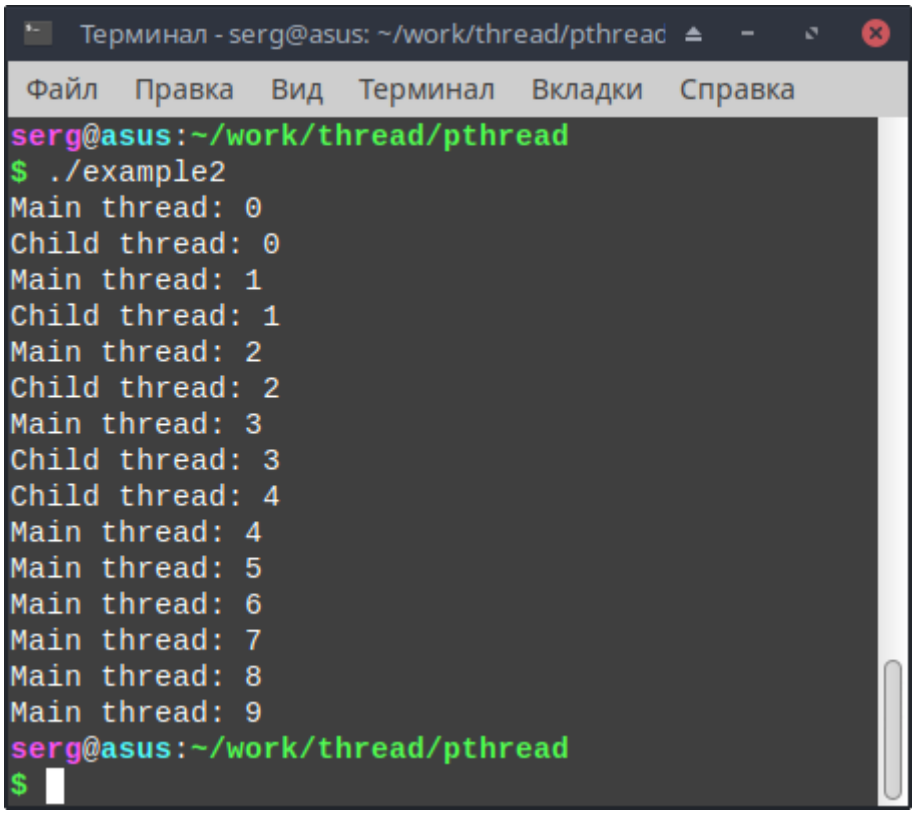


Рис. 2.4 – Приклад роботи програми example2 з примусовим завершенням потоку

Слід враховувати, що можливість примусового завершення потоку може бути скасована за допомогою функції pthread_setcancelstate().

int pthread_setcancelstate(int state, int *oldstate);

У разі успіху функція повертає значення 0, стан відміни поточного потоку встановлюється в нове значення **state**, а попереднє зберігається в параметрі **oldstate**. В якості значення перший параметр може приймати одне з наступних значень: **PTHREAD_CANCEL_ENABLE** (потік може бути примусово зупинений) або **PTHREAD_CANCEL_DISABLE** (навпаки). При успішному завершенні дана функція повертає 0.

Змінимо потокову функцію попереднього прикладу наступним чином.

```
void *thread_code(void *arg)
{
	int i = 0;
	// Блокування можливості примусового завершення потоку
	pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
	while (1)
	{
		printf("Child thread: %d\n",  i++);
		sleep(1);
	}
	return NULL;
}
```

Компіляція та запуск цієї програми приведе до наступного результату (рис. 2.5). З наведеного скріншоту видно, що спроба примусового завершення дочірнього потоку в головному виявилася невдалою, і він залишався активним

до того моменту, поки весь процес не був примусово завершений комбінацією клавіш <Ctrl>+<C>.

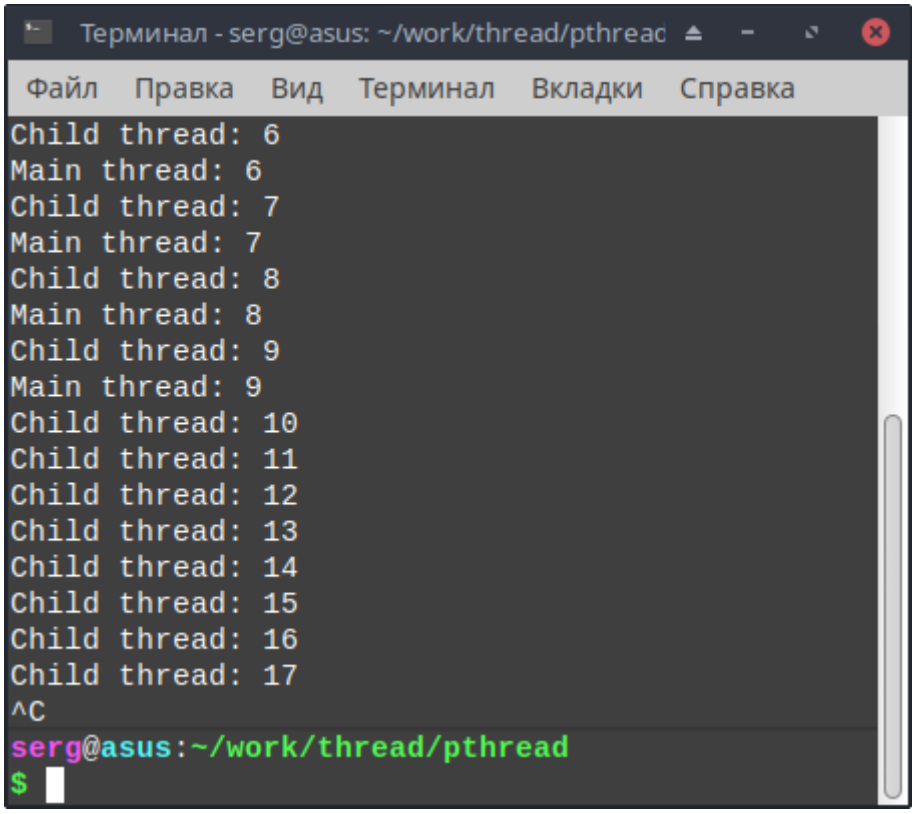


Рис. 2.5 – Робота програми example2 з заблокованою можливістю примусового завершення потоку

2.2.4 Передача параметрів у потоки

Розглянемо способи передачі параметрів у потокові функції. Нехай, наприклад, необхідно розрахувати суму факторіалів чисел, які зберігаються у масиві цілих чисел заданої довжини. Ця задача із застосуванням паралельної техніки «розділяй і володарюй» може бути легко розбита на задану кількість підзадач, які розв’язуються паралельно і незалежно одна від одної, а отримані результати наприкінці сумуються.

Розглянемо наступну структуру даних, в якій зберігаються параметри та результати роботи потоку.

```
// Структура для передачі параметрів у функцію потоку
typedef struct
{
	int len;	// Довжина масиву
	int *arr;	// Покажчик на масив
	int res;	// Результат
}
thread_params;
```

Тут поле **len** містить довжину масиву чисел, що підлягає обробці, **arr** – покажчик на нього, а **res** – призначене для зберігання результату, який буде отримано при роботі потокової функції, яку безпосередньо можна реалізувати таким чином.

```
// Потокова функція
void *thread_func(void *args)
{
	int len = ((thread_params*)args)->len,
	    *arr = ((thread_params*)args)->arr;

	for (int i = 0; i < len; i++)
		((thread_params*)args)->res += fact(arr[i]);
	return NULL;
}
```

В цьому коді безтиповий покажчик **args** використовується для передачі у потокову функцію інформації про оброблювані дані (адресу початку чергового блоку чисел та його довжину, а також результату обчислень). Функція fact() призначена для обчислення факторіалу і може бути із застосуванням рекурсії реалізована наступним чином.

```
// Рекурсивне обчислення факторіалу
int fact(int arg)
{
	return (arg == 0 || arg == 1) ? 1 : arg * fact(arg - 1);
}
```

Тоді решту програми можна буде реалізувати так (вихідний файл example3.c).

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Опис thread_params, функції підрахунку факторіалу тощо
// ...
// Функція створення потоку
void thread_create(pthread_t *tid, const pthread_attr_t *attr,
			void *(*start_routine) (void *), void *arg)
{
	if (pthread_create(tid, attr, start_routine, arg) != 0)
	{
		perror("pthread_create()");
```

```
            exit(1);
        }
}
int main(int argc, char **argv)
{
        int *arr = NULL, res = 0, step, num_thread, arr_len;
        pthread_t *tid = NULL;
        thread_params *tp = NULL;
        clock_t begin = clock();

        // Перевірка на кількість введених параметрів
        if (argc != 3)
        {
            printf("Too few arguments!\n");
            return 1;
        }

        // Обробка аргументів
        if ((num_thread = atoi(argv[1])) == 0)
        {
            printf("Wrong number of threads!\n");
            return 1;
        }
        if ((arr_len = atoi(argv[2])) == 0)
        {
            printf("Wrong length of data array!\n");
            return 1;
        }
```

```
// Виділення пам’яті для зберігання масивів
// ... чисел, що підлягають обробці
if ((arr = malloc(arr_len * sizeof(int))) == NULL)
{
        printf("Not enough memory!\n");
        return 1;
}
// ... ідентифікаторів потоків
if ((tid = malloc(num_thread * sizeof(pthread_t))) == NULL)
{
        printf("Not enough memory!\n");
        return 1;
}
// ... потокових параметрів
if ((tp = malloc(num_thread * sizeof(thread_params))) == NULL)
{
        printf("Not enough memory!\n");
        return 1;
}
// Ініціалізація даних псевдовипадковими числами
// в діапазоні від 0 до 4
for (int i = 0; i < arr_len; i++)
        arr[i] = rand() % 5;
// Розрахунок кількості чисел, що оброблюються кожним потоком
step = arr_len / num_thread;
// Створення та запуск потоків
for (int i = 0; i < num_thread; i++)
{
        // Підготовка параметрів
```

```
            tp[i].res = 0;
            tp[i].arr = arr + i * step;
            tp[i].len = (i == num_thread - 1) ? arr_len - i * step: step;
            // Створення потоку
            thread_create(&tid[i], NULL, thread_func, &tp[i]);
        }
        // Очікування результатів роботи всіх потоків
        for (int i = 0; i < num_thread; i++)
        {
            pthread_join(tid[i], NULL);
            res += tp[i].res;
        }

        // Звільнення пам’яті
        free(arr);
        free(tp);
        free(tid);
        // Виведення результату
        printf("Result for %d thread(s) and array len %d: %d\n", num_thread,
            arr_len, res);
        printf("Working time: %lf\n",
            (double)(clock() - begin) / CLOCKS_PER_SEC);
        return 0;
    }
```

Компіляція та запуск цієї програми приведуть до наступного результату (рис. 2.6). З наведеного скріншоту видно, що час виконання багатопотокових програм не завжди зменшується при збільшенні кількості потоків, що можна

пояснити як зростанням накладних витрат ОС на контекстне перемикання між потоками, так і багатьма іншими причинами.

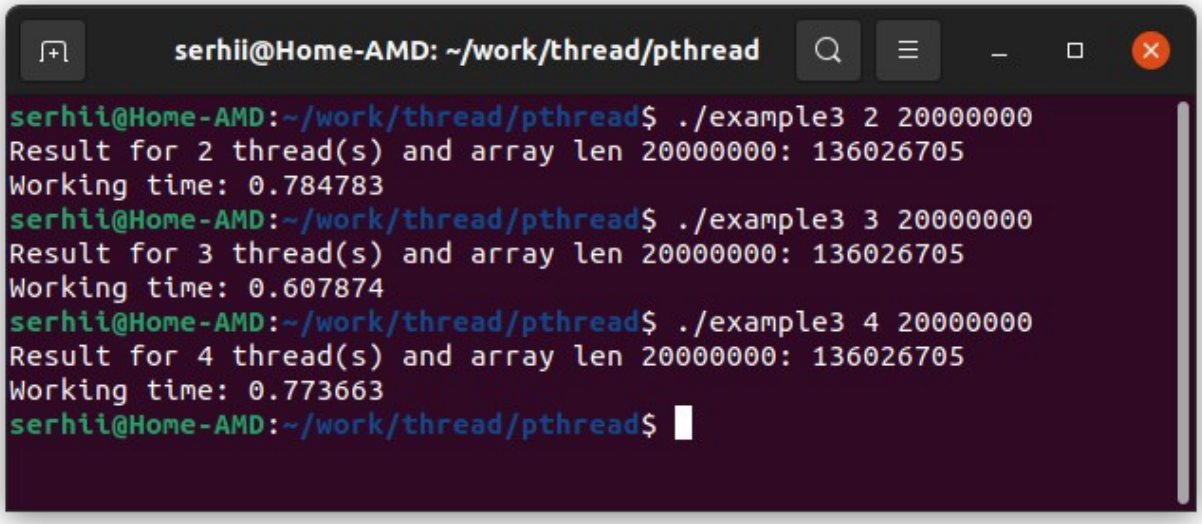


Рис. 2.6 – Результати роботи та час виконання підрахунку суми факторіалів у багатопотоковій програмі example3

2.2.5 Застосування м’ютексів

Розглянемо приклад невірно спроєктованої програми, який демонструє виникнення гонитви за даними. Нехай потрібно обчислити суму заданої кількості членів ряду $\sum_{i=0}^{N} \frac{1}{1+i^3}$. Для цього напишемо наступну програму (example4.c).

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <pthread.h>

// Глобальна змінна для збереження суми ряду
double Sum = 0;

// Структура для передачі параметрів у функцію потоку
typedef struct
{
	int begin;	// Початок діапазону розрахунку
	int end;	// Кінець...
}
thread_params;

// Потокова функція
void *thread_func(void *args)
{
	double begin = (double)((thread_params*)args)->begin,
		end = (double)((thread_params*)args)->end;

	// Обчислення суми членів заданого діапазону ряду
	for (double i = begin; i < end; i++)
		Sum += (1.0 / (1.0 + i * i * i));
	return NULL;
}

// Функція створення потоку
// ...

int main(int argc, char **argv)
```

```
{
	int step, num_thread, max_iter;
	pthread_t *tid = NULL;
	thread_params *tp = NULL;
	clock_t begin = clock();

	// Обробка аргументів: числа потоків і кількості члені ряду
	if (argc != 3)
	{
		printf("Too few arguments!\n");
		return 1;
	}
	if ((num_thread = atoi(argv[1])) == 0)
	{
		printf("Wrong number of threads!\n");
		return 1;
	}
	if ((max_iter = atoi(argv[2])) == 0)
	{
		printf("Wrong number of iteration!\n");
		return 1;
	}
	if ((tid = malloc(num_thread * sizeof(pthread_t))) == NULL)
	{
		printf("Not enough memory!\n");
		return 1;
	}
	if ((tp = malloc(num_thread * sizeof(thread_params))) == NULL)
	{
```

```
            printf("Not enough memory!\n");
            return 1;
        }

        step = max_iter / num_thread;
        // Створення та запуск потоків
        for (int i = 0; i < num_thread; i++)
        {
            tp[i].begin = i * step;
            tp[i].end = i == num_thread - 1 ? max_iter : (i + 1) * step;
            thread_create(&tid[i], NULL, thread_func, &tp[i]);
        }
        // Очікування завершення потоків
        for (int i = 0; i < num_thread; i++)
            pthread_join(tid[i], NULL);

        // Виведення результату
        printf("Sum of series for %d thread(s) and number of iteration %d:
            %lf\n", num_thread, max_iter, Sum);
        printf("Working time: %lf\n",
            (double)(clock() - begin) / CLOCKS_PER_SEC);
        return 0;
    }
```

В цій програмі для збереження результату розрахунку використовується глобальна змінна **Sum**, до якої кожен потік безпосередньо додає розраховані значення члені ряду.

Компіляція та запуск програми при різній кількості потоків дає наступний результат (рис. 2.7).

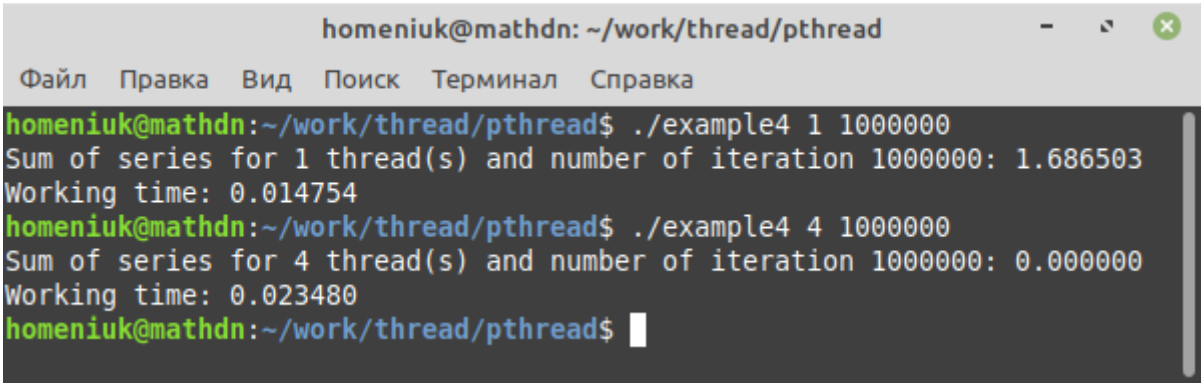


Рис. 2.7 – Приклад гонитви за даними в програмі example4

З наведеного скріншоту видно, що запуск цієї програми при різній кількості потоків може давати різні результати, що пояснюється гонитвою за даними (змінною **Sum**) в критичній області, яку складає наступна частина потокової функції.

```
// ...
Sum += (1.0 / (1.0 + i * i * i));
// ...
```

Як вже зазначалося, для вирішення цієї проблеми застосовуються спеціальні об’єкти – м’ютекси. В P-thread для для роботи з м’ютексами застосовуються дві основні функції pthread_mutex_lock() і pthread_mutex_unlock().

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Перша функція блокує м’ютекс, покажчик на який вона отримує в якості параметра **mutex**, а друга – виконує розблокування. В разі успіху обидві функції повертають нульове значення.

Таким чином, щоб запобігти гонитві даних, яка виникає у попередньому прикладі, в нього необхідно внести, наприклад, такі зміни.

```
// ...
// Глобальний м’ютекс
pthread_mutex_t mutex;

// Потокова функція
void *thread_func(void *args)
{
        int begin = ((thread_params*)args)->begin,
            end = ((thread_params*)args)->end;
        for (int i = begin; i < end; i++)
        {
                pthread_mutex_lock(&mutex);
                Sum += (1.0 / (1.0 + (double)(i * i * i)));
                pthread_mutex_unlock(&mutex);
        }
        return NULL;
}
```

Компіляція та запуск цієї програми дасть наступний результат (рис. 2.8).

