**Cells0:  
Пояснення Коду**

1. **Простір імен та using**: Програма знаходиться в просторі імен Cells0 і використовує стандартні бібліотеки для роботи з потоками та введенням/виведенням.
2. **Клас Cells0**: Головний клас програми, який моделює процес без синхронізації.
3. **Змінні Класу**:
   * n: кількість клітинок у кристалі.
   * k: кількість атомів домішок.
   * p: ймовірність переходу вліво.
   * TIME\_UNIT\_MS: часова одиниця в мілісекундах (100 мс).
   * cells: масив, який представляє клітинки кристалу. cells[i] містить кількість атомів у клітинці i.
   * running: флаг, який використовується для зупинки потоків після завершення моделювання.
   * atomTasks: масив задач (Task), які відповідають за виконання потоків атомів.
4. **Метод Main**:
   * Перевіряє, чи передано 3 аргументи командного рядка (N, K, p).
   * Створює екземпляр класу Cells0 та запускає моделювання.
5. **Конструктор Cells0**:
   * Парсує аргументи командного рядка та ініціалізує змінні n, k, p.
   * Перевіряє коректність введених значень.
   * Ініціалізує масив cells та встановлює всі атоми в першу клітинку (cells[0] = k).
6. **Метод StartSimulation**:
   * Виводить інформацію про початок моделювання.
   * Створює та запускає потоки для кожного атома домішки за допомогою Task.Run(() => ParticleRun()).
   * Запускає цикл на 60 секунд, кожну секунду викликаючи PrintSnapshot для виводу стану клітинок.
   * Після завершення циклу встановлює running = false, сигналізуючи потокам про зупинку.
   * Викликає VerifyTotalAtoms для перевірки, чи загальна кількість атомів залишилася незмінною.
7. **Метод GetCell**:
   * Повертає кількість атомів у заданій клітинці i.
   * Використовується для доступу до масиву cells ззовні класу.
8. **Метод MoveParticle**:
   * Виконує переміщення атома з клітинки from до клітинки to.
   * Зменшує кількість атомів у клітинці from та збільшує в клітинці to.
   * **У цій версії метод не використовує синхронізації**, тому можливі гонки даних при одночасному доступі кількох потоків.
9. **Метод PrintSnapshot**:
   * Формує рядок, що представляє стан клітинок на поточний момент часу.
   * Виводить цей рядок у консоль.
10. **Метод VerifyTotalAtoms**:
    * Обчислює загальну кількість атомів у всіх клітинках.
    * Порівнює цю кількість з початковою (k) та виводить результат.
11. **Метод ParticleRun**:
    * Виконується в окремому потоці для кожного атома.
    * Ініціалізує локальний генератор випадкових чисел Random.
    * Початкова позиція атома cell = 0.
    * У нескінченному циклі (поки running == true):
      + Генерує випадкове число m в інтервалі [0,1).
      + Визначає нову позицію newPos залежно від порівняння m з p.
      + Якщо нова позиція виходить за межі кристалу, відбувається "віддзеркалення" — атом залишається на місці.
      + Викликає MoveParticle для переміщення атома.
      + Оновлює поточну позицію атома cell = newPos.
      + Затримка на TIME\_UNIT\_MS мілісекунд.

**Пояснення Коду**

Код **Cells1.cs** майже ідентичний до **Cells0.cs**, з однією ключовою відмінністю — використанням синхронізації для забезпечення коректного доступу до спільного ресурсу cells.

1. **Додаткові Змінні**:
   * lockObj: об'єкт для блокування доступу до масиву cells. Використовується в lock-блоках для забезпечення монопольного доступу до масиву.
2. **Методи з Синхронізацією**:
   * **GetCell**: доступ до елементу масиву cells[i] здійснюється всередині lock (lockObj) блоку, щоб уникнути гонок даних при читанні.
   * **MoveParticle**: переміщення атома здійснюється всередині lock (lockObj) блоку, щоб забезпечити атомарність операцій з масивом cells.
   * **PrintSnapshot**: читання масиву cells для виводу знімку здійснюється всередині lock (lockObj) блоку.
   * **VerifyTotalAtoms**: обчислення загальної кількості атомів здійснюється всередині lock (lockObj) блоку.
3. **Інші Частини Коду**:
   * Решта коду ідентична до **Cells0.cs**, за винятком додавання синхронізації.

