Міністерство освіти та науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики і програмної інженерії

Звіт
з дисципліни «Технології розподілених обчислень»
Комп’ютерний практикум №1
«Розробка потоків та дослідження пріоритету запуску потоків»

Виконав:
Студент III курсу
гр. ІП-33
Соколов О. В.

Прийняв:
Дифучин А. Ю.
“14” Лютого 2026 р.

Київ – 2026

Комп'ютерний практикум №1

Тема: Розробка потоків та дослідження пріоритету запуску потоків.

Мета: Здобути практичні навички багатопоточної розробки мовою Java. Дослідити подію гонки потоків на прикладі лічильника за допомогою різних методів синхронізації і їх відмінність від непотокобезпечних операцій.

Завдання:

1. Реалізуйте програму імітації руху більярдних кульок, в якій рух кожної кульки відтворюється в окремому потоці (див. презентацію «Створення та запуск потоків в java» та приклад). Спостерігайте роботу програми при збільшенні кількості кульок. Поясніть результати спостереження. Опишіть переваги потокової архітектури програм. 10 балів.

2. Модифікуйте програму так, щоб при потраплянні в «лузу» кульки зникали, а відповідний потік завершував свою роботу. Кількість кульок, яка потрапила в «лузу», має динамічно відображатись у текстовому полі інтерфейсу програми. 15 балів.

3. Виконайте дослідження параметру priority потоку. Для цього модифікуйте програму «Більярдна кулька» так, щоб кульки червоного кольору створювались з вищим пріоритетом потоку, в якому вони виконують рух, ніж кульки синього кольору. Спостерігайте рух червоних та синіх кульок при збільшенні загальної кількості кульок. Проведіть такий експеримент. Створіть багато кульок синього кольору (з низьким пріоритетом) і одну червоного кольору, які починають рух в одному й тому ж самому місці більярдного стола, в одному й тому ж самому напрямку та з однаковою швидкістю. Спостерігайте рух кульки з більшим пріоритетом. Повторіть експеримент кілька разів, значно збільшуючи кожного разу кількість кульок синього кольору. Зробіть висновки про вплив пріоритету потоку на його роботу в залежності від загальної кількості потоків. 25 балів.

4. Побудуйте ілюстрацію методу join() класу Thread через взаємодію потоків, що відтворюють рух більярдних кульок різного кольору. Поясніть результат, який спостерігається. 25 балів.

5. Створіть клас Counter з методами increment() та decrement(), які збільшують та зменшують значення лічильника відповідно. Створіть два потоки, один з яких збільшує 100000 разів значення лічильника, а інший – зменшує 100000 разів значення лічильника. Запустіть потоки на одночасне виконання. Спостерігайте останнє значення лічильника. Поясніть результат. 10 балів. Використовуючи синхронізований доступ, добийтесь правильної роботи лічильника при одночасній роботі з ним двох і більше потоків. Опрацюйте використання таких способів синхронізації: синхронізований метод, синхронізований блок, блокування об’єкта. Порівняйте способи синхронізації. 15 балів.

Виконання

Завдання 1

За основу проєкту взято приклад із презентації «Створення та запуск потоків у Java». Реалізовано програму імітації руху більярдних кульок, у якій рух кожної кульки виконується в окремому потоці. Клас BallThread розширює Thread; у методі run() у циклі викликається ball.move() та Thread.sleep() для заданого інтервалу. При натисканні «Add Ball» або «Add 10» створюються нова кулька та новий потік, який одразу запускається. Канва відображає кількість кульок, кількість потоків кульок, FPS, навантаження CPU процесу JVM, кількість потоків у станах RUNNABLE та WAITING, а також графік залежності FPS і CPU від часу (у правому верхньому куті). UI та ця інформативність не змінюють логіку потоків і структуру класів.