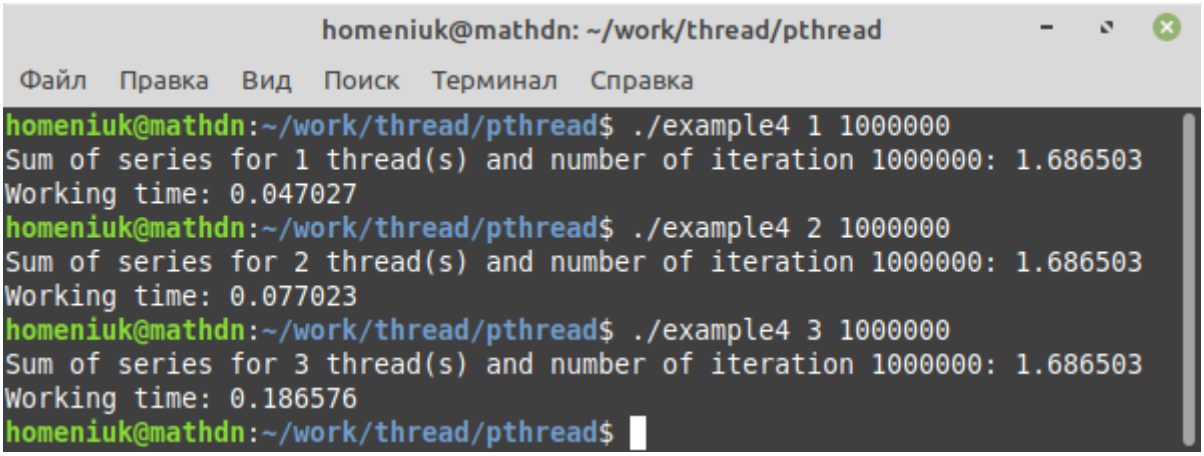


Рис. 2.8 – Результат роботи програми example4 із застосуванням м’ютексу

Тут слід зазначити, що така реалізація цієї програми є дуже неефективною, оскільки блокування м’ютексу відбувається в циклі, що призводить до великих витрат часу на очікування іншими потоками своєї черги. Істотно (на декілька порядків) пришвидшити роботу цієї програми можна, дещо переробив потокову функцію.

```
void *thread_func(void *args)
{
    int begin = ((thread_params*)args)->begin,
        end = ((thread_params*)args)->end;
    double sum = 0; // Локальна потокова змінна

    for (int i = begin; i < end; i++)
        sum += (1.0 / (1.0 + (double)(i * i * i)));
    // Критична область
    pthread_mutex_lock(&mutex);
    Sum += sum;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

В такій редакції локальна змінна **sum** є унікальною для кожного потоку (знаходиться в його стеці), а доступ до глобальної **Sum** (а, отже, і використання м’ютексу) відбувається лише один раз в процесі роботи кожної потокової функції.

2.2.6 Від’єднання потоків

На практиці при проєктуванні багатопотокових програм інколи виникає необхідність від’єднання дочірнього потоку від батьківського. В цьому випадку дочірній потік виконується асинхронно по відношенню до батьківського і останній вже не може отримати статусу завершення дочірнього потоку.

Для від’єднання потоку використовується функція pthread_detach(), яка має наступний шаблон.

int pthread_detach(pthread_t thread);

В разі успіху функція повертає нульове значення. Параметр **thread** визначає дескриптор потоку, який від’єднується.

Розглянемо приклад (вихідний файл example5.c), коли головний потік завершується раніше, ніж його дочірній, при цьому останній продовжує свою роботу.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Потокова функція
void *thread_func(void *args)
{
	for (int i = 0; i < 10; i++)
	{
		printf("Thread output %d\n", i);
	}
	return NULL;
```

```
}

// Функція створення потоку
// ...

int main()
{
	pthread_t tid;

	// Створення потоку
	thread_create(&tid, NULL, thread_func, NULL);

	// Від'єднання потоку
	pthread_detach(tid);

	// Завершення головного потоку
	pthread_exit(0);
	// Цей код ніколи не виконається
	printf("Stopped main thread\n");
	return 0;
}
```

Компіляція та запуск цієї програми дадуть наступні результати (рис. 2.9). З цього скріншоту можна зробити висновок, що якщо завершити головний потік програми (тіло функції main()) не оператором «return» або функцією завершення процесу exit(), а функцією завершення потоку pthread_exit(), то багатопотоковий процес не завершує негайно свою роботу, оскільки в ньому ще працює від’єднаний потік.

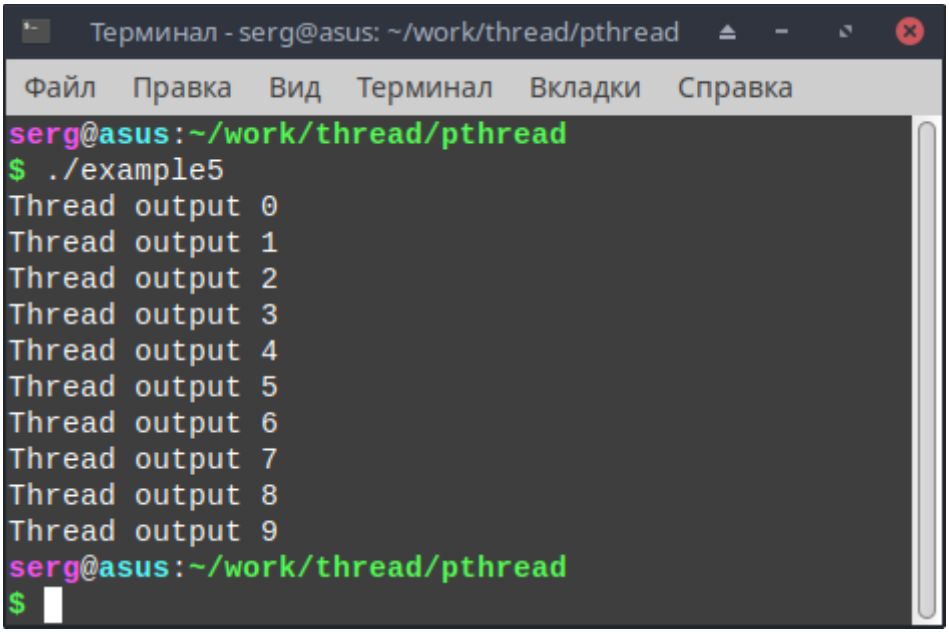


Рис. 2.9 – Результат роботи програми example5

Розглянемо ще один приклад. Нехай потрібно фіксувати в журналі всі числові значення, що вводяться користувачем. Спроєктуємо цю програму таким чином, щоб у головному потоці оброблювалися введені користувачем числові значення, а у дочірньому – вівся відповідний журнал. Ясно, що в такій постановці, потік ведення журналу може бути від’єднаним від головного, оскільки вони виконуються абсолютно асинхронно по відношенню один від одного.

Тоді цю програму можна реалізувати, наприклад, так (example6.c).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

// Потокова функція ведення журналу
void *thread_func(void *args)
{
```

```
	int value = 0, old_value = 0;
	FILE *out;

	if ((out = fopen("value.log", "w")) == NULL)
	{
		perror("fopen()");
		exit(1);
	}

	fprintf(out, "Logging started\n");
	// 3 однакових значень, що йдуть поспіль, записується лише одне
	do
	{
		if (value != old_value)
			fprintf(out, "%d\n", value);
		old_value = value;
		value = *(int*)args;
	}
	while (value != -1);

	fprintf(out, "Logging stopped\n");
	fclose(out);
	return NULL;
}

// Функція створення потоку
// ...

int main()
```

```
{
	int value = 0;
	pthread_t tid;

	// Створення потоку
	thread_create(&tid, NULL, thread_func, &value);

	// Від'єднання потоку
	pthread_detach(tid);

	// Робота головного потоку
	while (1)
	{
		printf("Enter int value (-1 for exit): ");
		scanf("%d", &value);
		if (value == -1)
			break;
		// doSomething(value);
	}
	// Очікування запису файлу
	sleep(0.5);
	return 0;
}
```

Результати роботи цієї програми наведені на рис. 2.10.

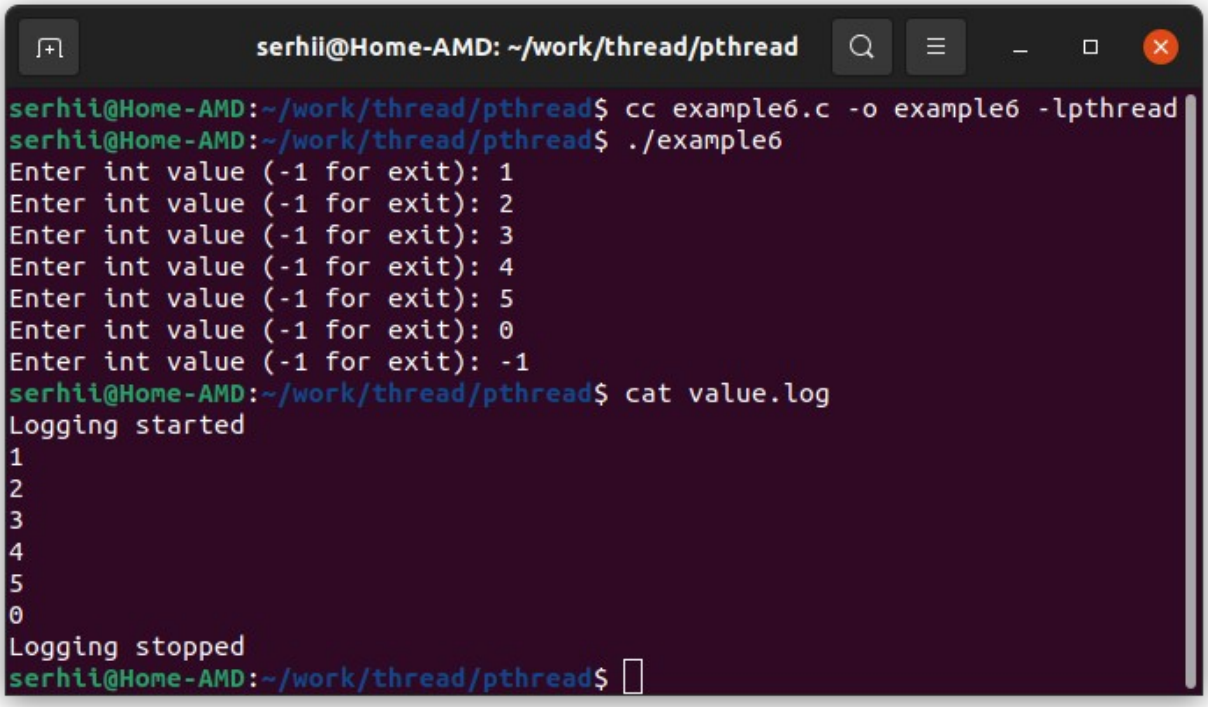


Рис. 2.10 – Результат роботи програми example6.c

Тут слід прокоментувати наступне. Обмін інформацією між потоками фактично здійснюється через змінну **value** головного потоку (тобто через пам’ять). Виклик функції sleep() потрібен, щоб дати можливість «скинути» файлові буфери на диск, інакше може виникнути ситуація, коли не всі дані будуть записані у файл.

Питання для самоперевірки

1. Нащадком якої операційної системи є Linux?
2. Що таке API?
3. Яка різниця між стандартами POSIX, SUS та LSB?
4. Яку назву мають потоки в стандарті POSIX?

5. В якому заголовному файлі міститься опис функцій бібліотеки P-thread?

6. На які групи можна умовно поділити всі функції цієї бібліотеки?

7. Які спеціальні типи даних бібліотеки P-thread ви знаєте?

8. За допомогою якої функції створюється потік виконання в P-thread? Що ця функція повертає?

9. Яким чином реалізується синхронізація потоків в P-thread?

10. За допомогою якої функції можна примусово завершити потік P-thread?

11. Яким чином здійснюється передача параметрів у потік P-thread?

12. Як здійснюється компіляція програм, що застосовують потоки P-thread?

13. Яким чином створюються м’ютекси в бібліотеці P-thread?

14. Яка принципова різниця між функціями pthread_join() та pthread_detach()?

15. Яка різниця між функціями pthread_exit() та exit()?

Вправи

1. Напишіть багатопотокову програму обчислення суми заданої кількості членів ряду $\sum_{i=0}^{N} \frac{1}{1+i^5}$. Визначте час виконання програми. Побудуйте графік залежності часу виконання від кількості застосованих обчислювальних потоків.

2. Напишіть програму, що реалізує паралельне перемноження матриць. Побудуйте графік залежності часу її виконання від кількості вживаних потоків.

3. Напишіть програму, що обчислює сумарний розмір всіх файлів, які розташовані в заданому каталозі.

ТЕМА 3 СТАНДАРТНІ ЗАСОБИ РОЗРОБКИ БАГАТОПОТОКОВИХ ПРОГРАМ МОВИ C++

3.1 Клас std::thread

До появи стандарту C++11 для розробки багатопотокових програм на мові програмування C++ необхідно було застосовувати сторонні бібліотеки, такі, наприклад, як P-thread, Qt або Boost. Проте, починаючи з 2011 року, це можна робити лише із застосуванням стандартної бібліотеки STL (Standard Template Library), оскільки до її складу включено цілу низку спеціальних класів для роботи з потоками і, в першу чергу, **std::thread**.

3.1.1 Запуск потоку

Як і у мові програмування C, програма на C++ складається мінімум з одного потоку, тіло якого утворюється кодом функції main(). Для запуску додаткових потоків, що працюють паралельно, необхідно створити об’єкт класу std::thread, який задекларовано в заголовному файлі thread (стандартні заголовні файли мови C++ не мають розширення .h, яке застосовується для аналогічних файлів у мові C), і передати в нього в якості параметра покажчик на потокову функцію (при необхідності можна додати й інші необов’язкові параметри, які призначені для передачі аргументів у потокову функцію). Зробити це можна, наприклад, таким чином.

```
std::thread thr(thread_func, arg1, arg2, arg3);
```

Тут створюється об’єкт **thr** класу std::thread, який буде виконувати потокову функцію **thread_func**, яка, в свою чергу, отримає три параметри **arg1**, **arg2** і **arg3**.

Розглянемо наступний простий приклад (cpp_example0.cpp).

```
#include <iostream>
#include <thread>

// Потокова функція
void thread_func(int no)
{
    std::cout << "This is child thread: " << no << std::endl;
}

int main()
{
    // Створення і запуск дочірнього потоку з параметром 1
    std::thread thr(thread_func, 1);

    std::cout << "This is main thread" << std::endl;
    return 0;
}
```

Компіляція і запуск цієї програми призведуть до наступного результату (рис. 3.1). Аналіз цього скріншоту показує, що, по-перше, програма була завершена аварійно, а, по-друге, інформація, що була виведена потоками виконання (thread) у стандартний потік виводу (stream), так би мовити перемішалася.

Спочатку будемо розбиратися з першою проблемою. Вона зазвичай виникає, коли потоки не синхронізуються. В даному випадку батьківський (головний) потік закінчив свою роботу раніше за дочірній (на створення потоку ОС потрібен певний час), що призвело до виклику деструктора класу std::thread для об’єкта thr, якій ще працював. А це, в свою чергу, привело до некоректної роботи з пам’яттю.

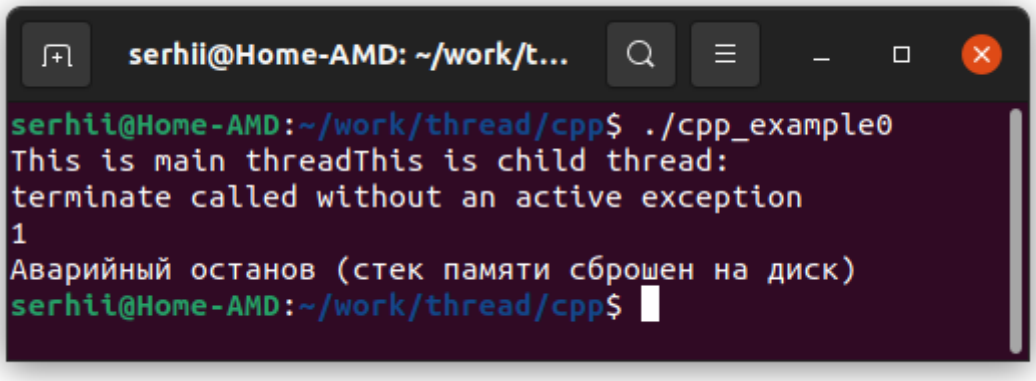


Рис. 3.1 – Результат роботи програми cpp_example0

Для запобігання цієї проблеми в класі std::thread є спеціальні методи синхронізації join() та detach(), які є аналогами відповідних функцій з P-thread.

3.1.2 Синхронізація потоків

Розглянемо приклад реалізації синхронізації виконання потоків шляхом блокування батьківського до завершення роботи дочірніх (cpp_example1.cpp).

```
#include <iostream>
#include <thread>
```

```
using namespace std;

// Потокова функція
void thread_work(int no)
{
    cout << "This is child thread: " << no << '\n';
}

int main()
{
    const int num_thread = 5;
    thread *thr[num_thread];

    // Створення потоків
    for (auto i = 0; i < num_thread; i++)
       thr[i] = new thread(thread_work, i + 1);
    // Робота головного потоку
    cout << "This is main thread\n";
    // Очікування завершення потоків (синхронізація)
    for (auto i = 0; i < num_thread; i++)
       thr[i]->join();
    // Звільнення пам’яті
    for (auto i = 0; i < num_thread; i++)
       delete thr[i];
    return 0;
}
```

Компіляція і запуск цієї програми дадуть результат, який зображено на (рис. 3.2). Тут слід прокоментувати наступне. Спочатку в циклі із застосуванням

оператора «new» динамічно створюються п’ять об’єктів класу std::thread. В конструктор кожного об’єкту передаються два аргументи: покажчик на потокову функцію і її параметр (номер потоку). Як тільки потоковий об’єкт створено, він негайно починає виконуватися. Для запобігання достроковому завершенню програми виконується блокування роботи головного потоку із застосуванням методу join() класу std::thread. Ну, і нарешті після завершення роботи потоків за допомогою оператора «delete» динамічно створені об’єкти знищуються (зазвичай при завершенні роботи процесу вся його пам’ять звільнюється ОС автоматично, проте правила хорошого тону вимагають від програміста звільнення пам’яті вручну[2]).

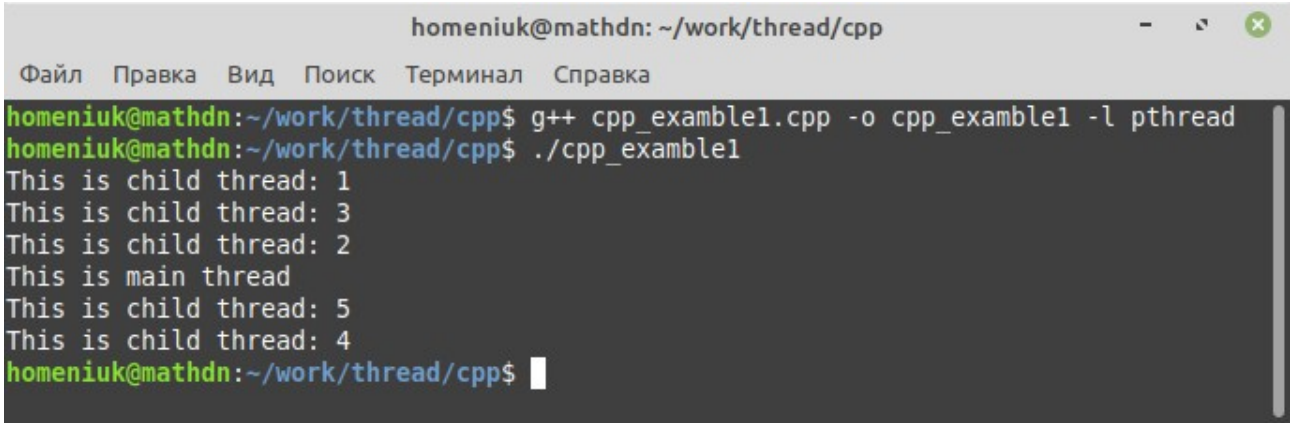


Рис. 3.2 – Результат роботи програми cpp_example1

Для від’єднання потоку у класі std::thread є спеціальний метод detach(). Як і у бібліотеці P-thread, він застосовується у тому випадку, коли головний потік не буде очікувати завершення його роботи. Після від’єднання потоку отримати результат його роботи вже неможливо.

Розглянемо наступний приклад (cpp_example2.cpp).