**Чому Це Важливо**

* **Відсутність Синхронізації (Cells0)**: У цій версії можливі гонки даних (data races), оскільки кілька потоків можуть одночасно змінювати масив cells. Це може призвести до некоректних результатів, наприклад, негативних значень атомів у клітинках або зміни загальної кількості атомів.
* **Синхронізація (Cells1)**: Використання lock гарантує, що лише один потік може виконувати критичну секцію коду в будь-який момент часу. Це запобігає гонкам даних і забезпечує коректність моделювання.

## Крок 2: Опис Вимог

### Опис Прикладної Задачі

Розглядається процес випадкового блукання (броунівського руху) атомів домішок у одномірному кристалі, який складається з N клітинок. Початково всі K атомів домішок знаходяться в лівій крайній клітинці. Кожен атом може перейти в сусідню клітинку вліво або вправо з ймовірністю, визначеною порогом p (де 0 ≤ p ≤ 1).

* **Правила Переходу**:
  1. **Генерація Випадкового Числа**: Кожен атом у своєму потоці генерує випадкове число m в інтервалі [0, 1).
  2. **Вибір Напрямку**:
     + Якщо m > p, атом переходить вправо.
     + Якщо m ≤ p, атом переходить вліво.
  3. **Віддзеркалення на Межах**: Якщо атом знаходиться на краю кристалу і намагається вийти за межі, він залишається на місці (віддзеркалюється).
* **Мета Моделювання**: Спостерігати за розподілом атомів домішок по клітинках кристалу протягом часу.

### Завдання

1. **Написати Програму Моделювання**:
   * Використовувати мову програмування C# або Java.
   * Кожному атому домішки відповідає окремий потік виконання.
   * Програма має моделювати поведінку атомів у кристалі протягом 1 хвилини.
   * Результатом роботи має бути набір "моментальних знімків" стану кристалу з дискретністю 1 секунда.
2. **Провести Дослідження**:
   * Проаналізувати, як гранулярність блокувальних операцій впливає на швидкодію програми.
   * Порівняти дві версії програми:
     + **Некоректна** (Cells0): без використання блокувань.
     + **Коректна** (Cells1): з використанням блокувань (синхронізації).

### Порядок Виконання

1. **Вивчити Організацію Паралельних Процесів**:
   * Розібратися з механізмами багатопотокового програмування у вибраній мові.
   * Для Java: ознайомитися з класом Thread, інтерфейсом Runnable, ключовим словом synchronized.
2. **Написати Програму**:
   * **Масив cells**: представляє кристал, де cells[i] — кількість атомів у клітинці i.
   * **Потоки**: кожен атом — окремий потік, який змінює свій індекс у масиві cells.
   * **Головний Метод main**:
     + Ініціалізує масив cells.
     + Запускає потоки атомів.
     + Чекає 1 хвилину.
     + Зупиняє потоки та завершує роботу.
   * **Перевірка**:
     + Після завершення моделювання обчислити загальну кількість атомів і переконатися, що вона не змінилася.
3. **Написати Дві Версії Програми**:
   * **Cells0**: без використання блокувань (некоректна версія).
   * **Cells1**: з використанням блокувань (lock у C#, synchronized у Java) для забезпечення коректності.
4. **Метод getcell(i)**:
   * У головних класах визначити публічний метод getcell(i), який повертає кількість атомів у клітинці i.
5. **Параметри Запуску**:
   * Параметрами методу main повинні бути N, K і p.

## Крок 3: Розмовний Опис Предмету та Його Відношення до Завдання

### Вступ

Уявіть собі одномірний кристал, який складається з ряду клітинок, наприклад, як намистинки на нитці. У цьому кристалі знаходяться атоми домішок, які можуть переміщуватися між клітинками, подібно до того, як люди рухаються по вулиці.

### Броунівський Рух у Кристалі

Атоми домішок підпорядковуються випадковому блуканню — броунівському руху. Це означає, що їхнє переміщення непередбачуване і визначається випадковими факторами. В нашому випадку, кожен атом вирішує, куди рухатися, генеруючи випадкове число і порівнюючи його з певним порогом ймовірності p.