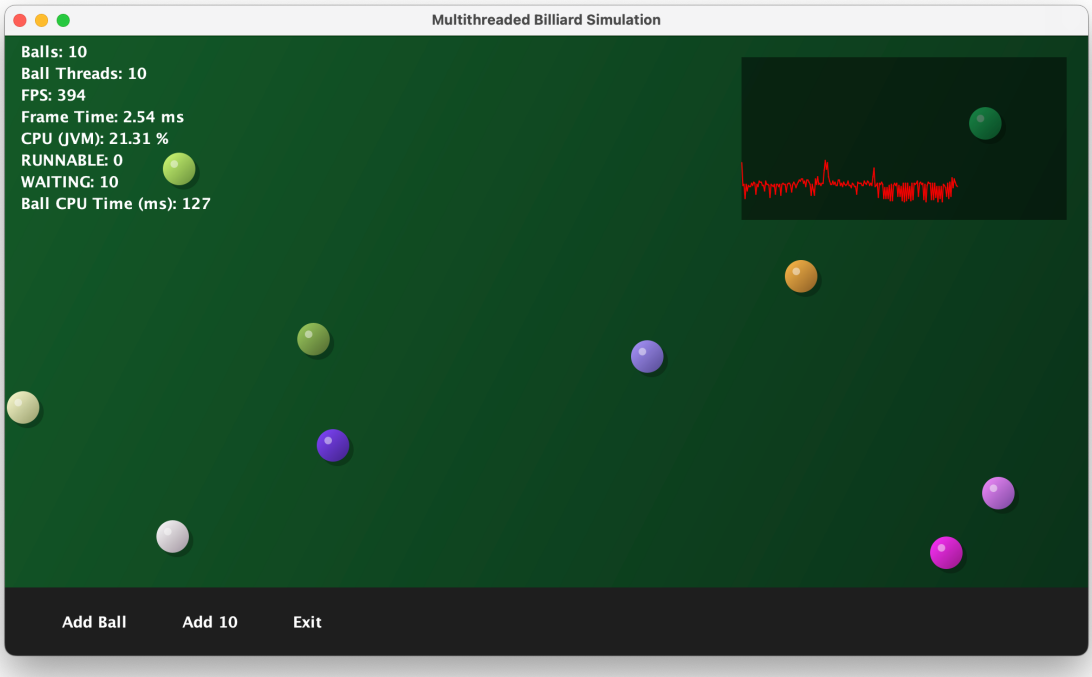

Рисунок 1.1 – Інтерфейс програми із невеликою кількістю кульок

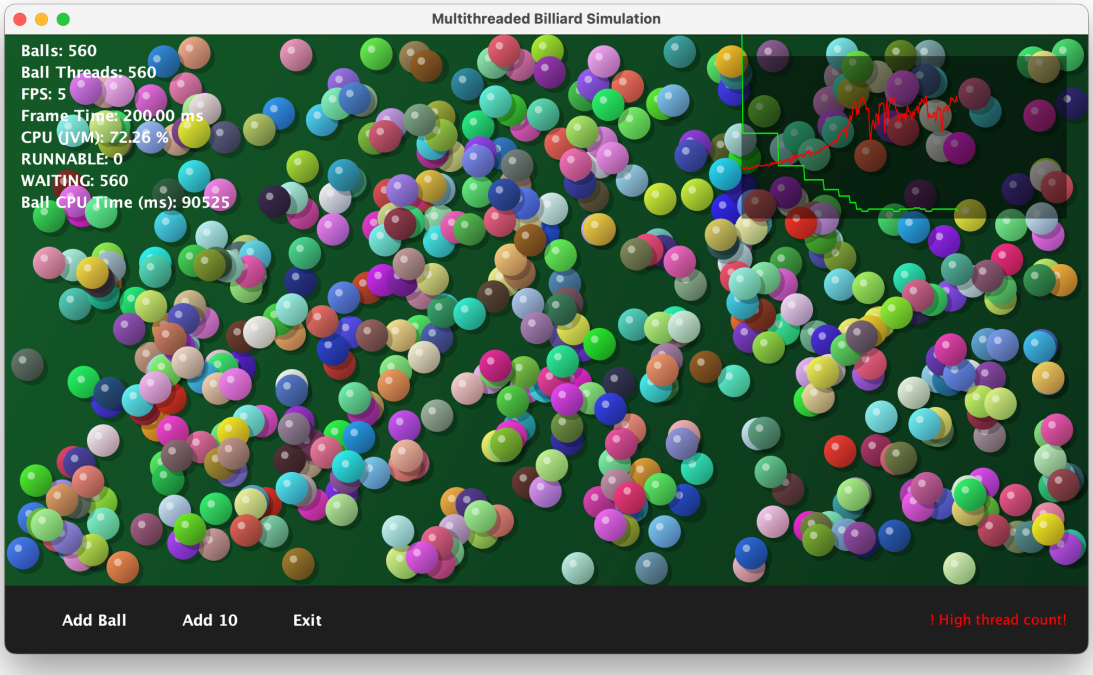

Рисунок 1.2 – Перевірка впливу кількості кульок на продуктивність

При збільшенні кількості кульок спостерігається наступне: зростає кількість одночасно активних потоків; навантаження на CPU (червона лінія на графіку) збільшується; FPS (зелена лінія) падає, тобто програма починає працювати повільніше. При великій кількості кульок (десятки та більше) інтерфейс може стати менш плавним.

Продуктивність програми зменшується при більшій кількості кульок тому, що кожна кулька – окремий потік, який періодично переміщує кульку та викликає перемальовування полотна. Більше кульок – більше потоків, більше контекстних перемикань і навантаження на CPU, а також частіші виклики repaint(). Перемикання між потоками та оновленням GUI обмежує швидкодію, тому FPS знижується.

Окремий потік на кожну кульку дає змогу відтворювати рух паралельно: кульки рухаються одночасно, а не по черзі. Код руху однієї кульки залишається простим (цикл переміщень із затримкою), без явної черги завдань.

Програма коректно реалізує імітацію руху більярдних кульок у окремих потоках. Спостереження підтверджують, що збільшення кількості кульок веде до зростання навантаження на CPU та зниження FPS через збільшення числа потоків і частоту оновлень інтерфейсу.