```
#include <iostream>
```

2 Оскільки в мові C++ немає вбудованого механізму збирання сміття (garbage collection), для автоматизації цього процесу можна використовувати розумні покажчики (smart pointer), які з’явилися, починаючи зі стандарту C++11.

```
#include <thread>

using namespace std;

// Потокова функція
void thread_work(void)
{
    for (auto i = 0; i < 5; i++)
    {
        cout << "Detached child thread iteration: " << i << '\n';
        // Затримка виконання на 0.5 с
        this_thread::sleep_for(0.5s);
    }
}

int main()
{
    // Створення дочірнього потоку без параметрів
    thread thr(thread_work);
    // Від'єднання дочірнього потоку
    thr.detach();
    // Робота головного потоку
    for (auto i = 0; i < 3; i++)
    {
        cout << "Main thread iteration: " << i << '\n';
        this_thread::sleep_for(0.5s);
    }
    return 0;
}
```

Результат виконання цієї програми буде мати такий вигляд (рис. 3.3). З наведеного скріншоту видно, що після завершення роботи головного потоку робота від’єднаного потоку була автоматично завершена (він не виконав всі свої дії). Це відбулося тому, що при завершенні функції main() було автоматично викликано деструктор для об’єкту **thr**, що призвело до завершення роботи від’єднаного потоку.

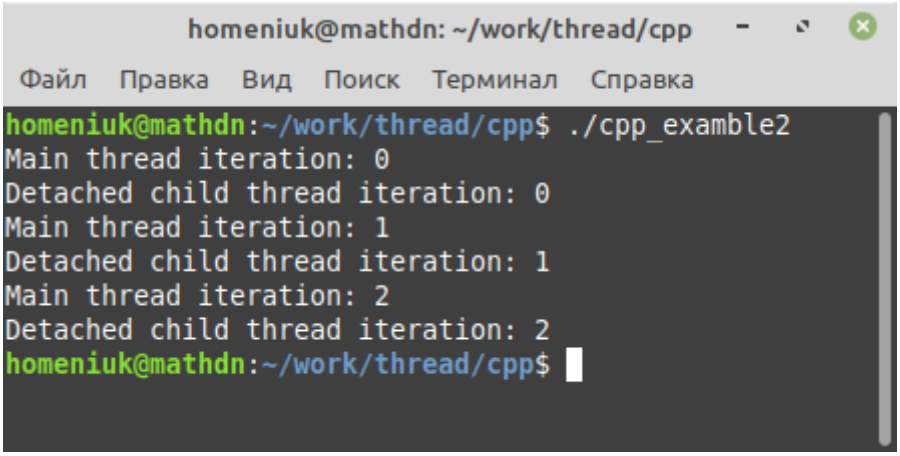


Рис. 3.3 – Результат роботи програми cpp_example2

Також слід звернути увагу на простір імен **std::this_thread**, в якому реалізовано зокрема функцію sleep_for(), що призупиняє роботу поточного потоку на задану кількість секунд.

Розглянемо ще один приклад застосування від’єднаних потоків. Нехай потрібно створити журнал, в якому фіксуються всі введені користувачем команди. Цю програму можна спроєктувати таким чином, щоб у головному потоці було реалізовано введення користувачем команд і їх обробка, а у від’єднаному дочірньому потоці здійснювалося ведення відповідного журналу. Тоді цю програму можна реалізувати таким чином (cpp_example3.cpp).

```
#include <iostream>
#include <fstream>
```

```
#include <thread>

using namespace std;

// Потокова функція ведення журналу
void thread_work(string &command, bool &need_write)
{
    ofstream log("log.txt");

    if (!log.is_open())
    {
       cerr << "Unable open log file!" << endl;
       exit(1);
    }
    while (1)
    {
       if (command == "quit")
              break;
       if (need_write)
       {
              log << command << '\n';
              need_write = false;
       }
    }
    log.close();
}

int main()
{
```

```
    string command;
    bool is_new_command = false;
    // Створення дочірнього потоку
    thread thr(thread_work, ref(command), ref(is_new_command));

    // Від'єднання дочірнього потоку
    thr.detach();
    // Запит команди та та її обробка
    while (1)
    {
        cout << "Enter query ('quit' for exit): ";
        cin >> command;
        if (command == "quit")
                break;
        is_new_command = true;
        // Обробка команди
        // ...
    }
    return 0;
}
```

Результат роботи цієї програми наведено на рис. 3.4. В наведеній вище програмі слід прокоментувати наступне. В потокову функцію передаються два параметри: посилання на об’єкт класу string, що містить введену користувачем команду, і посилання на логічну змінну, яка зберігає ознаку необхідності запису поточної команди в журналі. Для передачі параметра-посилання у конструктор класу std::thread необхідно використовувати спеціальну функцію-обгортку **std::ref()**.

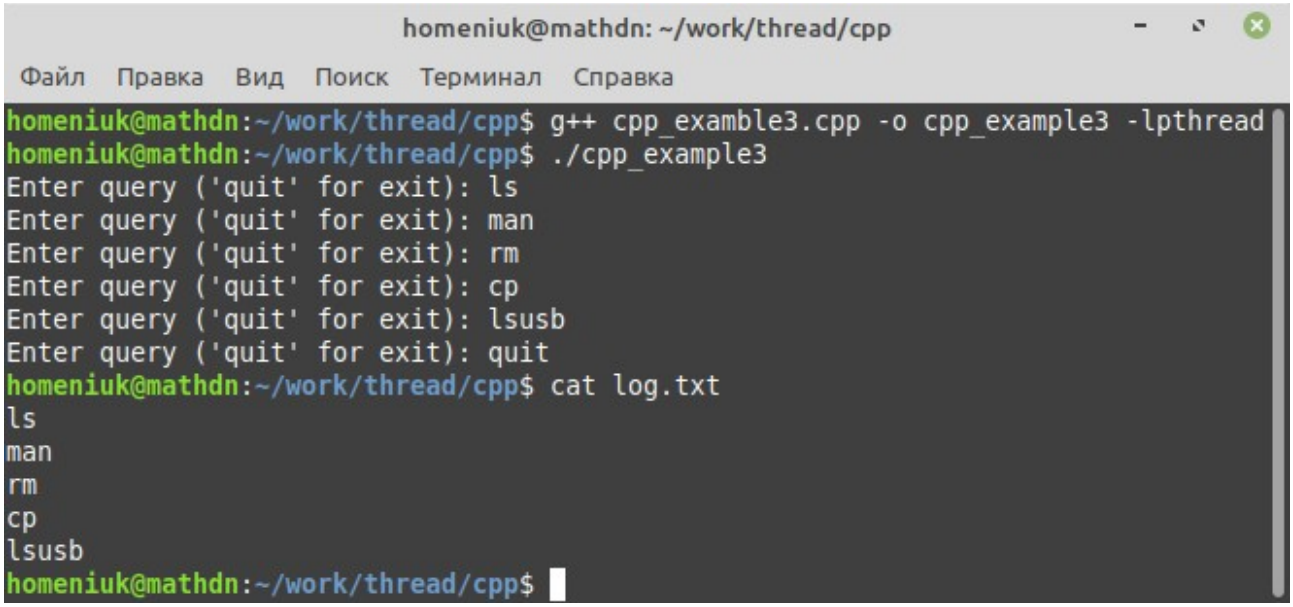


Рис. 3.4 – Результат роботи програми cpp_example3

3.1.3 Запуск потоків у класах

Розглянемо створення потоку, тіло якого складає функція-метод певного класу. Розглянемо простий клас ThreadWork, в якому один з його приватних методів thread_func() виводить свій параметр цілого типу у стандартний потік (stream) виводу. Тоді, якщо цей метод потрібно запустити як потік виконання з іншого методу цього класу, необхідно створити об’єкт класу std::thread наступним чином.

thread thr = thread(&ThreadWork::thread_func, this, i);

Тут перший параметр визначає покажчик на метод, який складає тіло потоку, другий – покажчик на поточний об’єкт, а третій – аргумент потокової функції.

Розглянемо повну реалізацію цієї програми (cpp_example4.cpp).

```
#include <iostream>
#include <thread>
#include <vector>
#include <algorithm>

using namespace std;

class ThreadWork
{
private:
    int num_thread = 1;
    // Потокова функція
    void thread_func(int no)
    {
       cout << "Thread no: " << no << '\n';
    }
public:
    ThreadWork(void) {}
    ThreadWork(int num) : num_thread(num) {}
    ~ThreadWork() = default;
    void set_num_thread(int num_thread)
    {
       this->num_thread = num_thread;
    }
    // Запуск потокової роботи
    void run(void)
    {
       vector<thread> thr;
```

```
        // Створення і запуск потоків
        for (auto i = 0; i < num_thread; i++)
            thr.push_back(thread(&ThreadWork::thread_func, this, i));
        // Синхронізація ...
        for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });
    }
};

int main()
{
    ThreadWork tw(10);

    // Запуск багатопотокової обробки
    tw.run();
    return 0;
}
```

Тут метод класу ThreadWork run() реалізує створення заданої кількості потоків, які зберігаються у стандартному векторі std::vector<std::thread>. Після їх створення і запуску із застосуванням функції std::for_each (описана у заголовному файлі algorithm) виконується їх синхронізація з головним потоком.

Компіляція та запуск цієї програми дадуть наступний результат (рис. 3.5).

Розглянемо ще один спосіб створення потоків. Якщо в класі перезавантажено оператор виклику функції «()», то створити потік можна так, як це показано нижче (cpp_example5.cpp).

```
#include <iostream>
#include <thread>
#include <vector>
```

#include <algorithm>

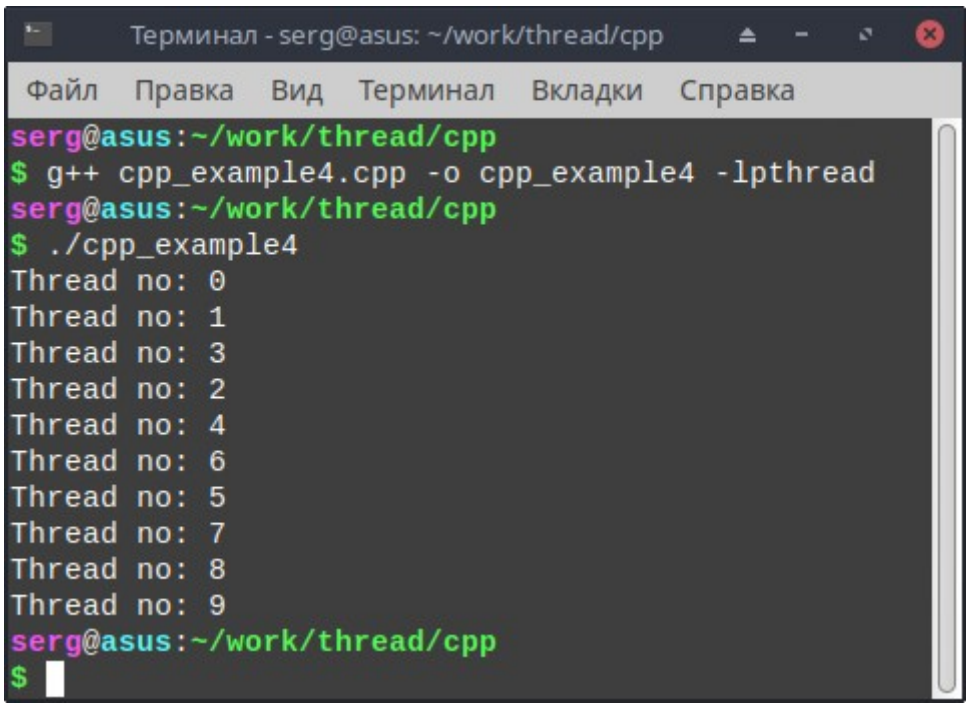


Рис. 3.5 – Результат роботи програми cpp_example4

```
using namespace std;
// Клас з перевантаженим оператором виклику функції
class DoWork
{
public:
    DoWork(void) {}
    ~DoWork() = default;
    // Оператор виклику функції
    void operator () (int no)
    {
       cout << "Thread no: " << no << endl;
    }
};
```

```
int main()
{
    int num_thread = 10;
    vector<thread> thr(num_thread);
    DoWork processor;
    // Створення і запуск потоків
    for (auto i = 0; i < num_thread; i++)
       thr[i] = thread(processor, i);
    // Синхронізація ...
    for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });
    return 0;
}
```

Трансляція та запуск цієї програми дадуть наступний результат (рис. 3.6).

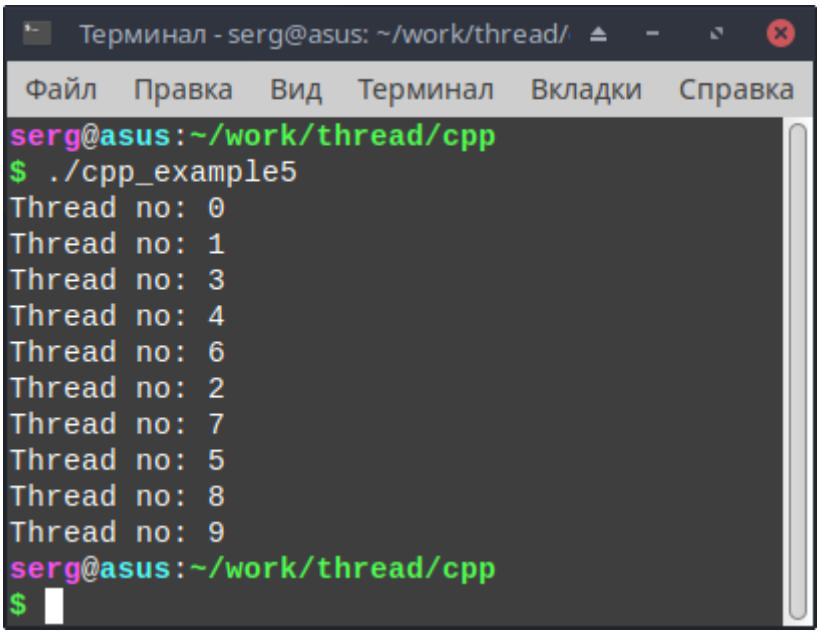


Рис. 3.6 – Результат виконання програми cpp_example5

Таким чином, якщо в класі перезавантажено оператор виклику функції, то в конструктор класу std::thread достатньо передати в якості покажчика потокової функції передати об’єкт цього класу.

3.2 Клас std::mutex

В бібліотеці STL для інкапсуляції поняття м’ютексу реалізовано клас std::mutex. Для його використання до програми необхідно додатки заголовний файл mutex. Базовий конструктор цього класу не вимагає параметрів, тому створити м’ютекс можна, наприклад, так.

```
#include <mutex>

// ...
std::mutex my_mutex;
// ...
```

Клас std::mutex містить декілька методів, найбільш часто вживаними серед яких є lock() і unlock().

```
// ...
my_mutex.lock(); // Захоплення м’ютексу
// Критична область
// ...
my_mutex.unlock(); // Звільнення м’ютексу
```

Розглянемо приклад багатопотокової реалізації програми підрахунку суми ряду $\sum_{i=0}^{N} \frac{1}{1+i^3}$ (див. пункт 2.2.5). На C++ програмно реалізувати цей алгоритм можна таким чином (cpp_example6.cpp).

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <functional>
#include <algorithm>

using namespace std;

// Глобальний м’ютекс
mutex mtx;

// Клас, що реалізує підрахунок суми заданого діапазону членів ряду
struct CalcSeries
{
    void operator () (int begin, int end, function<double(double)> fun,
                                          double &sum)
    {
       double s = 0;

       for (auto i = begin; i < end; i++)
             s += fun(double(i));
       // Критична область
       mtx.lock();
```

```
        sum += s;
        mtx.unlock();
    }
};

int main(int argc, char **argv)
{
    int num_thread,
        num_iter,
        step;
    double sum = 0;
    vector<thread> thr;
    CalcSeries cs;

    if (argc != 3)
    {
        cerr << "Too few arguments!" << endl;
        return 1;
    }
    if ((num_thread = atoi(argv[1])) <= 0)
    {
        cerr << "Wrong number of threads!" << endl;
        return 1;
    }
    if ((num_iter = atoi(argv[2])) <= 0)
    {
        cerr << "Wrong number of iteration!" << endl;
        return 1;
    }
```

```
    // Створення та запуск потоків
    step = num_iter / num_thread;
    for (auto i = 0; i < num_thread; i++)
       thr.push_back(thread(cs, i * step, (i == num_thread - 1) ?
             num_iter : (i + 1) * step,
             [](double x) -> double { return (1 / (1 + (x * x * x))); },
             ref(sum)));
    // Блокування головного потоку
    for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });
    // Виведення результатів
    cout << "Sum of series for " << num_thread << " thread(s) : "
        << sum << endl;
    return 0;
}
```

Компіляція та запуск цієї програми дадуть наступний результат (рис. 3.7). В цій програмі слід прокоментувати наступні моменти. По-перше, в мові програмування C++ структура (struct) відрізняється від свого аналога в мові C. В C++ структура – це тип класу, всі властивості та методи якого є відкритими. По-друге, для передачі функціональних об’єктів у якості параметрів, починаючи зі стандарту C++11, можна (і рекомендується) застосовувати спеціальний клас **std::function**, який є обгорткою для різних функціональних типів (функцій, лямбда-виразів тощо). По-третє, при створенні потоку в якості параметру в об’єкт класу CalcSeries передається лябда-функція, що розраховує заданий член числового ряду, суму якого обчислює потік.

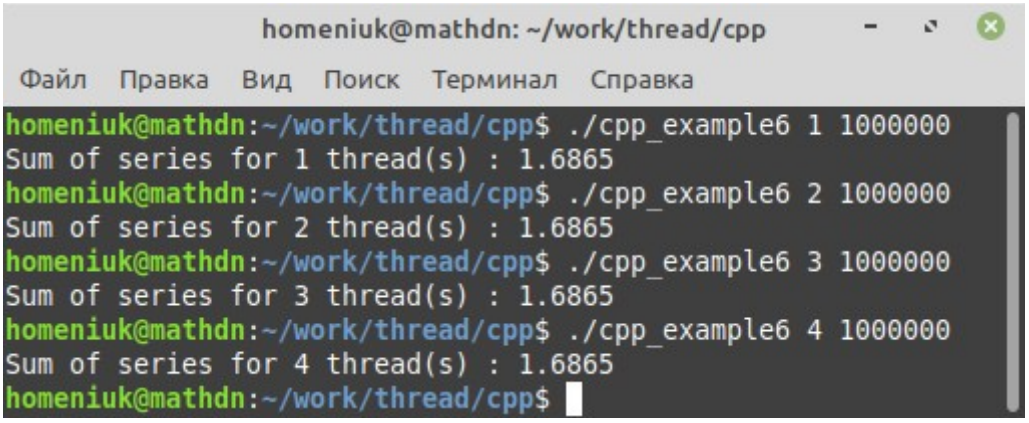


Рис. 3.7 – Багатопотокова реалізація розрахунку суми ряду

3.3 Клас std::atomic

Ще одним засобом синхронізації доступу потоків до спільних даних є застосування узагальненого класу std::atomic<>. Об’єкти цього типу є неділимими (атомарними) з точки зору виконання потокових операцій над ними. Тобто ні один потік не може побачити проміжний стан змінної такого типу, що гарантує коректність роботи з ними в багатопотокових процесах.