* Якщо випадкове число більше за p, атом рухається вправо.
* Якщо менше або дорівнює p, атом рухається вліво.

### Краї Кристалу та Віддзеркалення

Коли атом досягає краю кристалу і намагається рухатися далі, він не може покинути кристал. Замість цього він "віддзеркалюється" і залишається на місці. Це схоже на те, як м'ячик відскакує від стіни.

### Моделювання за Допомогою Багатопотокового Програмування

Щоб змоделювати цей процес, ми використовуємо багатопотокове програмування:

* **Потоки**: Кожен атом домішки моделюється окремим потоком. Це означає, що всі атоми діють незалежно один від одного і одночасно.
* **Спільний Ресурс**: Масив cells представляє кристал і є спільним ресурсом для всіх потоків. Всі потоки можуть читати та змінювати цей масив.

### Проблема Гонок Даних

Коли кілька потоків одночасно змінюють спільний ресурс без синхронізації, виникають **гонки даних**. Це призводить до некоректних результатів, таких як негативна кількість атомів у клітинці або зміна загальної кількості атомів.

### Синхронізація як Рішення

Використання механізмів синхронізації, таких як lock у C#, дозволяє уникнути гонок даних:

* **Блокування**: Коли потік входить у критичну секцію коду (наприклад, змінює масив cells), він блокує доступ для інших потоків до цієї секції. Інші потоки чекають, поки блокування не буде знято.
* **Коректність**: Це гарантує, що операції з масивом cells виконуються атомарно і дані залишаються цілісними.

### Мета Завдання

* **Практичне Розуміння**: Завдання допомагає зрозуміти, як працює багатопотокове програмування і чому синхронізація є важливою.
* **Аналіз Впливу Синхронізації**: Порівнюючи дві версії програми (з синхронізацією та без), можна побачити, як синхронізація впливає на коректність і швидкодію програми.

### Відношення до Розробки Програмного Забезпечення

* **Багатопоточність**: У сучасних додатках часто використовується багатопоточність для підвищення продуктивності та ефективності.
* **Синхронізація**: Розуміння механізмів синхронізації є критично важливим для написання надійного та безпечного програмного забезпечення.
* **Практичні Навички**: Це завдання допомагає набути практичних навичок у роботі з потоками, синхронізацією та виявленні та усуненні гонок даних.

### Підсумок

Ця лабораторна робота не лише демонструє теоретичні концепції багатопотокового програмування, але й надає практичний досвід у розв'язанні реальних проблем, пов'язаних з паралельним доступом до спільних ресурсів. Це важливий крок у підготовці кваліфікованих спеціалістів у галузі програмування та розробки програмного забезпечення.

## **Cells2.cs** Пояснення Коду

### 1. **Загальна Структура**

Код **Cells2.cs** базується на попередніх версіях, але з ключовою відмінністю — використанням блокування на рівні окремих клітинок замість блокування всього масиву.

### 2. **Додаткові Змінні**

* **cellLocks**: масив об'єктів object[], де кожен елемент cellLocks[i] використовується для блокування доступу до клітинки cells[i].

csharp

Копировать код

// Масив об'єктів для блокування кожної клітинки

private object[] cellLocks;

* У конструкторі Cells2 ми ініціалізуємо цей масив:

csharp

Копировать код

this.cellLocks = new object[n];

for (int i = 0; i < n; i++)

{

cellLocks[i] = new object();

}

### 3. **Метод** MoveParticle

У цьому методі ми блокуємо лише ті клітинки, які беруть участь у переміщенні атома.

csharp

Копировать код

public void MoveParticle(int from, int to)

{

// Щоб уникнути deadlock, блокуємо клітинки в порядку зростання індексів

object firstLock, secondLock;

int firstIndex, secondIndex;

if (from < to)

{

firstIndex = from;

secondIndex = to;

}

else

{

firstIndex = to;

secondIndex = from;

}

firstLock = cellLocks[firstIndex];

secondLock = cellLocks[secondIndex];

lock (firstLock)

{

lock (secondLock)

{

cells[from]--;

cells[to]++;

}

}

}

#### ****Уникнення Deadlock****

* **Проблема Deadlock**: Якщо два потоки намагаються одночасно перемістити атоми між двома клітинками в різних напрямках (наприклад, потік A переміщує з клітинки 1 до 2, а потік B — з 2 до 1), і вони блокують клітинки в різному порядку, може виникнути взаємне блокування (deadlock).
* **Рішення**: Завжди блокуємо клітинки в одному і тому ж порядку (за зростанням індексів). Це гарантує, що потоки не будуть чекати один одного, тримаючи заблокованими різні ресурси.