Завдання 2

Програму модифіковано так, що на полі з’явилися шість луз: чотири в кутах та дві посередині довгих сторін стола. Лузи відмальовуються як темні кола. Після кожного кроку руху кульки в BallThread.run() викликається перевірка canvas.isBallInPocket(ball): якщо центр кульки потрапляє в межах радіусу лузи, кулька вважається потрапившою. У такому разі викликається canvas.pocketBall(ball), який у потокобезпечному режимі видаляє кульку зі списку на полі, збільшує лічильник потрапивших у лузу та оновлює текст у інтерфейсі; потік виходить із циклу (break) і після завершення run() викликає canvas.unregisterBallThread(this), тобто відповідний потік коректно завершує роботу. Кількість кульок, що потрапили в лузу, динамічно відображається.

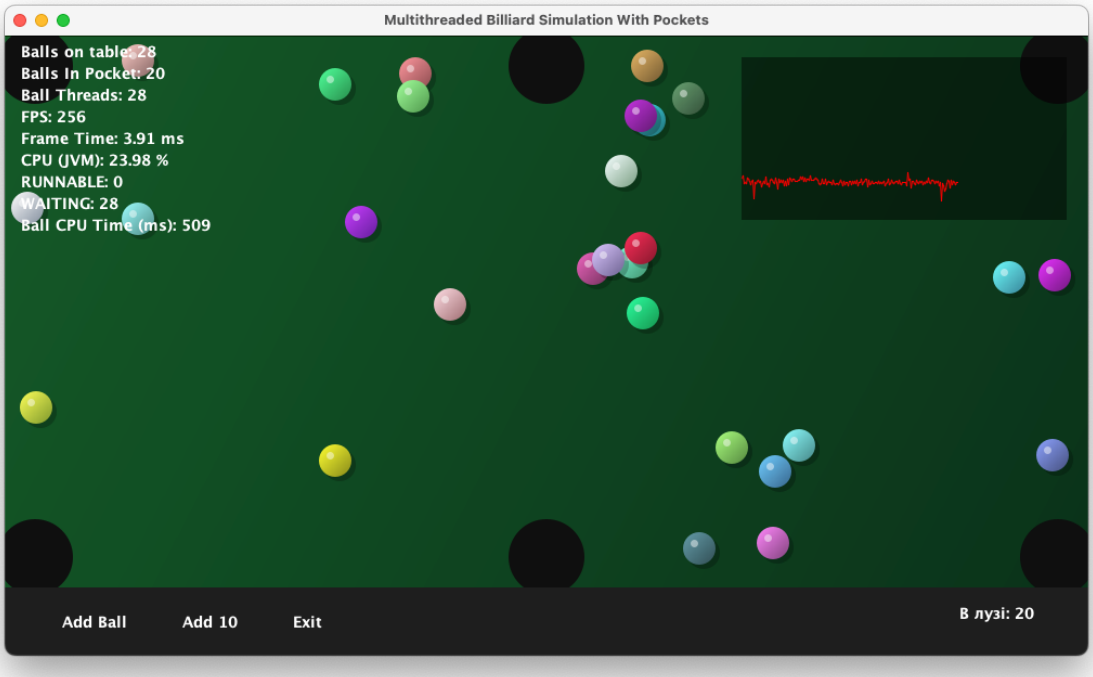

Рисунок 2.1 – Інтерфейс програми із лузами

Завдання 3

Програму модифіковано так, щоб червоні кульки створювалися з вищим пріоритетом потоку (Thread.MAX_PRIORITY), а сині – з нижчим (Thread.MIN_PRIORITY). Введено тип кульки Ball.BallType (RED, BLUE), кожен тип має свій пріоритет потоку; при створенні BallThread йому передається пріоритет і викликається setPriority(priority) перед start(). У інтерфейсі додано кнопки: «Add Red», «Add Blue», «Add 10 Blue», а також режим експерименту «Experiment: 1 Red + N Blue» з полем N: створюється одна червона та N синіх кульок з одним і тим же початковим місцем (наприклад, лівий край по центру висоти), однаковим напрямком (наприклад, dx=2, dy=0) та однаковою швидкістю; усі потоки запускаються після створення кульок, щоб експеримент був коректним.