Розглянемо наступний класичний приклад, в якому виникає гонитва за даними. Нехай є деяка змінна цілого типу, яку один потік в циклі зменшує на одиницю, а другий – збільшує. Ясно, що при однаковій кількості ітерацій в обох потоках, початкове значення змінної теоретично не повинно змінитися. Проте, якщо не виконувати синхронізацію, на практиці можуть бути варіанти. Розглянемо такий приклад програми (cpp_example7.cpp).

```
#include <thread>
#include <iostream>
#include <cassert>
```

```
using namespace std;

// Функція зменшення спільного ресурсу
void func_dec(int &value, int max_iter)
{
	for (auto i = 0; i < max_iter; i++)
		value--;
}

// Функція збільшення спільного ресурсу
void func_inc(int &value, int max_iter)
{
	for (auto i = 0; i < max_iter; i++)
		value++;
}
int main()
{
	const int max_iter{100000};
	int value{0};
	thread thr_dec(func_dec, ref(value), max_iter),
		thr_inc{func_inc, ref(value), max_iter};

	thr_dec.join();
	thr_inc.join();
	cout << "Value: " << value << endl;
	// Аварійне завершення програми, якщо value != 0
	assert(value == 0);
	return 0;
}
```

Запуск цієї програми може давати різні результати, наприклад, такий (рис. 3.8).

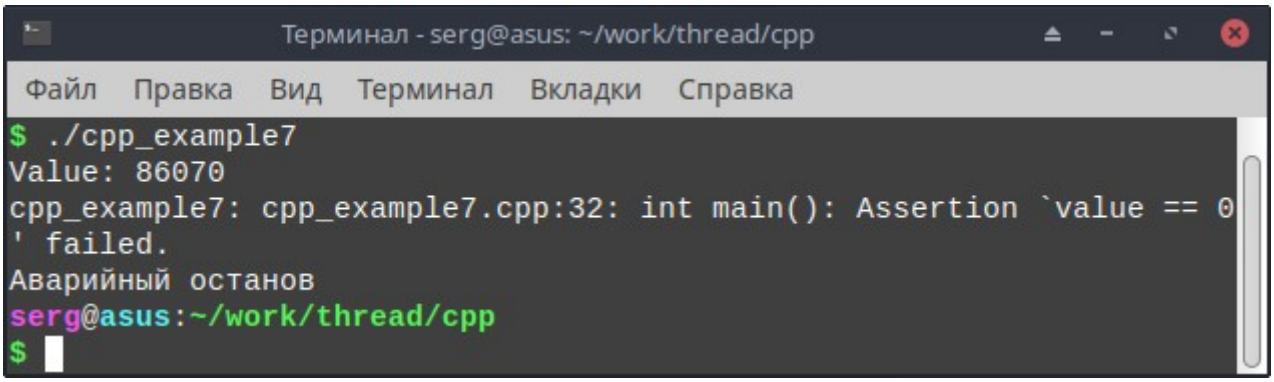


Рис. 3.8 – Гонитва за даними в програмі cpp_example7

Ясно, що для запобігання такому результату можна застосувати м’ютекс. Проте в даному випадку більш раціональним буде застосування std::atomic<>. Перепишемо програму cpp_example7.cpp таким чином (cpp_example8.cpp).

```
#include <thread>
#include <atomic>
#include <iostream>
#include <cassert>

using namespace std;

// Функція зменьшення спільного ресурсу
void func_dec(atomic<int> &value, int max_iter)
{
    for (auto i = 0; i < max_iter; i++)
        value--;
}
```

```
// Функція збільшення спільного ресурсу
void func_inc(atomic<int> &value, int max_iter)
{
    for (auto i = 0; i < max_iter; i++)
        value++;
}
int main()
{
    const int max_iter{100000};
    atomic<int> value{0};
    thread thr_dec(func_dec, ref(value), max_iter),
        thr_inc{func_inc, ref(value), max_iter};
    thr_dec.join();
    thr_inc.join();
    cout << "Value: " << value << endl;
    assert(value == 0);
    return 0;
}
```

Компіляція та запуск цієї програми дасть наступний результат (рис. 3.9).

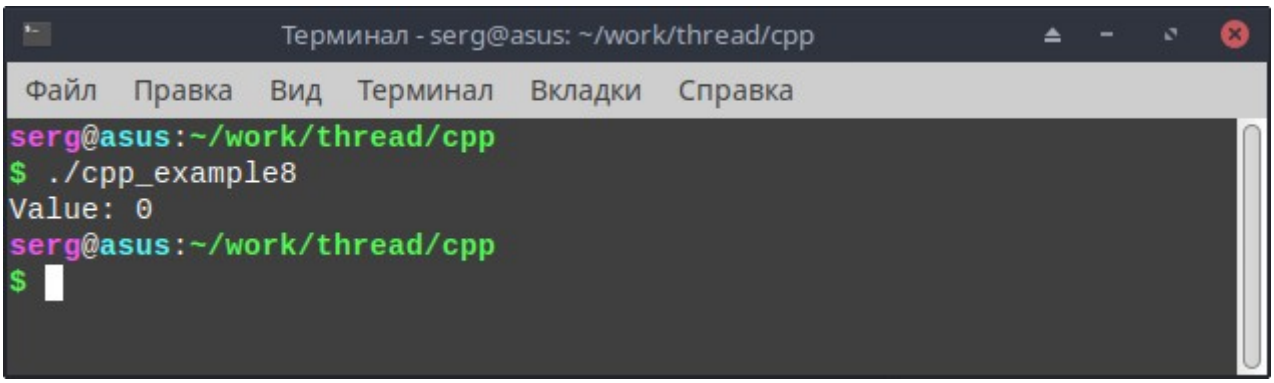


Рис. 3.9 – Запобігання гонитві за даними із застосуванням класу std::atomic<>

На жаль, клас std::atomic<> можна застосовувати не завжди, а лише для обмеженої кількості стандартних типів, таких, як bool, char, int тощо.

3.4 Асинхронні виклики

Як вже зазначалося, якщо однопотоковий процес виконує якусь тривалу операцію (наприклад, зчитує дані з великого файлу), то поки вона не закінчиться, процес не зможе виконувати ніяких інших завдань (оновлювати свій графічний інтерфейс, реагувати на команди користувача і тому подібне). Застосування потоків дозволяє вирішити цю проблему. Проте тут є певні нюанси. Після запуску дочірніх потоків батьківський потік повинен якимось чином синхронізувати свою роботу з ними. При використанні метод join() він фактично змушений очікувати завершення дочірніх потоків (тобто він фактично блокується). Якщо ж застосовується метод detach(), то це унеможливлює можливість отримання від дочірнього потоку результату його роботи.

Для уникнення цих проблем застосовується асинхронне програмування. Для цього в бібліотеці STL реалізовано спеціальну шаблонну функцію std::async(), яка оголошена в заголовному файлі future. Її застосування в загальному вигляді досить просте. В якості параметра в найпростішому випадку вона приймає покажчик на іншу функцію, що буде виконуватися паралельно й асинхронно по відношенню до поточного коду. Розглянемо наступний простий приклад (cpp_example9.cpp).

```
#include <iostream>
#include <future>

using namespace std;
```

```
// Асинхронна функція
void async_func(void)
{
    for (auto i = 0; i < 10; i++)
    {
        cout << "Async function: " << i << '\n';
        this_thread::sleep_for(0.1s);
    }
}

int main()
{
    // Асинхронний запуск функції
    auto res = async(async_func);
    for (auto i = 0; i < 10; i++)
    {
        cout << "Main function: " << i << '\n';
        this_thread::sleep_for(0.1s);
    }
    return 0;
}
```

Компіляція та запуск цієї програми призведе до наступного результату (рис. 3.10). З цього скріншоту видно, що функції main() та async_func() виконуються паралельно і абсолютно асинхронно по відношенню одна до одної (навіть при виведенні у стандартний потік виводу cout).

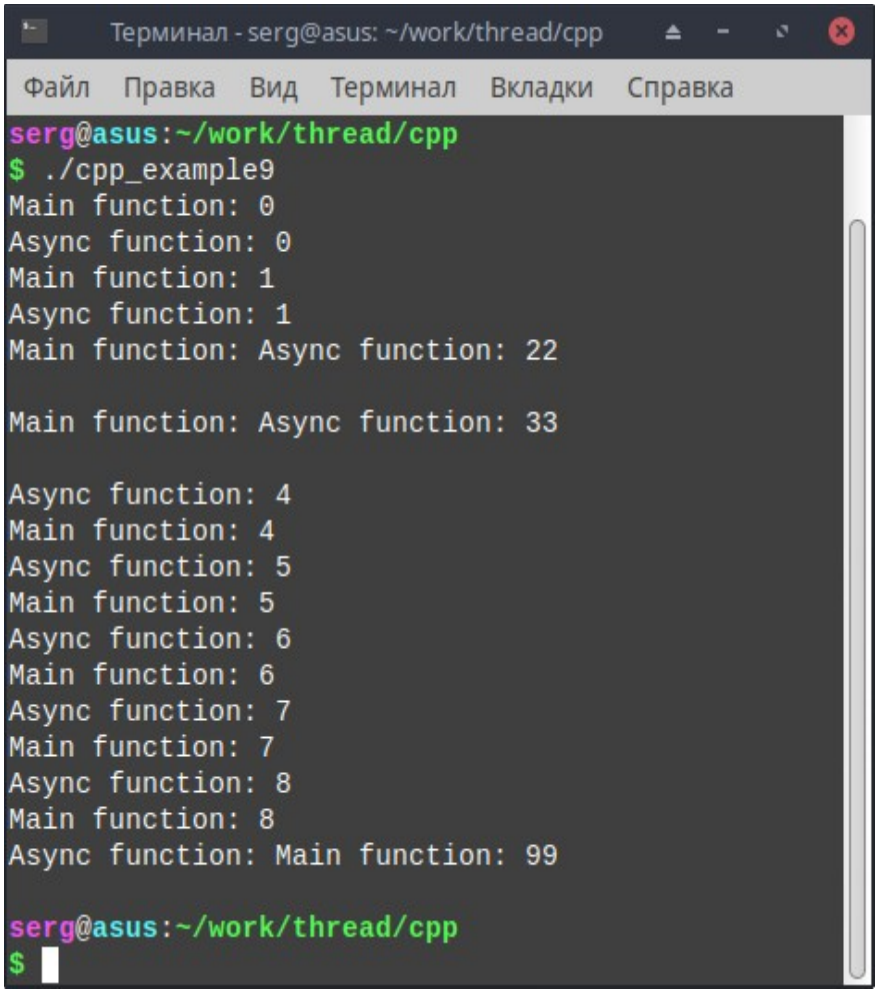


Рис. 3.10 – Асинхронне виконання двох функцій

В цій програмі необхідно прокоментувати наступне. Функція std::async() повертає об’єкт класу std::future, що призначений для збереження майбутнього результату (який буде повернено в майбутньому функцією, що асинхронно виконується). Найбільш часто вживаним методом класу std::future є get(), який повертає результат роботи асинхронної функції, коли він буде готовий.

Розглянемо такий простий приклад. Нехай потрібно обчислити значення виразу $\sum_{i=1}^{n} \frac{1}{1+i^2} \cdot \sum_{j=1}^{n} \frac{1}{2+j^3} \cdot \sum_{k=1}^{n} \frac{1}{3+k^4}$, яке утворюється добутком часткових сум трьох рядів. Для цього напишемо таку програму (cpp_example10.cpp).

#include <iostream>

```
#include <future>

using namespace std;
// Функція розрахунку суми ряду 1 / (1 + i * i)
double series_1(int n)
{
    double sum = 0;

    for (double i = 0; i < double(n); i++)
        sum += 1.0 / (1.0 + i * i);
    return sum;
}
// Функція розрахунку суми ряду 1 / (2 + i * i * i)
double series_2(int n)
{
    double sum = 0;

    for (double i = 0; i < double(n); i++)
        sum += 1.0 / (2.0 + i * i * i);
    return sum;
}

// Функція розрахунку суми ряду 1 / (3 + i * i * i * i)
double series_3(int n)
{
    double sum = 0;

    for (double i = 0; i < double(n); i++)
        sum += 1.0 / (3.0 + i * i * i * i);
```

```
    return sum;
}

int main()
{
    // Кількість ітерацій
    const auto n = 100000000;

    // Асинхронний запуск процедур розрахунку сум рядів
    auto res1 = async(series_1, n),
         res2 = async(series_2, n),
         res3 = async(series_3, n);
    auto start = chrono::system_clock::now();

    // Отримання асинхронного результату
    cout << "Async result: " << res1.get() * res2.get() * res3.get() << endl;
    cout << "Async time: " <<
                    double((static_cast<chrono::duration<double>>
                    (chrono::system_clock::now() - start).count())) <<
                    " sec" << endl;
    // Синхронізована перевірка ...
    start = chrono::system_clock::now();
    cout << "Sync result: " << series_1(n) * series_2(n) * series_3(n) << endl;
    cout << "Sync time: " <<
                     double((static_cast<chrono::duration<double>>
                    (chrono::system_clock::now() - start).count())) <<
                    " sec" << endl;
    return 0;
}
```

Компіляція та запуск цієї програми дадуть наступний результат (рис. 3.11).

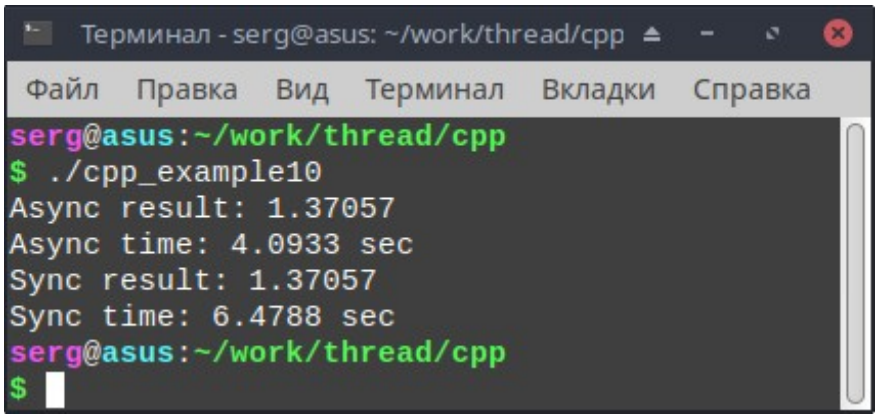


Рис. 3.11 – Результат роботи програми cpp_example10

3.5 Синхронізація доступу до стандартного потоку виводу

Як вже зазначалося, конкурентний доступ до стандартного потоку виведення (stream) може призводити до певних проблем, коли дані з різних потоків виконання (thread) змішуються (рис. 3.1, 3.10). Одним з можливих варіантів вирішення цієї проблеми є застосування синхронізації із застосуванням м’ютексів. Розглянемо наступний приклад (cpp_example11.cpp).

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <algorithm>

using namespace std;
```

```
int main()
{
    int num_thread = 10;
    vector<thread> thr;
    mutex mtx;

    cout << "Access to cout without mutex" << '\n';
    for (auto i = 0; i < num_thread; i++)
      thr.push_back(thread([] (int no)
      {
            cout << "Running " << "current " << "thread " << "no: " << no
                  << '\n';
            this_thread::sleep_for(0.2s);
      }, i));
    for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });
    thr.clear();
    cout << "Access to cout with mutex" << '\n';
    for (auto i = 0; i < num_thread; i++)
      thr.push_back(thread([&mtx] (int no)
      {
            mtx.lock();
            cout << "Running " << "current " << "thread " << "no: " << no
                  << '\n';
            mtx.unlock();
            this_thread::sleep_for(0.2s);
      }, i));
    for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });
```

```
    return 0;
}
```

Компіляція та запуск цієї програми може дати, наприклад, такий результат (рис. 3.12).

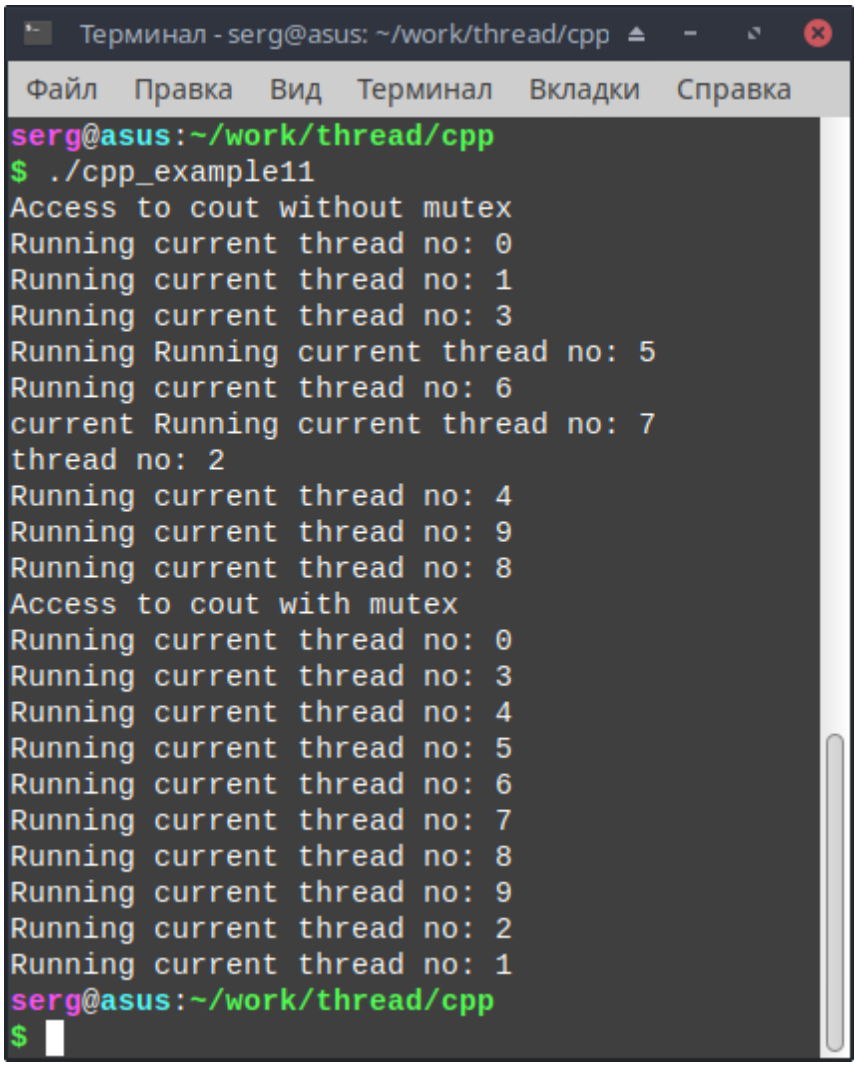


Рис. 3.12 – Приклад доступу до стандартного потоку виведення без синхронізації та із застосуванням синхронізації

Тут слід прокоментувати наступне. У конструктор класу std::thread можна передавати не тільки покажчик на потокову функція, але й лямбда вираз. Розглянемо, наприклад, таку лямбда-функцію.