### 4. **Метод** GetCell

При доступі до окремої клітинки ми блокуємо лише її.

csharp

Копировать код

public int GetCell(int i)

{

if (i >= 0 && i < n)

{

lock (cellLocks[i])

{

return cells[i];

}

}

else

throw new IndexOutOfRangeException();

}

### 5. **Метод** PrintSnapshot

При виводі стану всіх клітинок ми блокуємо кожну клітинку окремо під час доступу до неї.

csharp

Копировать код

private void PrintSnapshot(int second)

{

StringBuilder sb = new StringBuilder();

sb.Append($"[{second}s] ");

for (int i = 0; i < n; i++)

{

lock (cellLocks[i])

{

sb.Append($"{cells[i]} ");

}

}

Console.WriteLine(sb.ToString());

}

### 6. **Метод** VerifyTotalAtoms

Аналогічно, при підрахунку загальної кількості атомів ми блокуємо кожну клітинку під час читання.

csharp

Копировать код

private void VerifyTotalAtoms()

{

int total = 0;

for (int i = 0; i < n; i++)

{

lock (cellLocks[i])

{

total += cells[i];

}

}

Console.WriteLine($"Початкова кількість атомів: {k}");

Console.WriteLine($"Кінцева кількість атомів: {total}");

if (total != k)

{

Console.WriteLine("Увага: Загальна кількість атомів змінилася!");

}

else

{

Console.WriteLine("Загальна кількість атомів залишилася незмінною.");

}

}

### 7. **Метод** ParticleRun

Логіка цього методу залишається такою ж, як і в попередніх версіях. Атом генерує випадкове число та вирішує, куди рухатися.

### 8. **Ініціалізація Генератора Випадкових Чисел**

Кожен потік використовує свій власний екземпляр Random, ініціалізований унікальним насінням для запобігання отримання однакових послідовностей випадкових чисел.

csharp

Копировать код

Random random = new Random(Guid.NewGuid().GetHashCode());

## Порівняння З Cells1.cs

### **Cells1.cs** — Блокування Всього Масиву

У версії **Cells1.cs** використовується один об'єкт lockObj для блокування доступу до всього масиву cells. Це означає, що коли один потік змінює будь-яку клітинку, всі інші потоки чекають, навіть якщо вони працюють з іншими клітинками.

csharp

Копировать код

lock (lockObj)

{

// Операції з масивом cells

}

### **Cells2.cs** — Блокування Окремих Клітинок

У версії **Cells2.cs** ми блокуємо лише ті клітинки, з якими працює потік. Це дозволяє іншим потокам одночасно змінювати інші клітинки, підвищуючи паралелізм і потенційно збільшуючи продуктивність.

### **Переваги Блокування Окремих Клітинок**

* **Підвищений Паралелізм**: Потоки можуть одночасно змінювати різні клітинки без взаємного блокування.
* **Зменшення Часу Очікування**: Менша ймовірність того, що потоки будуть чекати один на одного, що може зменшити загальний час виконання.

### **Недоліки Блокування Окремих Клітинок**

* **Складність Реалізації**: Потрібно уважно слідкувати за порядком блокування ресурсів, щоб уникнути deadlock.
* **Ресурси Пам'яті**: Необхідно зберігати додатковий масив об'єктів для блокування.

## Висновок

У **Cells2.cs** ми реалізували блокування на рівні окремих клітинок, що дозволяє підвищити паралелізм у програмі. Цей підхід може бути більш ефективним у випадках, коли кількість клітинок і атомів велика, і необхідно максимізувати продуктивність.