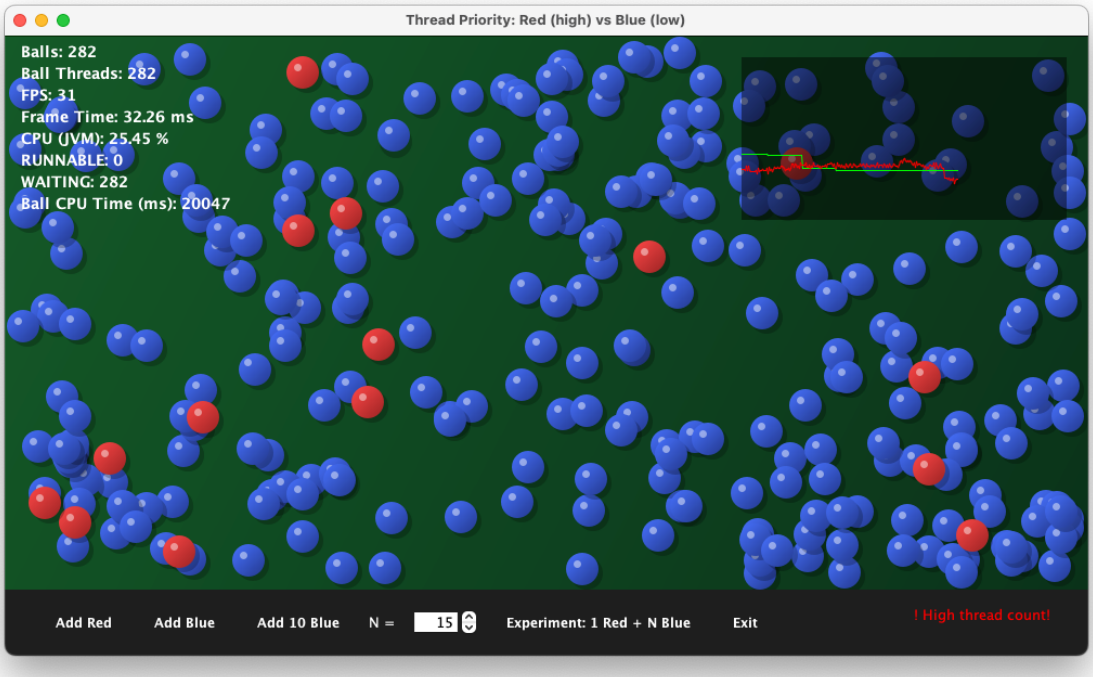

Рисунок 3.1 – Змішана кількість червоних та синіх кульок

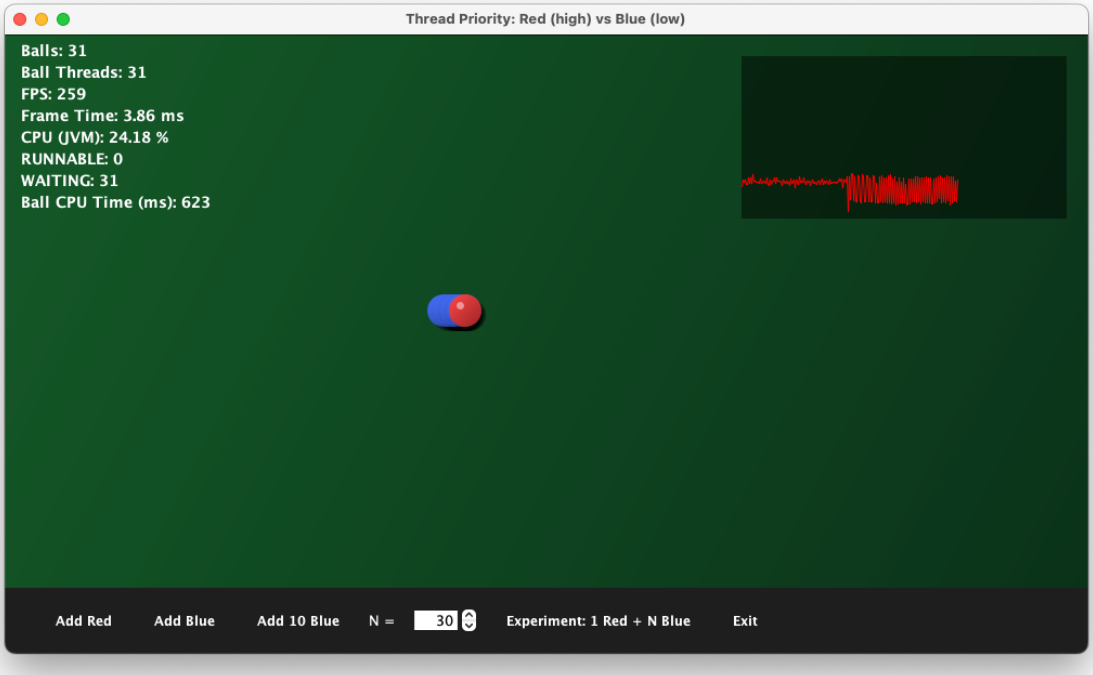

Рисунок 3.2 – Одна червона кулька та невелика кількість синіх

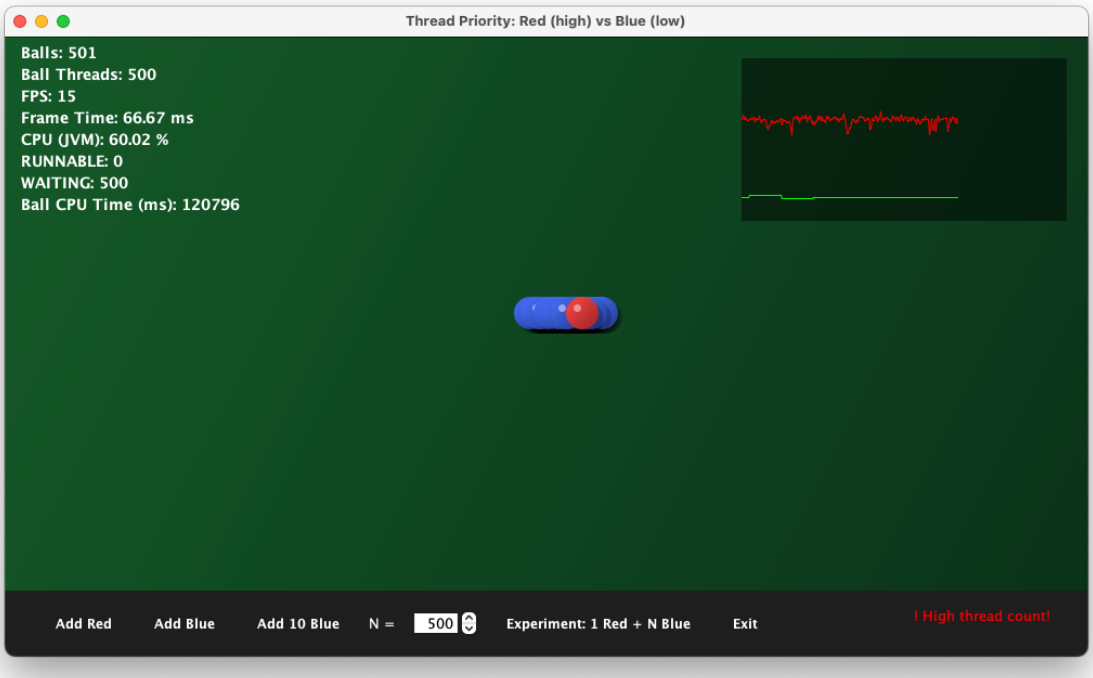

Рисунок 3.3 – Одна червона кулька та велика кількість синіх

При змішаній кількості червоних і синіх кульок червоні часто рухаються плавніше або «випереджають» сині при однаковій логіці руху, оскільки планувальник ОС частіше надає час потокам з вищим пріоритетом. У режимі експерименту (1 червона + багато синіх, однаковий старт): при невеликому N (наприклад, 10–30) червона кулька зазвичай випереджає сині; при великому N (100, 200, 500) сині кульки можуть «розтягуватися» вперед, а червона залишатися в центрі – це нормально: сумарно 200 синіх потоків отримують набагато більше часу CPU, ніж один червоний, тому позиції синіх оновлюються частіше і вони випереджають.

Завдання 4

Було реалізовано демонстрацію методу join() класу Thread через потоки, що відтворюють рух більярдних кульок різного кольору (червона, синя, зелена). Кожна кулька рухається в окремому потоці; потік виконує фіксовану кількість кроків, після чого завершується. Кнопка експерименту запускає допоміжний потік, який:

1. створює червону кульку та її потік і запускає його;
2. викликає redThread.join() — блокується до завершення потоку червоної кульки;
3. після повернення з join() створює синю кульку та її потік і запускає;
4. викликає blueThread.join();
5. аналогічно запускає потік зеленої кульки та викликає greenThread.join().

Рисунок 4.1 – Інтерфейс програми із кнопкою запуску експерименту join()