```
[&mtx] (int no)
        {
                mtx.lock();
                cout << "Running " << "current " << "thread " << "no: " << no
                        << '\n';
                mtx.unlock();

                this_thread::sleep_for(0.2s);
        }
```

В ній відбувається захоплення із зовнішньої області видимості посилання на змінну **mtx** (м’ютекс). Крім того, вона приймає один цілий параметр **no** (номер потоку). Код цієї лямбда-функції реалізує захоплення м’ютексу, виведення у стандартний потік std::cout і звільнення м’ютексу.

Ще одним способом вирішення проблеми синхронізації доступу до потоку виведення є створення власного класу, що реалізує коректний доступ до std::cout у конкурентному середовищі. Реалізувати такий клас можна, наприклад, таким чином.

```
// Реалізація класу для виведення у std::cout у конкурентному середовищі
struct pcout : public stringstream
{
    static inline mutex mtx;
    ~pcout()
    {
       mtx.lock();
       cout << rdbuf();
       cout.flush();
```

```
        mtx.unlock();
    }
};
```

Цей клас є нащадком від std::stringstream (оголошений у sstream), який, в свою чергу, наслідує свій функціонал від std::iostream і реалізує виведення інформації у строку (string) із застосуванням інтерфейсу std::iostream. В pcout оголошено статичний м’ютекс (є спільним для всіх об’єктів цього класу). В деструкторі із застосуванням м’ютексу виконується виведення накопичених даних у стандартний потік std::cout.

Застосовувати клас pcout можна, наприклад, таким чином (cpp_example12.cpp).

```
#include <thread>
#include <mutex>
#include <iostream>
#include <sstream>
#include <algorithm>
#include <vector>

using namespace std;

// Опис класу pcout
// ...
int main()
{
    int num_thread = 10;
    vector<thread> thr;
```

```
    cout << "Using standard cout" << '\n';
    for (auto i = 0; i < num_thread; i++)
       thr.push_back(thread([] (int no)
       {
              cout << "Running " << "current " << "thread " << "no: " << no
                     << '\n';
       }, i));
    for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });
    thr.clear();
    cout << "Using pcout" << '\n';
    for (auto i = 0; i < num_thread; i++)
       thr.push_back(thread([] (int no)
       {
              pcout() << "Running " << "current " << "thread " << "no: "
                            << no << '\n';
       }, i));
    for_each (thr.begin(), thr.end(), [](auto &it) { it.join(); });

    return 0;
}
```

Запуск цієї програми приведе до такого можливого результату (рис. 3.13). Тут слід прокоментувати наступне. При використанні конструкції «pcout() << ...» динамічно створюється об’єкт класу pcout, в нього виводиться інформація, після чого цей об’єкт знищується і відповідно автоматично запускається деструктор, який реалізує синхронізоване виведення даних у стандартний потік std::cout.

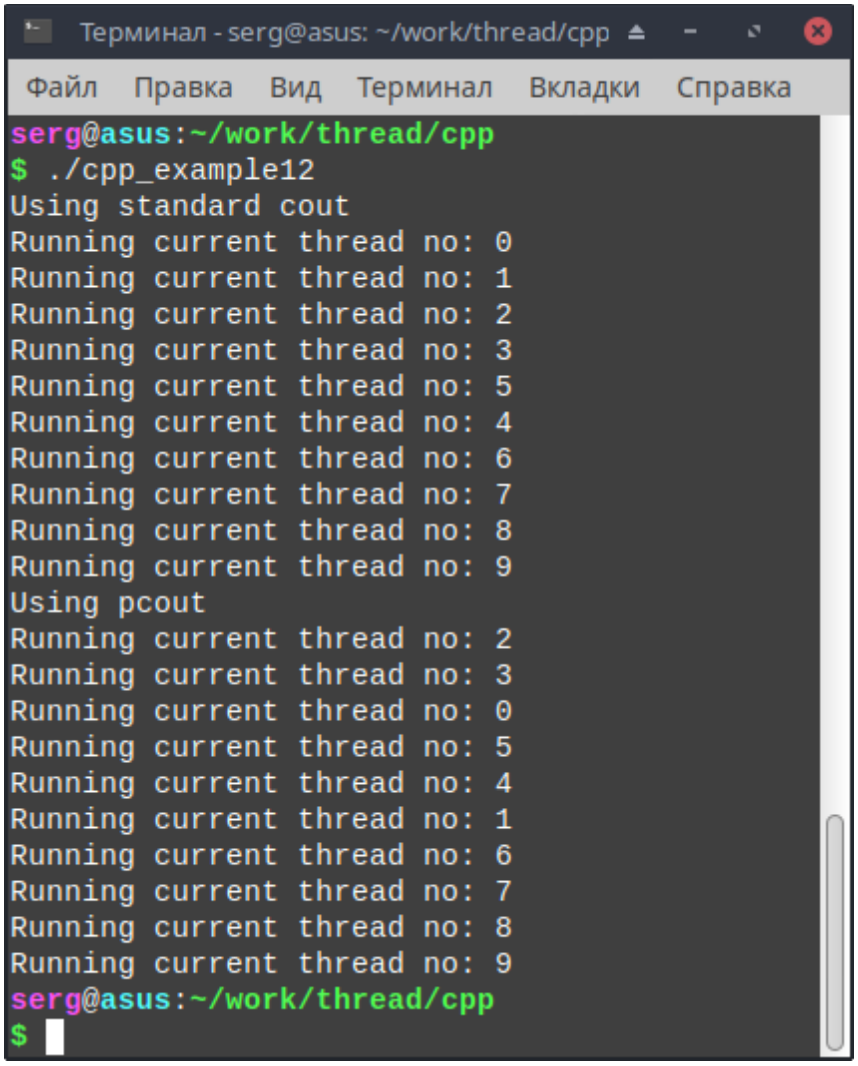


Рис. 3.13 – Результат роботи програми cpp_example12

3.6 Автоматичне розпаралелювання програмного коду із застосуванням стандартних алгоритмів

В бібліотеці STL реалізовано велику кількість стандартних алгоритмів для роботи з контейнерами (пошук, сортування, копіювання, видалення тощо). Починаючи зі стандарту C++17 багато з них отримали підтримку паралелізму, яка реалізується за рахунок додавання параметру ExecutionPolicy, який визначає політику виконання алгоритму (послідовну чи паралельну).

Розглянемо наступний приклад (cpp_example13.cpp).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <execution>

using namespace std;

int main()
{
    const size_t size{10000000};
    vector<int> arr(size);
    auto begin{chrono::system_clock::now()};
    for (auto &it : arr)
        it = rand();

    auto max = max_element(arr.begin(), arr.end());
    cout << "Max element of random array: " << *max << '\n';
    cout << "Nonparallel locate time: " <<
        (static_cast<chrono::duration<double>>(chrono::system_clock::now() -
        begin).count()) << " s" << '\n';

    begin = chrono::system_clock::now();
    max = max_element(std::execution::par, arr.begin(), arr.end());
    cout << "Max element of random array: " << *max << '\n';

    cout << "Parallel locate time: " <<
```

```
        (static_cast<chrono::duration<double>>(chrono::system_clock::now() -
        begin).count()) << " s" << endl;

    return 0;
}
```

Для того, щоб ця програма скомпілювалася, необхідно встановити бібліотеку **tbb**. В Debian-подібних версіях Linux (наприклад, Ubuntu) зробити це можна такою командою «sudo apt install libtbb-dev» (для цього потрібно мати права адміністратора). Крім того, при компіляції цієї програми потрібно додати цю бібліотеку (ключ «-ltbb»), а також вказати, що компіляція здійснюється із застосуванням стандарту C++17 (ключ «-std=c++17»). Таким чином, повна команда компіляції цієї програми буде мати такий вигляд «g++ cpp_example13.cpp -o cpp_example13 -std=c++17 -lpthread -ltbb».

Компіляція та запуск цієї програми дадуть наступний результат (рис. 3.14).

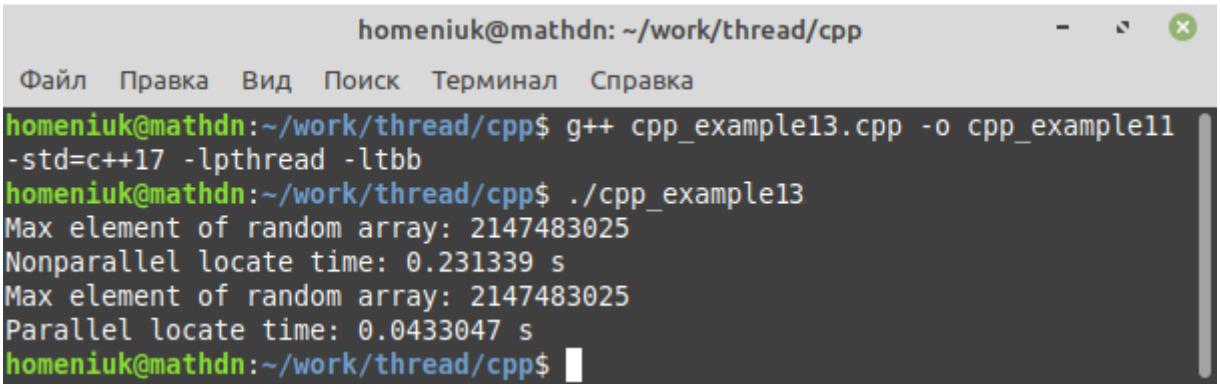


Рис. 3.14 – Час роботи стандартного алгоритму пошуку максимального значення в контейнері при послідовній та паралельній роботі

З наведеного скріншоту видно, що паралельний алгоритм виконується приблизно в п’ять разів швидше, ніж послідовний (на конкретному комп’ютері).

Таким чином, навіть без явного написання багатопотокових програм можна досягти значного автоматичного зростання швидкості роботи програм.

Питання для самоперевірки

1. Починаючи з якого стандарту в мові програмування C++ з’явилася підтримка потоків?
2. В якому заголовному файлі описуються класи, що використовуються для створення та керування потоками?
3. Як називається клас, із застосуванням якого створюються потоки виконання в C++?
4. Які методи класу std::thread використовуються для синхронізації потоку?
5. З якою функцією асоціюється головний потік програми на C++?
6. Що відбудеться, якщо не виконати синхронізацію дочірніх потоків з головним?
7. Для чого застосовується функція sleep_for()? В якому просторі імен вона визначена?
8. Який клас бібліотеки STL інкапсулює поняття м’ютексу?
9. За допомогою якого методу класу std::mutex здійснюється захоплення м’ютексу?
10. Для яких цілей застосовується клас std::function?
11. Для чого використовується узагальнений клас std::atomic<>?
12. Як ви розумієте асинхронне програмування?
13. Для чого використовується функція std::async()?
14. Для чого застосовується клас std::future?

15. Чи модна без явного написання багатопотокових програм досягти зростання швидкості роботи програм при застосуванні бібліотеки STL?

Вправи

1. Із застосуванням класу std::mutex напишіть багатопотокову програму, що обчислює значення інтегралу функції $y=\sin(x)$ на заданому відрізку методом прямокутників.

2. Напишіть програму, яка здійснює сканування заданого каталогу на предмет знаходження текстових файлів, які містять вказаний користувачем рядок символів.

3. Перепишіть програму обчислення інтегралу таким чином, щоб замість std::mutex використовувався клас std::atomic. Порівняйте час виконання програм.

ТЕМА 4 ВИСОКОРІВНЕВА БІБЛІОТЕКА BOOST

Boost – це високорівнева кросплатформна бібліотека класів, написаних на мові програмування C++, що реалізує різноманітні задачі по роботі з даними, такі, наприклад, як багатопотокове програмування, робота з текстами, ітератори, контейнери, стандартні алгоритми тощо.

Відомо, що в бібліотеці Boost проходять попереднє тестування можливості, які потім додаються до наступного стандарту мови C++. Отже, Boost – це потужна і популярна сучасна бібліотека, в якій в тому числі реалізовані всі необхідні класи для створення багатопотокових програм.

Для застосування Boost спочатку слід встановити цю бібліотеку. В Debian-подібних ОС це можна зробити наступною командою «sudo apt-get install libboost-all-dev». Для компіляції програм, що використовують Boost, необхідно підключати відповідні бібліотеки. Так, наприклад, наступні ключі компілятора «-lboost_system -lboost_thread» вказують на необхідність додати до поточної програми системну та потокову бібліотеки Boost. Оскільки повна команда компіляції багатопотокової програми із застосуванням бібліотеки Boost може бути достатньо громіздкою (наприклад, «g++ boost_example0.cpp -o boost_example0 -lboost_system -lboost_date_time -lboost_thread -lpthread»), то в даному випадку зручно буде застосовувати автоматичне збирання програми.

4.1 Автозбирання програм в Linux

Збиранням прийнято називати процес підготовки програми до безпосереднього використання. Найпростішим прикладом збирання є компіляція та компонування програм. Складні проєкти можуть включати в себе

додаткові проміжні етапи (різноманітні операції над файлами, конфігурування, перевірку залежностей і тому подібне). Кожен раз виконувати вручну ці дії незручно, тому програмісти, як правило, використовують різні програмні засоби, що дозволяють автоматизувати цей процес (виконувати **автозбирання**). Найбільш відомими засобами автозбирання в ОС Linux є такі програми, як make, cmake і qmake.

Утиліта **make** є найбільш популярним засобом автоматичного збирання проєктів в Linux, оскільки вона застосовується ще з часів Unix. Створення програм за її допомогою зазвичай здійснюється за наступним алгоритмом:

1) готуються вихідні файли проєкту;

2) створюється make-файл, який містить всі відомості про проєкт, при цьому ім’я цього файлу може бути довільним, але на практиці найчастіше використовують стандартні – Makefile або makefile (в ОС Linux розрізняються імена файлів в різних регістрах), що прийняті за замовчанням;

3) запускається утиліта make, яка збирає проєкт згідно з командами, описаними в make-файлі.

Таким чином, щоб налаштувати автозбирання проєкту, в першу чергу, необхідно створити файл з ім’ям Makefile. В ньому можуть бути присутніми такі елементи:

- однорядкові коментарі (починаються з символу «#»);
- оголошення констант, що використовуються для підстановки (подібні константам препроцесору C++);
- цільові зв’язки, за допомогою яких задаються залежності між різними частинами програми, а також визначаються дії, які будуть виконуватися при збиранні програми (в будь-якому make-файлі повинна бути хоча б одна цільова зв’язка).

Для коректного написання make-файлу необхідно визначити основну й допоміжні цілі збирання проєкту (якщо вони існують). Як правило, основною ціллю будь-якого проєкту є створення бінарного виконуваного файлу (програми

або бібліотеки). Розглянемо створення make-файлу для наступної простої програми, яка складається з двох вихідних файлів (func.cpp та main.cpp).

```
// -------------- func.cpp --------------
#include <iostream>

void thread_func(void)
{
	std::cout << "Boost thread" << std::endl;
}

// -------------- main.cpp --------------
#include <boost/thread/thread.hpp>
#include <iostream>

int main()
{
	void thread_func(void);
	boost::thread thr{thread_func};

	std::cout << "Main thread" << std::endl;
	thr.join();
	return 0;
}
```

Нехай основною метою є створення бінарного файлу boost_example0. Щоб його отримати потрібно отримати об’єктні файли main.o та func.o (це проміжні цілі створення програми). Крім того, очевидно, що досягнення основної мети неможливо без проміжних – це залежності нашого проекту. У

свою чергу, отримати, наприклад, файл main.o без main.cpp неможливо. Це ще одна залежність. І так далі.

Таким чином, кожна цільова зв’язка у make-файлі складається з наступних компонентів:

– ім’я цілі, що закінчується двокрапкою (якщо ціллю є файл, то вказується його ім’я);

– список залежностей (тут просто перераховуються через пробіл імена файлів або імена проміжних цілей; якщо мета ні від чого не залежить, то цей список буде порожнім);

– інструкції (команди, які повинні виконуватися для досягнення мети, наприклад, в цільової зв’язці main.o інструкцією буде команда компіляції файлу main.cpp).

Кожна інструкція в make-файлі пишеться з нового рядка і починається з символу табуляції.

Розглянемо приклад створення файлу makefile для проєкту boost_example0.

```
# Makefile for boost_example0
boost_example0: main.o func.o
	g++ -o boost_example0 main.o func.o -lpthread -lboost_system \
	-lboost_thread

main.o: main.cpp
	g++ -c main.cpp

func.o: func.cpp
	g++ -c func.cpp

clean:
```

```
rm -f *.o
rm -f boost_example0
```

Тепер після створення make-файлу, зібрати проєкт можна за допомогою простої команди «make» (рис. 4.1).

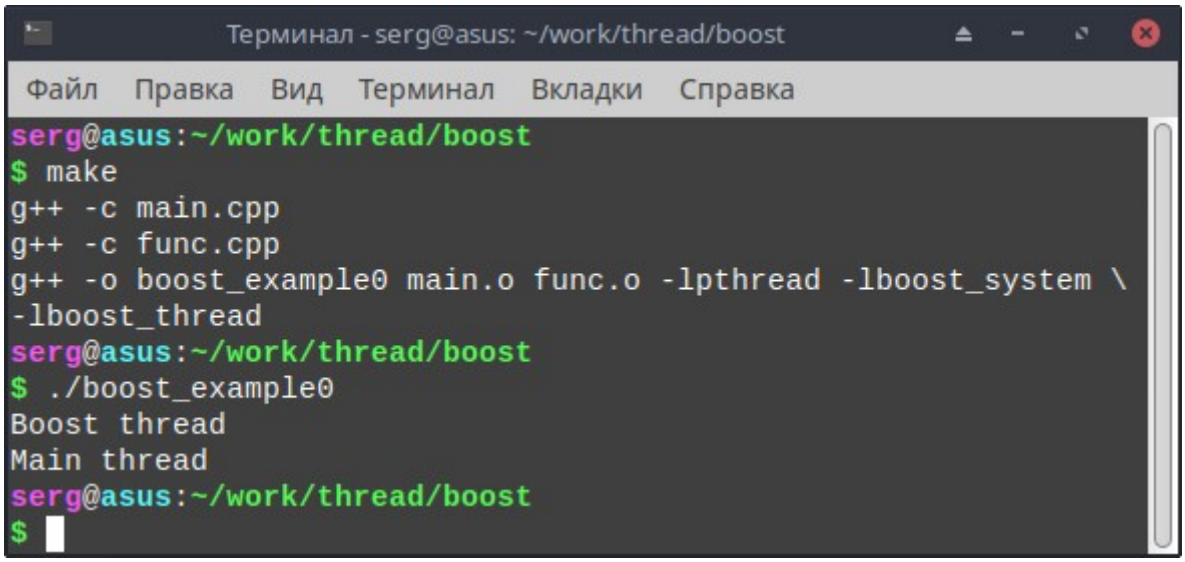


Рис. 4.1 – Приклад застосування утиліти make

Остання секція (clean) у make-файлі потребує пояснення:

1) спочатку вказується ім’я мети (clean);

2) після двокрапки йде порожній список залежностей, який означає, що дана зв’язка не вимагає наявності будь-яких файлів і не передбачає попереднього виконання проміжних цілей;

3) в наступних двох рядках прописані інструкції, що видаляють об’єктні файли і власне саму програму.

За допомогою цієї цілі програміст може швидко очистити каталог свого проекту від усіх файлів, автоматично створених при збиранні. Щоб зробити це, потрібно виконати наступну команду: «make clean».

4.2 Створення потоків із застосуванням класу boost::thread

Клас boost::thread (заголовний файл boost/thread/thread.hpp) в цілому є аналогом std::thread (точніше кажучи, std::thread є реалізованою в STL версією boost::thread). Він має декілька конструкторів: пустий конструктор, який створює відповідний об’єкт без прив’язки до нього коду виконання; конструктор переміщення; а також конструктор, що приймає функцію (або функціональний об’єкт), і її (його) параметри (при необхідності). Крім того, як і у випадку std::thread, клас boost::thread містить методи join() та detach().

Розглянемо приклад, у якому в одному потоці виконується тривала операція, а у іншому – анімація прогресу виконання (boost_example1.cpp).

```
#include <boost/thread/thread.hpp>
#include <iostream>

using namespace std;

// Анімація прогресу виконання
void show_progress(bool &is_process)
{
	char chr[] = { '/', '-', '\\', '|' };
	int i = 0;

	while (is_process)
	{
		cout << "\rProgress: " << chr[i++ % 4];
		boost::this_thread::sleep_for(boost::chrono::milliseconds(1));
	}
```

```
}

// Функція, що виконується 10 секунд
void long_term_func(bool &is_process)
{
	boost::this_thread::sleep_for(boost::chrono::seconds(10));
	is_process = false;
}

int main()
{
	bool is_process; // Індикатор виконання тривалої операції

	// Створення потоку, що виконує анімацію прогресу виконання
	(boost::thread(show_progress, boost::ref(is_process = true))).detach();
	// Запуск тривалої операції
	cout << "Start long term operation\n";
	long_term_func(is_process);
	cout << "\rFinish long term operation" << endl;
	return 0;
}
```

Для компіляції цієї програми слід додати ще одну бібліотеку boost_chrono, яка реалізує різноманітні операції з часом (опція «-lboost_chrono»). Крім того, тут слід прокоментувати наступне. Функція show_progress() в циклі виводить в один і той же рядок на екрані по черзі послідовність символів '/', '-', '\\' та '|', що створює у користувача ілюзію обертання риски. Цей процес виконується до того моменту, доки значення параметру **is_process** не стане мати значення «false», що свідчить про завершення тривалої операції.

Функція long_term_func() не виконує ніякої корисної дії, вона лише імітує виконання потоку тривалий проміжок часу. Для цього використовується функція boost::this_thread::sleep_for() з простору імен boost::this_thread, що призупиняє роботу поточного потоку виконання на вказаний проміжок часу, який задається із застосуванням, наприклад, функції boost::chrono::seconds().

Також слід відзначити, що оскільки результат роботи потоку, що виконує анімацію процесу виконання, не має значення, то для його створення фактично застосовується тимчасовий об’єкт, який від’єднується від основного потоку.

4.3 Синхронізація доступу до спільних ресурсів

Для синхронізації доступу потоків до спільних ресурсів в бібліотеці Boost застосовується клас boost::mutex, який описано у заголовному файлі boost/thread/mutex.hpp. В цілому його застосування не відрізняється від раніше наведених прикладів.

Розглянемо реалізацію прикладу з пункту 3.3 із використанням м’ютексу (boost_example2.cpp).

```
#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <iostream>
#include <cassert>

using namespace std;

// Глобальний м'ютекс
boost::mutex mtx;
```

```
// Функція зменшення спільного ресурсу
void func_dec(int &value, int max_iter)
{
	// Початок критичної області
	boost::lock_guard<boost::mutex> lock(mtx);

	for (auto i = 0; i < max_iter; i++)
		value--;
	// Кінець критичної області
}

// Функція збільшення спільного ресурсу
void func_inc(int &value, int max_iter)
{
	// Початок критичної області
	boost::lock_guard<boost::mutex> lock(mtx);

	for (auto i = 0; i < max_iter; i++)
		value++;
	// Кінець критичної області
}
int main()
{
	const int max_iter = 100000;
	int value = 0;
	boost::thread thr_dec(func_dec, boost::ref(value), max_iter),
			thr_inc{func_inc, boost::ref(value), max_iter};
	thr_dec.join();
	thr_inc.join();
```

```
    cout << "Value: " << value << endl;
    // Аварійне завершення програми, якщо value != 0
    assert(value == 0);
    return 0;
}
```

Результат роботи цієї програми наведено на рис. 4.2.

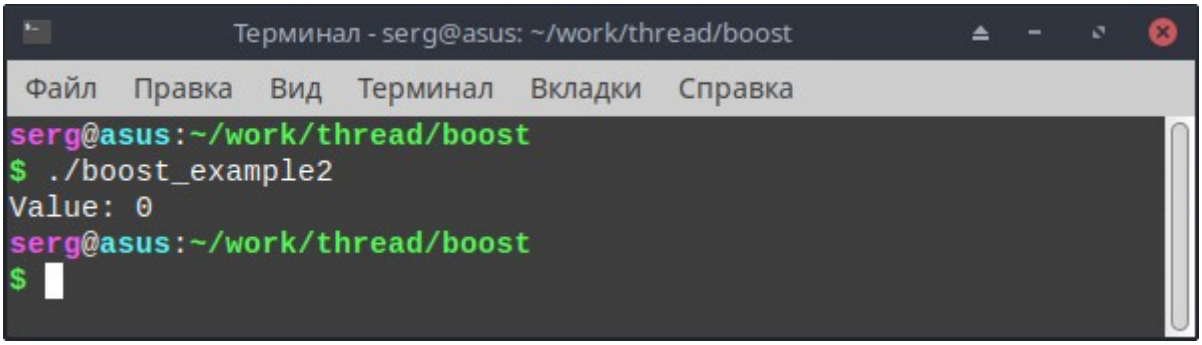


Рис. 4.2 – Результат роботи програми boost_example2

Проте, якщо закоментувати у функціях func_dec() і func_inc() рядки boost::lock_guard<boost::mutex> lock(mtx), то результат роботи може змінитися, наприклад, на такий (рис. 4.3).

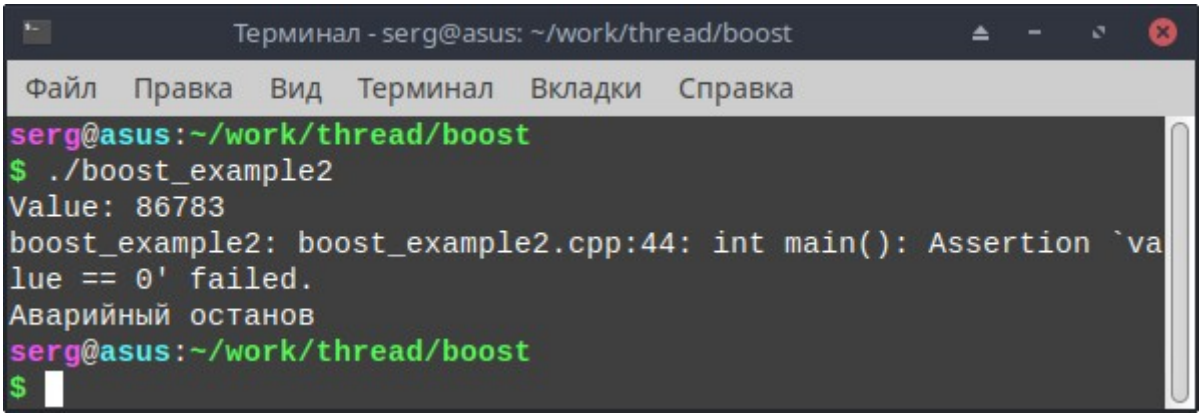


Рис. 4.3 – Можливий результат роботи програми boost_example2 без застосування м'ютексу

Тут слід лише прокоментувати застосування узагальненого класу boost::lock_guard<>, який реалізує концепцію RAII[3], що дозволяє автоматично захопити м’ютекс при створенні об’єкту boost::lock_guard<boost::mutex> і звільнити його при знищенні цього об’єкту.

Аналогічно тому, як це робилося із застосуванням бібліотеки STL, в Boost для синхронізації можна застосовувати шаблонний клас boost::atomic<>. Попередній приклад із застосуванням цього класу можна реалізувати таким чином (boost_example3.cpp).

```
#include <boost/thread/thread.hpp>
#include <boost/atomic.hpp>
#include <iostream>
#include <cassert>

using namespace std;

void func_dec(boost::atomic<int> &value, int max_iter)
{
	for (auto i = 0; i < max_iter; i++)
		value--;
}

void func_inc(boost::atomic<int> &value, int max_iter)
{
	for (auto i = 0; i < max_iter; i++)
		value++;
```

3 RAII (Resource Acquisition Is Initialization) – ідіома об’єктно-орієнтованого програмування, яка полягає в тому, що деякий ресурс автоматично отримується при створенні певного об’єкта, і звільняється при його знищенні. Наприклад, файл може автоматично відкриватися при створенні об’єкту типа ifstream (виклику відповідного конструктора) і закриватися при знищенні цього об’єкта (виклику деструктора). Застосування RAII дозволяє, зокрема, запобігати такій проблемі, як витік пам’яті (memory leak).

```
}

int main()
{
	const int max_iter{100000};
	boost::atomic<int> value{0};
	boost::thread thr_dec(func_dec, boost::ref(value), max_iter),
			thr_inc{func_inc, boost::ref(value), max_iter};

	thr_dec.join();
	thr_inc.join();
	cout << "Value: " << value << endl;
	assert(value == 0);
	return 0;
}
```

4.4 Переривання потоків

На відміну від поточної реалізації std::thread у класі boost::thread наявні засоби переривання роботи потоку виконання. Для цього застосовується метод interrupt(), який примусово завершує роботу вказаного потоку. Перепишемо приклад з пункту 2.2.3 із застосуванням класу boost::thread (boost_example4.cpp).

```
#include <boost/thread/thread.hpp>
#include <iostream>
```

```
using namespace std;

// Функція потоку
void thread_code(void)
{
	auto i = 0;

	while (1)
	{
		cout << "Child thread: " << i++ << '\n';
		boost::this_thread::sleep_for(boost::chrono::milliseconds(1));
	}
}

int main (void)
{
	const int max_iter{10};
	boost::thread thr(thread_code);

	// Функціонал головного потоку
	for (auto i = 0; i < max_iter; i++)
	{
		if (i == 4)
			thr.interrupt(); // Примусове завершення потоку
		cout << "Main thread: " << i << '\n';
		boost::this_thread::sleep_for(boost::chrono::milliseconds(1));
	}
	// Блокування головного потоку до завершення дочірнього
	thr.join();
```

```
    return 0;
}
```

Його компіляція та запуск може привести, наприклад, до такого результату (рис. 4.4). З наведено скріншоту видно, що після четвертої ітерації циклу у головному потоці дочірній потік примусово завершив свою роботу.

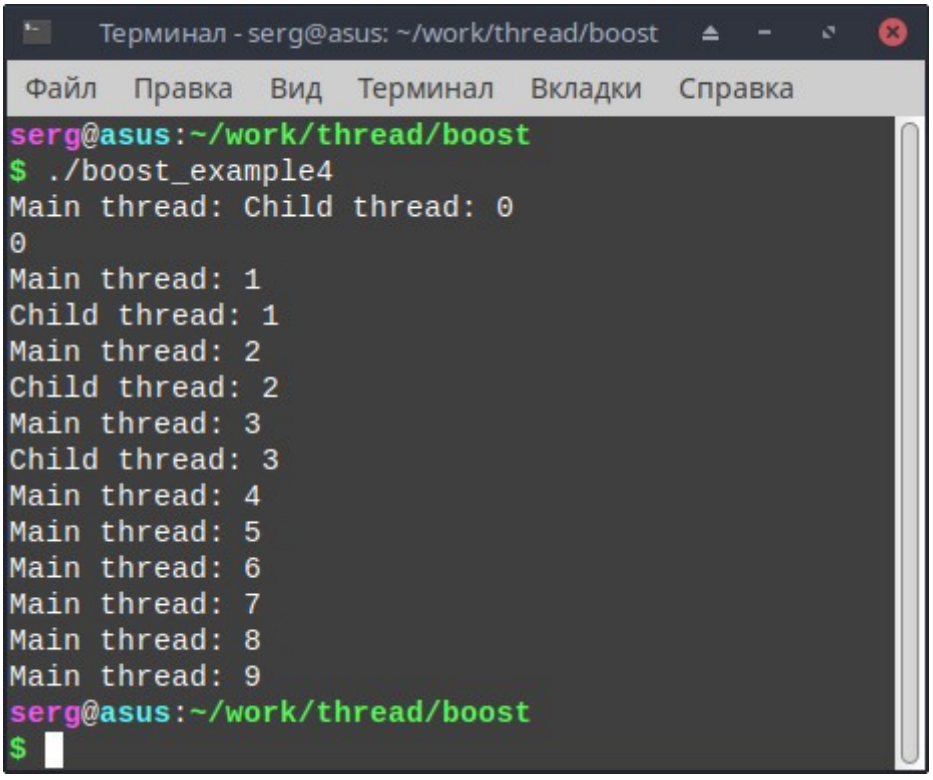


Рис. 4.4 – Результат роботи програми boost_example4

Як і у бібліотеці P-thread у Boost є можливості блокування потоком свого завершення іззовні. Для цього він повинен створити у себе об’єкт класу boost::this_thread::disable_interruption. Внесемо, наприклад, у вищенаведену функцію thread_code() внести такі зміни.

```
void thread_code(void)
{
```

```
        auto i = 0;
        boost::this_thread::disable_interruption di;

        while (1)
        {
            cout << "Child thread: " << i++ << '\n';
            boost::this_thread::sleep_for(boost::chrono::milliseconds(1));
        }
}
```

Результатом роботи програми boost_example4 в такій редакції стане її фактичне зациклювання. Тут слід також відзначити, що для компіляції цієї програми потрібно додати ще одну бібліотеку – boost_chrono (ключ компілятора «-lboost_chrono»).

Якщо потоку потрібно буде відновити можливість його переривання ззовні, йому потрібно буде створити ще одну змінну – об’єкт класу boost::this_thread::restore_interruption. Зробити це потрібно, наприклад, наступним чином.

```
void thread_code(void)
{
        auto i = 0;
        boost::this_thread::disable_interruption di;

        while (1)
        {
            boost::this_thread::restore_interruption ri(di);

            cout << "Child thread: " << i++ << '\n';
            boost::this_thread::sleep_for(boost::chrono::milliseconds(1));
```

```
        }
}
```

4.5 Групова робота з потоками

Для групового запуску потоків в бібліотеці Boost є спеціальний клас boost::thread_group. Розглянемо застосування цього класу, переписавши приклад із пункту 2.2.5 із застосуванням бібліотеки Boost (boost_example5.cpp).

```
#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <iostream>


using namespace std;

// Глобальний м'ютекс
boost::mutex mtx;

// Потокова функція
void thread_func(int begin, int end, double &sum)
{
        double local_sum = 0;

        // Обчислення суми членів заданого діапазону ряду
        for (double i = begin; i < end; i++)
                local_sum += (1.0 / (1.0 + i * i * i));
```

```
        boost::lock_guard<boost::mutex> lock(mtx);
        sum += local_sum;
}

int main(int argc, char **argv)
{
        int step,
            num_thread,
            max_iter;
        double sum = 0;
        boost::thread_group threads;
        boost::chrono::steady_clock::time_point start =
                                        boost::chrono::steady_clock::now();
        // Обробка аргументів: числа потоків і кількості члені ряду
        if (argc != 3)
        {
                cerr << "Too few arguments!" << endl;
                return 1;
        }
        if ((num_thread = atoi(argv[1])) == 0)
        {
                cerr << "Wrong number of threads!" << endl;
                return 1;
        }
        if ((max_iter = atoi(argv[2])) == 0)
        {
                cerr << "Wrong number of iteration!" << endl;
                return 1;
```

```
        }

        step = max_iter / num_thread;
        // Створення та запуск потоків
        for (auto i = 0; i < num_thread; i++)
                threads.add_thread(new boost::thread(thread_func, i * step,
                                (i == num_thread - 1 ? max_iter : (i + 1) * step),
                                boost::ref(sum)));
        threads.join_all();
        // Виведення результату
        cout << "Sum of series for " << num_thread <<
                        " thread(s) and number of iteration " << max_iter <<
                        " :" << sum << '\n';
        // ... часу виконання
        cout << "Working time: " <<
                ((boost::chrono::duration<double>)
                        (boost::chrono::steady_clock::now() - start)).count() <<
                        " sec" << endl;
        return 0;
}
```

Результат роботи цієї програми наведено на рис. 4.5.

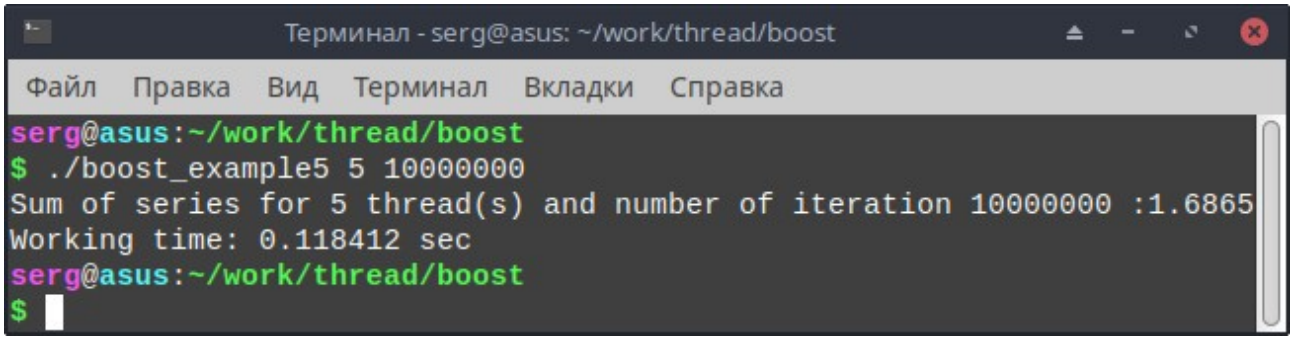


Рис. 4.5 – Результат роботи програми boost_example5

Питання для самоперевірки

1. Дайте визначення поняття «збирання».
2. Що таке автозбирання? Які утиліти застосовуються для автозбирання?
3. З якою метою використовується утиліта make? Чим вона відрізняється від cmake та qmake?
4. Які стандартні імена підтримуються програмою make?
5. Яку функцію виконує секція clean у make-файлі?
6. Для яких цілей застосовується бібліотека Boost?
7. У якому заголовному файлі описано клас boost::thread? Для чого він застосовується?
8. Опишіть алгоритм створення потоку виконання із застосуванням бібліотеки Boost.
9. Які способи синхронізації потоків засобами бібліотеки Boost ви знаєте?
10. За допомогою якого класу в Boost реалізуються м’ютекси?
11. Чи є засоби переривання потоків в Boost?
12. Чи є в Boost можливості блокування потоком свого примусового завершення?
13. Для чого застосовується клас boost::this_thread::disable_interruption?
14. Для чого застосовується клас boost::thread_group? Чи є його аналог в бібліотеці STL?
15. Які можливості з написання багатопотокових програм бібліотеки Boost відсутні в STL?

Вправи

1. Напишіть багатопотокову програму, що обчислює методом трапецій значення інтегралу функції $y=\cos(x)$ на заданому відрізку.

2. Із використанням бібліотеки Boost напишіть програму, яка здійснює пошук файлів, що містять вказаний користувачем рядок символів. Передбачте можливість переривання користувачем роботи програми.

3. За допомогою бібліотеки Boost напишіть багатопотокову програму перемноження матриць. Порівняйте час її виконання з програмою, написаної із застосуванням бібліотеки P-thread.