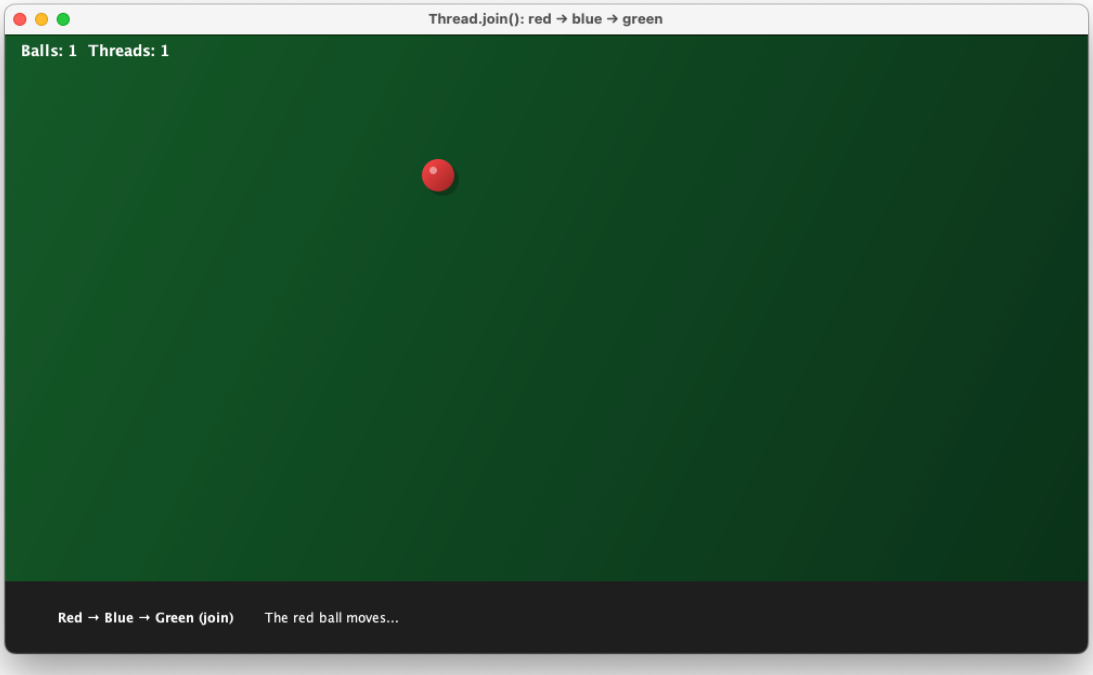

Рисунок 4.2 – Рух червоної кульки

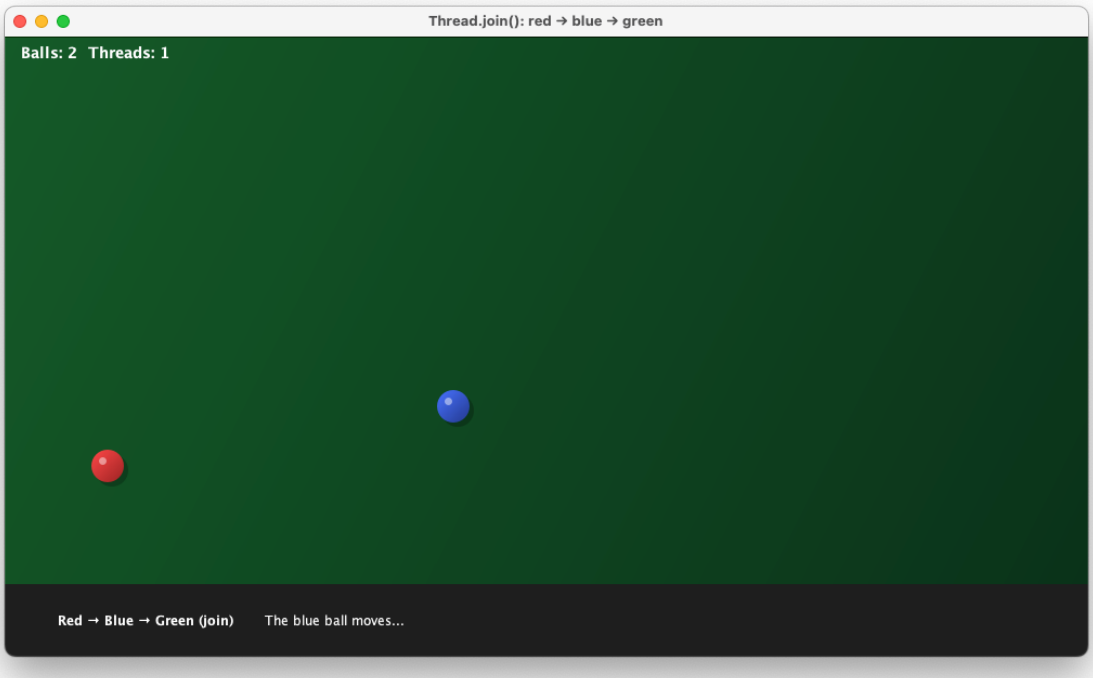

Рисунок 4.3 – Червона кулька завершила рух, синя створилась та почала рух

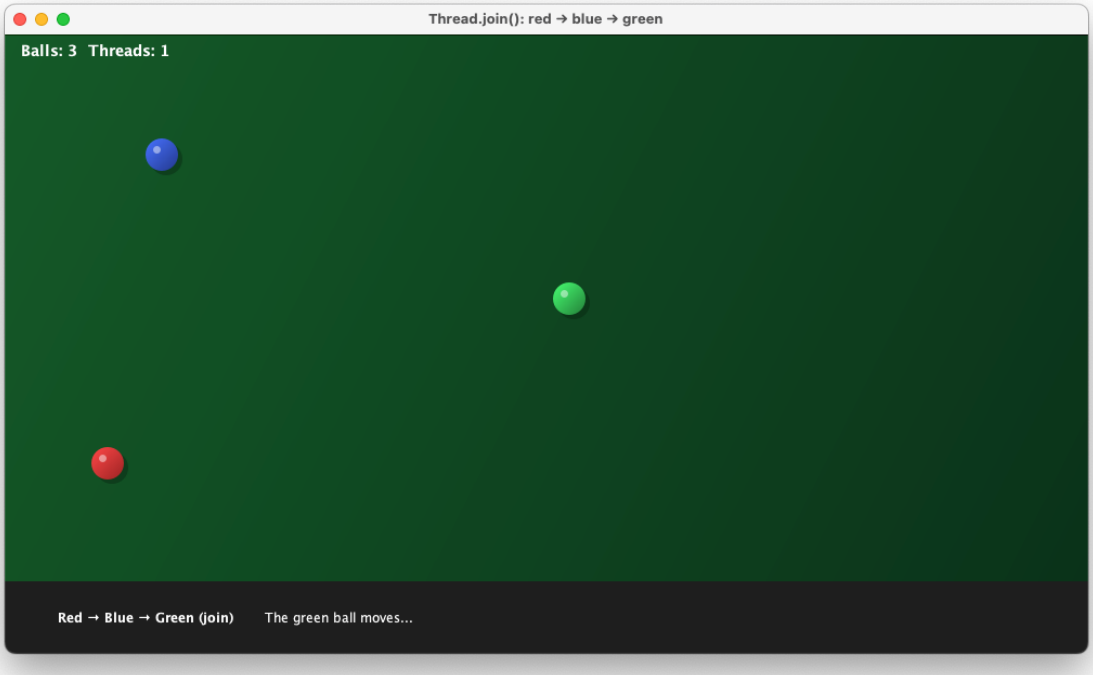

Рисунок 4.4 – Синя кулька завершила рух, зелена створилась та почала рух

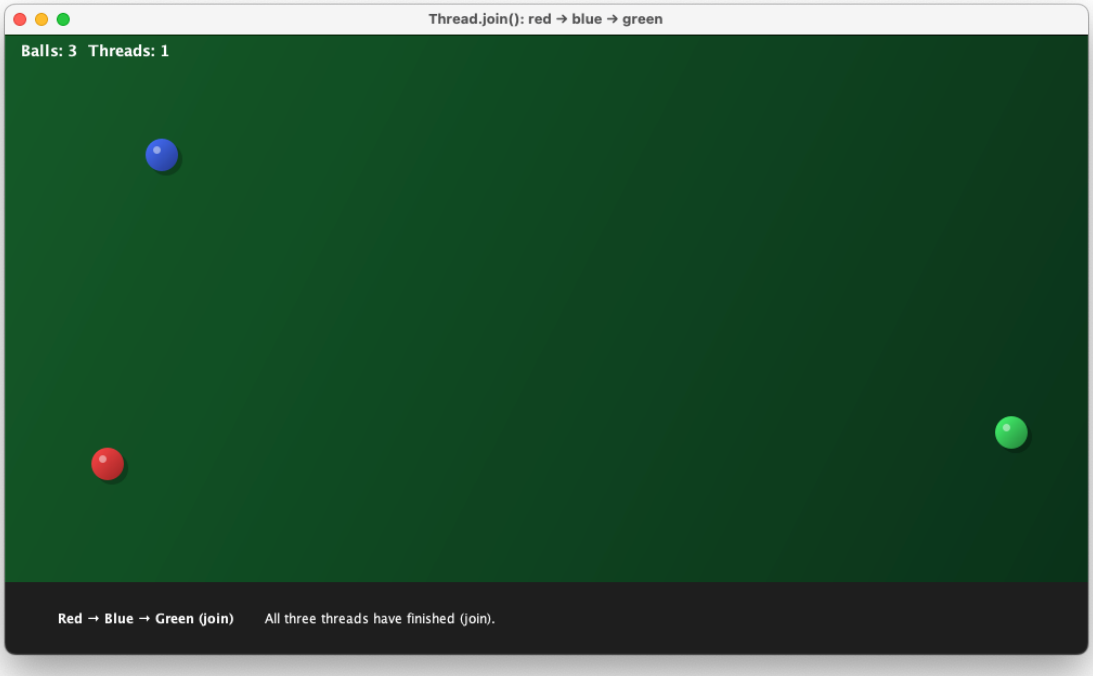

Рисунок 4.5 – Зелена кулька завершила рух, кінець експерименту

Спочатку на полі рухається лише червона кулька. Після того як її потік завершується (вичерпано кількість кроків), з’являється синя кулька і рухається сама. Після завершення потоку синьої з’являється зелена. Тобто кульки рухаються послідовно, а не одночасно: наступна починає рух лише після завершення попередньої. Це безпосередньо демонструє сенс join(): поточний потік (тут – допоміжний) чекає завершення викликаного потоку, перш ніж продовжити виконання. Без join() усі три потоки були б запущені одразу і кульки рухались би паралельно.

Метод join() дозволяє одному потоку дочекатися завершення іншого. У експерименті це реалізовано як послідовний запуск потоків кульок: червона –> синя –> зелена; спостережувана послідовність руху підтверджує, що наступний потік стартує лише після повернення з join().

Завдання 5

Було створено клас Counter з методами increment() та decrement(), які збільшують і зменшують значення лічильника. Клас CounterDemo запускає два потоки: один викликає increment() 100000 разів, інший – decrement() 100000 разів; потоки стартують одночасно, основний потік чекає їх завершення через join() і виводить остаточне значення. Очікуваний коректний результат: 0.