ТЕМА 5 ТЕХНОЛОГІЯ OPENMP

OpenMP (Open Multi-Processing) – відкритий стандарт для розпаралелювання програм, написаних мовами програмування C, C++ та Fortran. В ньому описана сукупність директив компілятора, бібліотечних процедур та змінних оточення, що призначені для створення багатопотокових паралельних програм, що будуть працювати у багатопроцесорних обчислювальних системах із загальною пам’яттю.

Розробка специфікацій стандарту ведеться некомерційною організацією OpenMP Architecture Review Board (ARB), до якої входять усі основні виробники процесорів, а також ряд суперкомп’ютерних лабораторій та університетів. Перша версія специфікації була оприлюднена у 1997 р. і підтримувала лише мову Fortran. У 1998 р. вийшла версія для C/C++.

Важливою перевагою технології OpenMP у порівнянні з іншими підходами до розробки паралельного ПЗ є можливість використання інкрементального підходу до програмування, коли програміст покроково переробляє у паралельні певні ділянки послідовної програми. Це дозволяє полегшити переробку однопотокових програм у паралельні та спростити їх налагодження та супровід.

5.1 Директиви OpenMP

Паралельні обчислення в OpenMP реалізуються за допомогою багатопотоковості, де головний (master) потік створює набір дочірніх потоків, після чого завдання розподіляється між ними. Передбачається, що потоки

виконуються паралельно на машині з декількома процесорами або ядрами, при цьому їх кількість не обов’язково має збігатися з кількістю потоків.

Код, що виконується потоками паралельно, як і відповідні дані, описуються за допомогою спеціальних директив препроцесора відповідної мови – так званих **прагм**. Наприклад, наступна прагма препроцесора компілятора мов C/C++ **#pragma omp parallel num_threads(10)** визначає паралельну область у програмі, що буде виконуватися у 10 потоках. Основні директиви OpenMP наведено у табл. 5.1.

Таблиця 5.1. Основні прагми OpenMP

Директива	Опис
parallel	Визначає паралельний регіон у коді, який виконуватиметься кількома потоками паралельно
for	Визначає, що робота в циклі в середині паралельного регіону буде розділена між потоками
sections	Визначає розділи коду, які необхідно поділити між усіма потоками
single	Дозволяє вказати, що розділ коду повинен виконуватись в одному потоці, а не в основному потоці.

У головному потоці та для синхронізації роботи потоків можна використовувати наступні директиви (табл. 5.2).

Таблиця 5.2. Прагми головного потоку та синхронізації

Директива	Опис
master	Вказує, що лише основний потік має виконувати розділ програми
critical	Вказує, що код виконується лише в одному потоці одночасно
barrier	Синхронізує всі потоки в команді: кожен потік зупиняється на бар’єрі, доки всі інші не дійдуть до цієї точки
atomic	Визначає розташування пам’яті, яке оновлюватиметься атомарно
flush	Вказує, що всі потоки мають однакове подання пам’яті для всіх спільних об’єктів
ordered	Вказує, що код паралельного циклу for повинен виконуватися як послідовний

Крім того, для визначення локальних (приватних) у потоці змінних використовується прагма **threadprivate**, а для визначення пам’яті, що буде оновлюватися атомарним чином, використовується прагма **atomic**.

Таким чином, всі директиви OpenMP можна розділити на три категорії: визначення паралельної області, розподіл роботи та синхронізація. Кожна директива може мати кілька додаткових опцій (clause).

Розглянемо наступний приклад (openmp_example1.cpp).

```
#include <iostream>

int main() {
   int count = 0;
// Визначення паралельного регіону, що буде виконуватися 10 потоками
#pragma omp parallel num_threads(10)
   {
// Атомарне збільшення змінної
#pragma omp atomic
      count++;
   }
   std::cout << "Number of threads: " << count << "\n";
   return 0;
}
```

У цьому1 прикладі за допомогою директиви **#pragma omp parallel num_threads(10)** оголошується секція коду, що буде паралельно виконуватися у вказаній кількості потоків. У кожному з них завдяки прагмі **#pragma omp atomic** буде атомарно (безпечно) збільшуватися значення спільної для всіх потоків змінної count. Компіляція та запуск цієї програми дадуть наступний результат (рис. 5.1).



Рис. 5.1 – Результат виконання програми openmp_example.cpp

Як видно з наведеного скріншоту, програма виконувалася в одному потоці. Це пояснюється тим, що для отримання паралельності при компіляції слід додати відповідний ключ (**-fopenmp** у даному випадку) (рис. 5.2).



Рис. 5.2 – Паралельне виконання openmp_example.cpp

Розглянемо ще один приклад. Нехай потрібно обчислити значення константи π. Це можна зробити шляхом обчислення суми наступного ряду: $\frac{1}{n}\sum_{i=1}^{n}\frac{4}{1+x_i^2}$, де $x_i=\frac{i-0,5}{n}$. Послідовна реалізація такого обчислення може бути виконана так (calc_pi.cpp).

```
#include <iostream>

using namespace std;
```

```
int main()
{
    double sum = 0.0;
    int n = 1000000;

    for (int i = 1; i <= n; ++i)
        sum += 4.0/(1.0 + (i - 0.5)*(i - 0.5)/n/n);
    cout << "Sum = " << sum/n << endl;
    return 0;
}
```

Компіляція та запуск цієї програми приведуть до наступного результату (рис. 5.3).

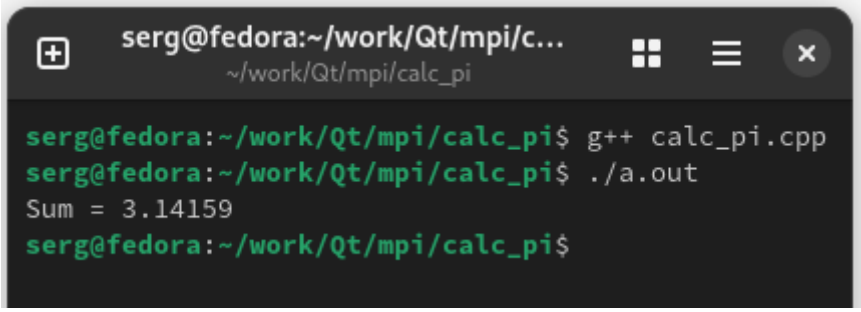


Рис. 5.3 – Обчислення числа π

Переробити цю програму на паралельну з використанням OpenMP дуже просто. Достатньо перед циклом дописати одну директиву розпаралелювання циклів **#pragma omp for**:

```
// ...
#pragma omp for
```

```
    for (int i = 1; i <= n; ++i)
// ...
```

Це приклад інкрементального програмування, яке може застосовуватися для переробки раніше написаних послідовних програм у паралельні.

5.2 Паралельні та послідовні області

У момент запуску програми автоматично створюється один master-потік, який при використанні мов програмування C/C++ визначається кодом функції main(). Основний потік монопольно виконує всі послідовні області програми, але при вході до паралельної області коду запускаються додаткові потоки виконання.

Як вже зазначалося, для створення паралельної області у програмі на мовах C/C++ вживається директива **#pragma omp parallel [опція1[, опція2, ...]]**.

Найбільш часто на практиці ця директива використовуються з опцією **num_threads (кількість потоків**) для явного завдання кількості потоків, що будуть виконувати паралельну область (за замовчуванням ця величина буде дорівнювати кількості наявних у системі процесорів або ядер). Кількість потоків також можна явно визначити з використанням спеціальної функції:

void omp_set_num_threads(int num_threads);

яка описана у заголовному файлі omp.h.

Розглянемо наступний приклад (openmp_example2.cpp).

```
#include <iostream>
#include <omp.h>

int main( )
{
   omp_set_num_threads(2);

#pragma omp parallel
   {
      // Виведення інформації про номер потоку
      std::cout << "Thread " << omp_get_thread_num() << ' ';
   }
   std::cout << std::endl;



#pragma omp parallel num_threads(5)
   std::cout << "Thread " << omp_get_thread_num() << ' ';

   std::cout << std::endl;
   return 0;
}
```

Компіляція та запуск цієї програми приведуть до наступного результату (рис. 5.4). З наведеного прикладу видно, що у першій паралельній області було створено два потоки (кількість яких визначена викликом omp_set_num_threads()), а у другій – кількість потоків була визначена явно за допомогою відповідної опції у прагмі. Крім того, у наведеному коді для виведення інформації про номер потоку використовувалась функція omp_get_thread_num(), яка також оголошена у файлі omp.h.



Рис. 5.4 – Приклад виведення інформації із застосуванням різної кількості потоків

Якщо паралельна секція повинна виконатися за певною умовою, то можна скористатися оцією **if (умова)**. Тоді входження у паралельну область буде здійснено лише за виконання вказаного предикату. Розглянемо наступний приклад (openmp_example3.cpp).

```
#include <iostream>
#include <omp.h>

void omp_region(int val)
{
   #pragma omp parallel if (val)
   if (omp_in_parallel())
   {
      #pragma omp single
      std::cout << "val = " << val << " parallelized with " <<
            omp_get_num_threads() << " threads" << '\n';
   }
   else
      std::cout << "val = " << val << " serialized" <<  '\n';
}
```

```
int main( )
{
    omp_set_num_threads(2);
    omp_region(0);
    omp_region(2);
    return 0;
}
```

Компіляція та запуск цієї програми приведуть до наступного результату (рис. 5.5). У наведеній програмі паралельний регіон реалізується у окремій функції. Крім того, для визначення того факту, чи виконується код у паралельній чи послідовній секції, використовується виклик omp_in_parallel().



Рис. 5.5 – Приклад використання опції if

Ще однією корисною опцією є **reduction** (редукція). Вона дозволяє вказати оператор редукції (зміни) та список змінних, для яких у кожному потоці буде створена та відповідним чином проініціалізована (для адитивних операцій – 0, для мультиплікативних – 1) локальна копія, яка наприкінці паралельного регіону згідно з вказаним оператором буде заданим способом додана до відповідної змінної послідовної секції. Розглянемо наступний приклад (openmp_example4.cpp).

```
#include <iostream>

using namespace std;

int main()
{
   int num_thread = 0;

#pragma omp parallel reduction(+ : num_thread)
   num_thread++;

   cout << "Number of threads " << num_thread << endl;
   return 0;
}
```

Тут у кожному потоці буде створена локальна версія змінної num_thread, яка буде спочатку проініціалізована нульовим значенням (у випадку для типу редукції «+»), а потім збільшена на одиницю. Наприкінці паралельної області ця локальна змінна буде додана до змінної num_thread, оголошеної у послідовній секції. Також слід додати, що тип змінних редукції повинен бути скалярним (int, float, double тощо).

Компіляція та запуск цієї програми дадуть наступний результат (рис. 5.6).



Рис. 5.6 – Обчислення кількості потоків за допомогою редукції

5.3 Цикли OpenMP

Якщо в паралельному регіоні коду буде знаходитись оператор циклу, то всі його ітерації згідно з загальним підходом OpenMP будуть виконані усіма заданими потоками. Щоб запобігти цьому та коректно розпаралелити виконання циклу у всіх потоках в OpenMP використовується спеціальна директива **#pragma omp for [опція1[, опція2, ...]]**.

Розглянемо наступний простий приклад (openmp_example5.cpp).

```
#include <iostream>

using namespace std;

int main()
{
   constexpr  int n = 10; // Кількість потоків
   constexpr int max_iter = 50000; // Кількість ітерацій
   double sum = 0; // Змінна для збереження суми ряду

#pragma omp parallel for num_threads(n)
      for (int i = 1; i <= max_iter; i++)
#pragma omp critical
         sum += 1.0 / (i * i);

   cout << "Sum of series: " << sum << endl;
   return 0;
}
```

Тут за допомогою директиви **#pragma omp parallel for** виконується розпаралелювання виконання циклу, що обчислює суму заданої кількості членів ряду $\sum_{n=1}^{\infty} \frac{1}{n^2}$. Слід зазначити, що в даному прикладі за допомогою директиви **#pragma omp critical** оголошується критична область коду (яка може виконуватися одночасно лише у одному потоці). Таким чином усувається можлива гонитва за даними. Компіляція та запуск цієї програми дадуть наступний результат (рис. 5.7).

Якщо в цій програмі закоментувати рядок з директивою #pragma omp critical, то результат її виконання кожного разу буде різним (тобто невизначеним), оскільки при збільшенні значення змінної sum кожного разу може виникати гонитва за даними (рис. 5.8).



Рис. 5.7 – Приклад реалізації паралельного циклу

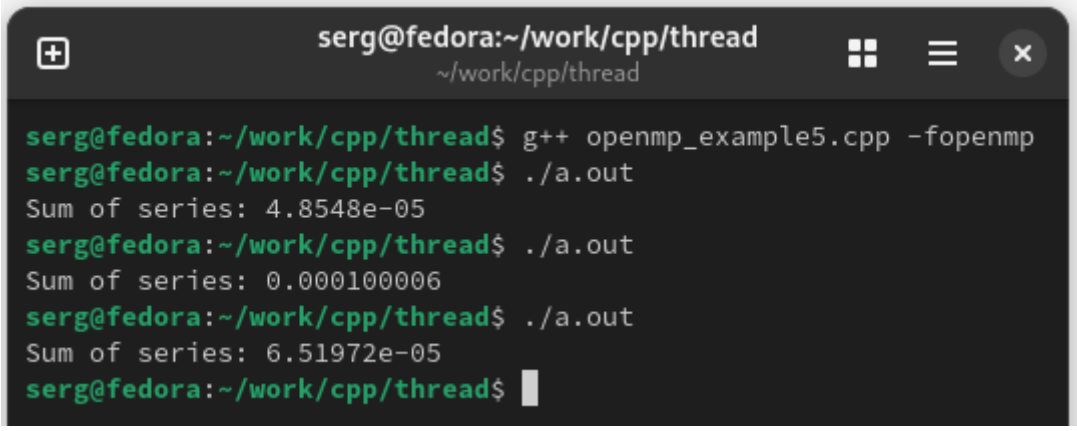


Рис. 5.8 – Приклад невизначеної поведінки програми

Тут лише слід зазначити, що для запобігання гонитви у наведеному прикладі можна замість **#pragma omp critical** скористатися раніше розглянутою прагмою **#pragma omp atomic** або використати редукцію.

```
#include <iostream>

int main()
{
   constexpr  int n = 10;
   constexpr int max_iter = 50000;
   double sum = 0;

#pragma omp parallel for num_threads(n) reduction(+: sum)
   for (int i = 1; i <= max_iter; i++)
      sum += 1.0 / (i * i);

   std::cout << "Sum of series: " << sum << std::endl;
   return 0;
}
```

5.4 Синхронізація потоків

Найбільш поширеним способом синхронізації є використання директиви **#pragma omp barrier**. Будь-який потік, що доходить до цієї точки програми, призупиняє свою роботу до тих пір, поки всі інші потоки також сюди не дійдуть. Після чого вони розблоковуються і продовжують свою роботу.

Розглянемо наступний приклад (openmp_example6.cpp).

```
#include <iostream>
int main()
{
#pragma omp parallel num_threads(3)
    {
        std::cout << "Message 1\n";
        std::cout << "Message 2\n";
#pragma omp barrier
        std::cout << "Message 3\n";
    }
    return 0;
}
```

Результат компіляції та запуску цієї програми наведено на рис. 5.9.



Рис. 5.9 – Приклад використання бар'єру

З наведеного скріншоту видно, що повідомлення «Message 3» виводиться на екран лише після того, як всі потоки виведуть на екран повідомлення «Message 1» та «Message 2». Тобто робота потоків певним чином була синхронізована. Таким чином прагма **#pragma omp barrier** в певному сенсі є аналогом методу join() класу std::thread мови C++.

Питання для самоперевірки

1. В якому році вперше було оприлюднено стандарт OpenMP?
2. Які мови програмування підтримують стандарт OpenMP?
3. Яка організація займається розвитком OpenMP?
4. Назвіть основні переваги використання OpenMP.
5. Для яких типів комп’ютерів розроблявся стандарт OpenMP?
6. З виконання якого потоку розпочинає свою роботу OpenMP-програма?
7. Що так прагма OpenMP? Які їх категорії ви знаєте?
8. За допомогою якої директиви створюється паралельний регіон в програмі?
9. Яким чином можна явно задати кількість вживаних потоків виконання?
10. Як можна взнати номер поточного потоку виконання?
11. Що таке редукція в OpenMP?
12. Якими способами в OpenMP можна запобігти гонитві за даними?
13. Як розпаралелити роботу циклу в OpenMP?
14. В який спосіб виконується синхронізація роботи потоків в OpenMP?
15. Назвіть особливості компіляції OpenMP-програм.

Вправи

1. Із використанням технології OpenMP напишіть багатопотокову програму, що без використання математичної бібліотеки обчислює значення функції $y=\cos(x)$.

2. Із використанням OpenMP напишіть програму, що виконує пошук у вказаній теці текстових файлів, які містять заданий користувачем рядок символів.

3. Із використанням OpenMP напишіть програму, що обчислює добуток двох матриць.

ТЕМА 6 РЕАЛІЗАЦІЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MPI

MPI (Message Passing Interface, інтерфейс передачі повідомлень) – це найбільш поширена на практиці технологія програмування для паралельних комп’ютерів з розподіленою пам’яттю. Головним засобом взаємодії між обчислювальними процесами, що можуть виконуватися як на одному, так і на декількох обчислювальних вузлах у мережі, є обмін повідомленнями. На сьогодні MPI реалізована для більшості комп’ютерних платформ і широко використовується для розробки паралельних програм, що виконуються на суперкомп’ютерах та обчислювальних кластерах.

Головною перевагою використання MPI при розробці паралельних програм є можливість відносно легкого перенесення MPI-програм з одної апаратної платформи на іншу.

MPI підтримує два способи написання паралельних програм MIMD і SIMD. У першому випадку на кожному обчислювальному вузлі запускається процес з власним початковим кодом, проте розробляти на налагоджувати такі програми доволі складно. Тому на практиці набагато частіше використовують другий тип програм, коли у всіх паралельних процесах працює один і той же програмний код.

Перша версія стандарту була оприлюднена у 1994 році. Як і у випадку з OpenMP MPI реалізується у вигляді відповідної бібліотеки для мов програмування Fortran та C/C++. Стандарт MPI 2.0 було оприлюднено у 1997 році, проте оскільки деякі його функції до кінця так і не були реалізовані, то розробники частіше за все використовують саме перший стандарт MPI (точніше його версію 1.1, яка вийшла у 1995 році).

6.1 Налаштування та тестування MPI

Для використання технології MPI спочатку слід встановити відповідну бібліотеку. Розглянемо це на прикладі однієї з найбільш популярних безкоштовних реалізацій MPI – бібліотеки MPICH (MPI Chameleon). При використанні дистрибутиву Linux Fedora це можна зробити, наприклад, за допомогою команди «sudo dnf install mpich-devel». В Ubuntu – за допомогою «sudo apt-get install mpich-dev».

Альтернативним варіантом може бути завантаження з офіційного сайту та встановлення пакету Intel oneAPI Math Kernel Library (oneMKL), який також містить власну реалізацію MPI.