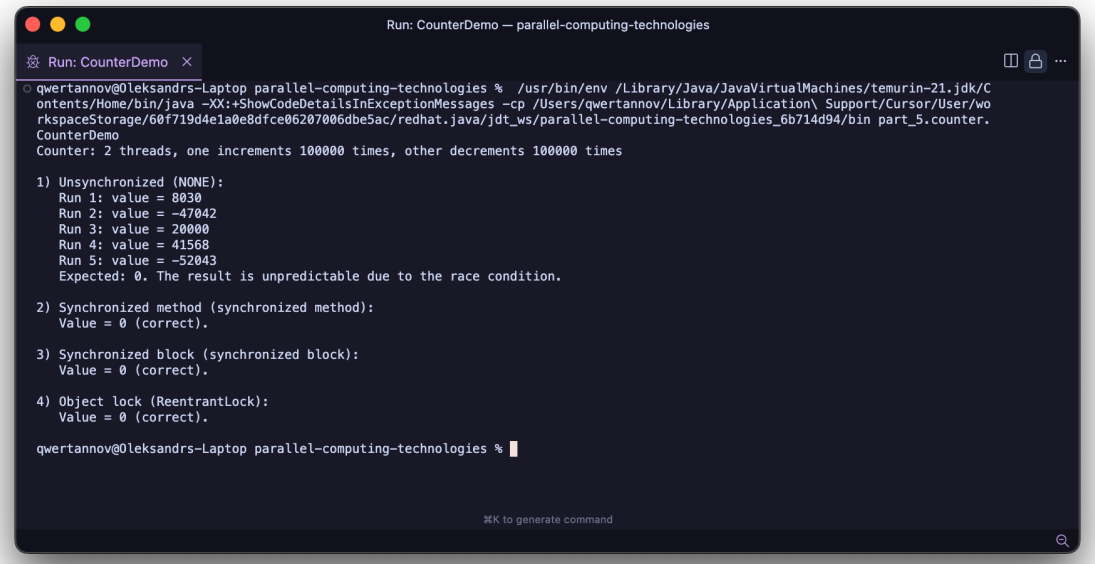

Рисунок 5.1 – Запуск експерименту із лічильником

При режимі Counter.SyncType.NONE операції value++ та value-- не є атомарними (читання – зміна – запис можуть переплітатися між потоками). Спостерігаються різні остаточні значення при повторних запусках. Це гонка потоків (race condition): кілька потоків одночасно змінюють спільний стан без координації, наслідок залежить від порядку виконання і є непередбачуваним.

Опис способів синхронізації:

1. Синхронізований метод. Методи increment() та decrement() викликають допоміжні incrementSyncMethod() та decrementSyncMethod(), позначені як synchronized. Монітор – об’єкт his (сам лічильник). Одночасно виконуватиметься лише один із цих методів для даного об’єкта, тому результат завжди 0.

2. Синхронізований блок. Усередині increment() / decrement() використовується блок synchronized (this) { value++; } або synchronized (this) { value--; }. Монітор також this. Ефект той самий, що й у синхронізованого методу, але критична ділянка обмежена одним виразом; решта коду (наприклад, перемикання за типом) може виконуватися поза блоком.

3. Блокування об’єкта. Використовується ReentrantLock: перед зміною значення викликається lock.lock(), після – lock.unlock() у блоці finally. Явне блокування дає гнучкість (наприклад, умовні блокування, спроба захоплення з таймаутом); потік, що володіє замком, може повторно його захопити (reentrant). Результат так само коректний – 0.

Усі три способи забезпечують взаємне виключення і правильний результат. Синхронізований метод – найпростіший у написанні, але блокує весь метод. Синхронізований блок дозволяє мінімізувати критичну ділянку і зменшити час утримання монітора. ReentrantLock дає більше можливостей (умовні черги, fair lock, tryLock) і може бути ефективнішим у складних сценаріях; необхідно не забувати викликати unlock() у finally.

Висновок

У межах комп’ютерного практикуму №1 отримано практичні навички багатопоточної розробки мовою Java та досліджено основні механізми роботи з потоками. Реалізовано п’ять частин завдань.

У частині 1 побудовано програму імітації руху більярдних кульок, де кожна кулька виконується в окремому потоці; перевірено вплив кількості потоків на навантаження CPU та FPS і сформульовано переваги потокової архітектури. У частині 2 додано лузи, завершення потоку при потраплянні кульки в лузу та динамічне відображення лічильника в інтерфейсі. У частині 3 досліджено пріоритет потоків: червоні та сині кульки створюються з різними пріоритетами; спостереження показали, що пріоритет дає лише ймовірнісну перевагу і при великій кількості потоків його вплив зменшується. У частині 4 проілюстровано метод join(): послідовний запуск потоків кульок (червона –> синя –> зелена) наочно демонструє очікування завершення одного потоку перед стартом наступного. У частині 5 створено клас Counter і продемонстровано гонку потоків при одночасному збільшенні та зменшенні лічильника двома потоками; реалізовано три способи синхронізації (синхронізований метод, синхронізований блок, блокування об’єкта ReentrantLock) і порівняно їх.

Підсумовуючи, виконані завдання охоплюють створення та запуск потоків, завершення потоку за подією, використання пріоритетів, координацію потоків за допомогою join() та забезпечення потокобезпеки спільного стану за допомогою синхронізації. Отримані навички є основою для подальшої розробки багатопоточних та розподілених програм.