Для зручності компіляції MPI-програм без використання інтегрованих середових розробки (IDE) рекомендується використовувати автозбирання. Наприклад, для компіляції програми, початковий код якої розташовано у файлі main.cpp, можна створити наступний простий Makefile.

```
# Makefile for program with MPI

mpi_example1: main.o
    /usr/lib64/mpich/bin/mpicxx -o mpi_example1 main.o \
    -L/usr/lib64/mpich/lib/ -lmpifort -lmpi

main.o: main.cpp
    /usr/lib64/mpich/bin/mpicxx -c main.cpp -I/usr/include/mpich-x86_64/

clean:
    rm -f *.o
    rm -f mpi_example1
```

Для його тестування, створимо у поточній теці відповідний файл початкового коду (main.cpp).

```
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char** argv)
{
    // Ініціалізація MPI
    MPI_Init(&argc, &argv);

    // Отримання кількості процесів, що виконуються
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Отримання рангу процесу
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Отримання назви процесора
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Друк повідомлення «Hello world»
    cout << "Hello world from processor " << processor_name << ", rank " <<
                world_rank << " out of " << world_size << " processors\n";
```

```
    // Фіналізація MPI
    MPI_Finalize();
    return 0;
}
```

Якщо все було зроблено вірно, то компіляція та запуск цієї програми буде виглядати так (рис. 6.1).



Рис. 6.1 – Компіляція та запуск MPI-програми

Тут слід прокоментувати наступне. Для запуску MPI-програми застосовується спеціальна утиліта mpiexec, параметром якої є кількість обчислювальних вузлів і безпосередньо шлях до програми, що запускається.

6.2 Базові функції бібліотеки MPI

Шаблони всіх функцій бібліотеки MPI та необхідні макровизначення описані у заголовному файлі mpi.h.

У будь-якій MPI-програмі першим має бути виклик функції MPI_Init(), що має наступний шаблон:

int MPI_Init(int *argc, char ***argv);

Ця функція ініціалізує середовище MPI. Її аргументами є покажчики на відповідні аргументи функції main(): **argc** – це покажчик на кількість аргументів командного рядку процесу; **argv** – покажчик на масив рядків, що містять параметри, передані у поточний процес. Функція MPI_Init() повертає або цілочисельне значення 0 (MPI_SUCCESS) у разі успіху, або певний код помилки.

Слід зазначити, що MPI-процеси можуть об’єднуватися в різноманітні групи, які називаються комунікаторами. Глобальний комунікатор MPI_COMM_WORLD посилається на всі запущені процеси. При використанні функцій передачі даних в якості одного з їх параметрів, як правило, передається ідентифікатор комунікатора, що використовується. Якщо одночасно виконується багато програм, то використання механізму комунікаторів дозволяє відокремлювати різноманітні групи процесів за певними критеріями.

Функція MPI_Comm_size() повертає кількість процесів, що були запущені у заданому комунікаторі. Її сигнатура наступна:

int MPI_Comm_size(MPI_Comm comm, int *size);

Тут параметр **comm** визначає поточний комунікатор, а в аргумент **size** у разі успіху буде записано кількість процесів, що працюють у заданому комунікаторі. При виникненні помилки ця функція повертає ненульове значення − код помилки.

Процеси в групі (комунікаторі) нумеруються послідовно від 0 до n-1, де n – їх загальна кількість. Цей номер (ідентифікатор) називається рангом процесу.

Функція MPI_Comm_rank() повертає ранг поточного процесу. Її шаблон має наступний вигляд:

int MPI_Comm_rank(MPI_Comm comm, int *rank);

Параметр **comm** визначає поточний комунікатор, а у **rank** у разі успіху буде записано ранг процесу, що викликає цю функцію, у групі комунікатора. При виникненні помилки ця функція також повертає ненульове значення − код помилки.

Функція MPI_Get_processor_name() використовується для ідентифікації обладнання, на якому виконується поточний процес. Її сигнатура наступна:

int MPI_Get_processor_name(char *name, int *resultlen);

Параметр **name** тут є покажчиком на область пам’яті (буфер), куди буде записано назву процесора, а **resultlen** − покажчик на цілу змінну, куди буде записано довжину назви. Як і у всіх попередніх функціях MPI_Get_processor_name() повертає або значення MPI_SUCCESS, або код помилки.

Функція MPI_Finalize() призначена для очищення всіх станів, пов’язаних з MPI. Її опис наступний:

int MPI_Finalize(void);

Вона не отримує жодних параметрів і після її виклику інші функції MPI не можуть викликатися. Вона також повертає або MPI_SUCCESS, або код помилки.

6.3 Обмін повідомленнями

Як вже зазначалося, основою MPI є обмін повідомленнями між процесами, що паралельно працюють (можливо розподіленими) (рис. 6.2).

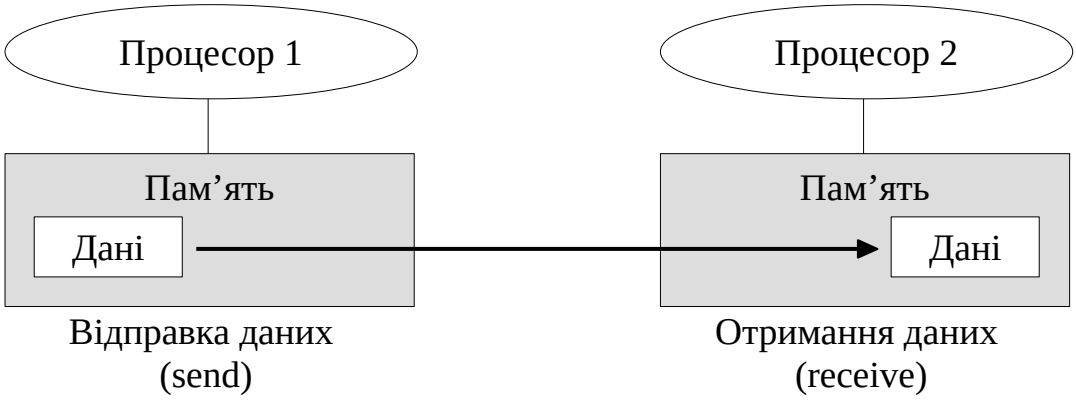


Рис. 6.2 − Обмін повідомленнями між процесами

Для відправлення повідомлення у синхронному режимі (з блокуванням) використовується функція MPI_Send(), яка має наступний шаблон:

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,
             int destination, int tag, MPI_Comm communicator);
```

Тут параметр **buffer** − це покажчик на буфер, що містить повідомлення, яке потрібно відправити; **count** − кількість елементів, що відправляються; **datatype** − тип даних повідомлення, такий як MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR тощо (тут також підтримуються типи даних, визначені користувачем); **destination** − ранг процесу, якому надсилається повідомлення; **tag** − ціле число, задане користувачем, що визначає мітку повідомлення (якщо вказати константу MPI_ANY_TAG, то тег перевірятися не буде); **communicator** − комунікатор, що визначає групу процесів, куди слід надіслати повідомлення.

Блокування призупиняє виконання процесу до тих пір, доки операція, яка виконується, не буде завершена, що забезпечує захист даних, що пересилаються.

Для синхронізованого отримання повідомлення використовується функція MPI_Recv(), яка має наступний синтаксис:

int MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status *status);

Тут параметри **buffer**, **count**, **datatype**, **tag** та **communicator** такі самі, як і у попередньому випадку; **source** − ранг процесу, від якого має бути отримано повідомлення (якщо в якості цього параметру використати константу MPI_ANY_SOURCE, то повідомлення буде прийнято з будь-якого джерела); **status** − покажчик на структуру MPI_Status, в якій зберігаються відомості про отримане повідомлення (кількість отриманих записів, джерело повідомлення, тег, код помилки тощо). В разі помилки обидві ці функції повертають ненульове значення − код помилки.

Для об’єднання (редукції) результатів, отриманих у різних процесах використовується функція MPI_Reduce(), яка описується наступним чином:

int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

Тут параметр **sendbuf** − покажчик на буфер, що містить дані, які потрібно відправити головному процесу; **recvbuf** − покажчик на буфер, для отримання результату (має сенс тільки у головному процесі); **count** − кількість елементів, що відправляються; **datatype** − тип даних, що відправляються; **op** − визначена в mpi.h операція редукції (наприклад, MPI_SUM або MPI_MAX); root − ранг приймаючого процесу у вказаному комунікаторі; comm − комунікатор.

Розглянемо наступний приклад, в якому процес з рангом 1 надсилає процесу з рангом 0 текстове повідомлення:

```
#include "mpi.h"
#include <iostream>

int main(int argc, char *argv[])
{
   const int max_len{100};
   std::string hello{"Hello"};
   char data[max_len];
   int pid, len = hello.length() + 1;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);

   if (pid == 0)
      MPI_Send(hello.c_str(), len, MPI_CHAR, 1, 10,
               MPI_COMM_WORLD);
   else
   {
      MPI_Recv(data, len, MPI_CHAR, 0, 10,
               MPI_COMM_WORLD, &status);
      std::cout << "Received data: " << data << std::endl;
   }
   MPI_Finalize();
   return 0;
}
```

Компіляція та запуск цієї програми повинні дати наступний результат (рис. 6.3).

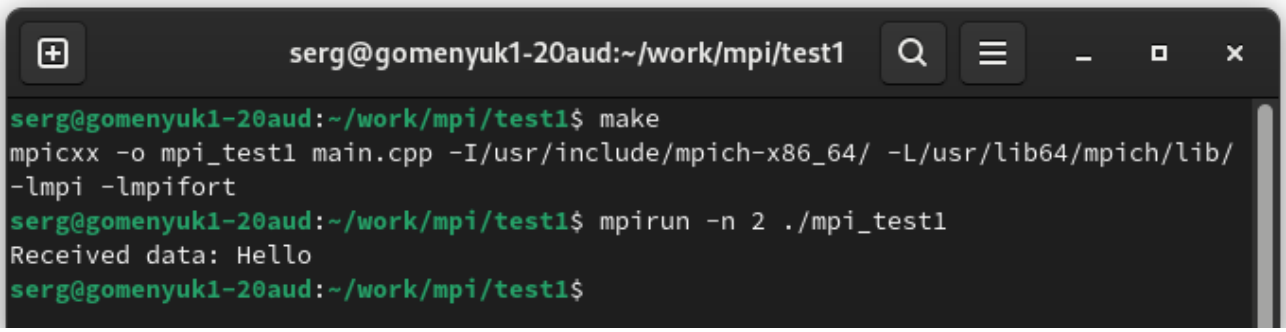


Рис. 6.3 − Обмін повідомленнями між процесами

Слід зазначити, що у певних випадках у залежності від налаштувань системи, при запуску таких програм можуть виникати проблеми, пов’язані з тим, що не знаходяться деякі файли бібліотеки MPI. Знайти місце розташування потрібної бібліотеки в Linux можна за допомогою команди locate. Тоді, вказати системі шлях до потрібної бібліотеки можна, наприклад такою командою (рис. 6.4):

export LD_LIBRARY_PATH=/usr/lib64/mpich/lib:$LD_LIBRARY_PATH

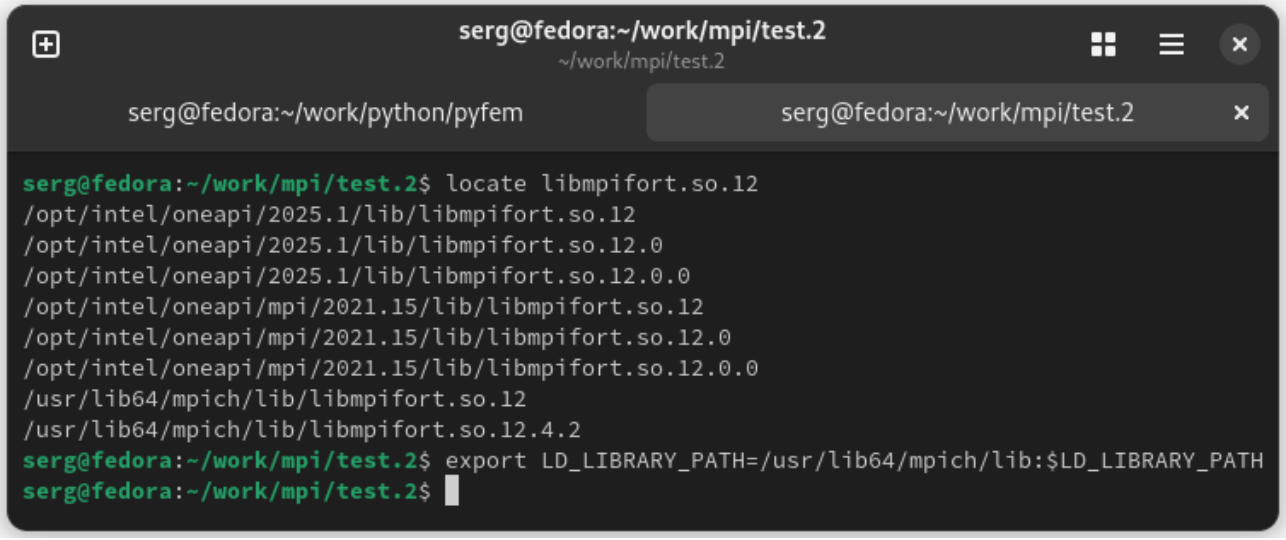


Рис. 6.4 − Налаштування доступу до бібліотек MPI

Розглянемо ще один приклад, в яком обчислюється значення числа π. Можливий варіант програмної реалізації такого розрахунку може виглядати наступним чином:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "mpi.h"

int main(int argc, char **argv)
{
   int n = 1000000, pid, numprocs;
   double local_pi, pi, h, sum, x, begin = 0;

   // Ініціалізація MPI
   MPI_Init(&argc, &argv);
   // Отримання розміру комунікатора MPI_COMM_WORLD
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   // Отримання рангу поточного процесу
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);

   if (pid==0)
      begin = MPI_Wtime();

   h = 1.0/(double) n;
   sum = 0.0;
   // Обчислення частини суми ряду у кожному процесі
   for (int i = pid + 1 ; i <= n; i += numprocs)
   {
```

```
        x = h*(i - 0.5);
        sum += (4.0/(1.0 + x*x));
    }
    local_pi = h*sum;

    // Редукція отриманих результатів
    MPI_Reduce(&local_pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD);

    // Вивід результату у головному процесі
    if (pid == 0)
    {
        std::cout << "PI is approximately " << std::setprecision(16) << pi <<
                            ", Error is " << fabs(pi - M_PI) << std::endl;
        std::cout << "Calculation time: " << MPI_Wtime() - begin << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Компіляція та запуск цієї програми дадуть наступний результат (рис. 6.5). Тут слід прокоментувати наступне. Функція MPI_Wtime() повертає дійсне число, яке є кількістю секунд, що пройшли з якось певного моменту у минулому. Для обчислення часу роботи потрібно обчислити різницю результатів викликів цієї функції перед початком і після закінчення обчислень.

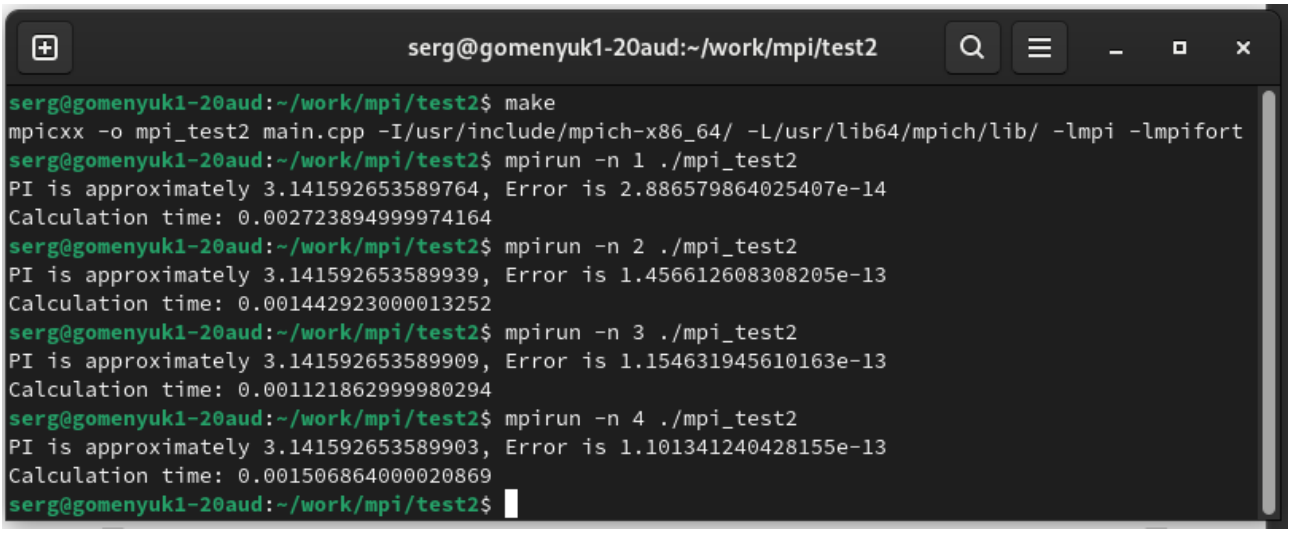


Рис. 6.5 − Обчислення числа π

Питання для самоперевірки

1. В якому році було оприлюднено стандарт MPI?
2. Які мови програмування підтримують стандарт MPI?
3. Назвіть переваги технології MPI.
4. Яким способом здійснюється взаємодія між процесами в MPI?
5. Що таке MIMD-програма?
6. Що таке SIMD-програма?
7. Які реалізації MPI ви знаєте?
8. Для чого призначена утиліта mpiexec?
9. Які заголовні файли описують функції, структури даних та константи MPI?
10. Для чого використовується функція MPI_Init()? Які параметри вона отримує?
11. Що таке комунікатор?
12. Дайте визначення рангу процеса.

13. Як визначити кількість процесів у комунікаторі?
14. Як відбувається передача повідомлення між процесами?
15. Для чого використовується функція MPI_Reduce()?

Вправи

1. Із використанням MPI напишіть програму обчислення значення функції $y=\sin(x)$ шляхом розкладання її в ряд.
2. Із використанням MPI напишіть програму, що виконує сортування злиттям великого масиву даних.
3. Із використанням MPI напишіть програму, що обчислює добуток двох матриць.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна:

1. Швець Н. В. Операційна система Linux : посіб. для самостійної роботи. Одеса : Одеська державна академія холоду, 2010. 132 с.

2. Харченко В. П., Знаковська Є. А., Бородін В. А. Операційні системи та системи програмування: навч. посіб. Київ : Вид-во Нац. авіац. ун-ту «НАУ-друк», 2012. 360 с.

3. Галкін О. В., Верес М. М. Мова програмування С++: конспект лекцій. Київ : ДП «Вид. дім «Персонал», 2017. 260 с.

4. Горбань Г. В., Кандиба І. О. Операційна система Linux : навчальний посібник. Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2019. 276 с.

Додаткова:

1. Love R. Linux Kernel Development. Third edition. Boston : Addidon Wesley, 2010. 480 p.

2. Prata S. C Primer Plus. 6th Edition. Addison-Wesley, 2013. 1067 p.

3. Meyers S. Effective Modern C++. 1st Edition. O’Reilly, 2014. 334 p.

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Love R. Linux System Programming. O’Reilly, 2013. 456 p.

2. Persson J.-B. Linux System Programming Techniques. Packt Publishing Ltd, 2012. 432 p.

3. OpenMP – Home. URL: https://www.openmp.org/ (дата звернення 04.04.2025).

4. MPICH | High-Performance Portable MPI. URL: https://www.mpich.org/ (дата звернення 15.04.2025).

153

Навчальне видання
(українською мовою)

Гоменюк Сергій Іванович
Лісняк Андрій Олександрович
Кудін Олексій Володимирович
Гребенюк Сергій Миколайович
Чопорова Оксана Володимирівна

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Навчальний посібник

для здобувачів ступеня вищої освіти магістра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Інженерія програмного забезпечення»

Рецензент Г. М. Шило

Відповідальний за випуск А. О. Лісняк

Коректор С. І. Гоменюк