**Примітка**: Важливо завжди забезпечувати уникнення deadlock, особливо при блокуванні декількох ресурсів. У нашому випадку ми блокуємо клітинки в порядку зростання індексів, що гарантує відсутність взаємного блокування потоків.

Пояснення Коду

1. Простір імен та підключення бібліотек

o using System: базові типи даних і функціональність.

o using System.Text: для використання StringBuilder при формуванні рядків.

o using System.Threading: для роботи з потоками та синхронізацією.

o using System.Threading.Tasks: для використання задач (Task) при створенні потоків.

2. Оголошення класу Cells0

o Клас Cells0 містить основну логіку моделювання без синхронізації.

3. Змінні класу

o n: кількість клітинок у кристалі.

o k: кількість атомів домішок.

o p: поріг ймовірності для переходу вліво.

o TIME\_UNIT\_MS: константа, що визначає затримку між переміщеннями атомів (100 мс).

o cells: масив цілих чисел, де cells[i] — кількість атомів у клітинці i.

o running: флаг, що використовується для зупинки потоків після завершення моделювання.

o atomTasks: масив задач (Task), кожна з яких відповідає окремому атому (потоку).

4. Метод Main

o Перевіряє, чи передано рівно три аргументи командного рядка (N, K, p).

o Створює екземпляр класу Cells0 та викликає метод StartSimulation().

5. Конструктор Cells0

o Парсує аргументи командного рядка та ініціалізує змінні n, k, p.

o Перевіряє коректність введених параметрів (n та k повинні бути більше 0, p — між 0 та 1).

o Ініціалізує масив cells розміром n.

o Встановлює cells[0] = k, тобто всі атоми спочатку знаходяться в першій клітинці.

6. Метод StartSimulation

o Виводить інформацію про початок моделювання.

o Створює масив задач atomTasks для кожного атома.

o Запускає кожен атом як окремий потік, викликаючи ParticleRun() у Task.Run().

o У циклі протягом 60 секунд кожну секунду викликає PrintSnapshot() для виводу стану клітинок.

o Після завершення циклу встановлює running = false, що сигналізує потокам про необхідність зупинки.

o Викликає Task.WaitAll(atomTasks), щоб дочекатися завершення всіх потоків.

o Виводить повідомлення про завершення моделювання та викликає VerifyTotalAtoms().

7. Метод GetCell

o Повертає кількість атомів у клітинці з індексом i.

o Використовується для доступу до масиву cells ззовні класу.

8. Метод MoveParticle

o Виконує переміщення атома з клітинки from до клітинки to.

o Зменшує cells[from] на 1 та збільшує cells[to] на 1.

o У цій версії немає синхронізації, тому можливі гонки даних при одночасному доступі кількох потоків до масиву cells.

9. Метод PrintSnapshot

o Формує рядок, що містить номер секунди та кількість атомів у кожній клітинці.

o Використовує StringBuilder для ефективного формування рядка.

o Виводить сформований рядок у консоль.

10. Метод VerifyTotalAtoms

o Обчислює суму атомів у всіх клітинках після завершення моделювання.

o Порівнює отриману суму з початковою кількістю атомів k.

o Виводить результат перевірки в консоль.

o Через відсутність синхронізації може статися ситуація, коли загальна кількість атомів змінюється (може стати меншою або більшою за k).

11. Метод ParticleRun

o Метод, що виконується в окремому потоці для кожного атома.

o Ініціалізує свій екземпляр генератора випадкових чисел Random з унікальним насінням (Guid.NewGuid().GetHashCode()), щоб уникнути однакових послідовностей випадкових чисел у різних потоках.

o Змінна cell зберігає поточну позицію атома (починається з 0).

o У циклі (поки running == true):

 Генерує випадкове число m в інтервалі [0,1).

 Визначає нову позицію newPos залежно від порівняння m з p:

 Якщо m > p, рух вправо (cell + 1).

 Інакше, рух вліво (cell - 1).

 Якщо newPos виходить за межі масиву (менше 0 або більше n-1), атом залишається на місці (віддзеркалення).

 Викликає MoveParticle(cell, newPos) для переміщення атома.

 Оновлює cell = newPos.

 Виконує затримку на TIME\_UNIT\_MS мілісекунд.

,,

,,,,

Пояснення Коду

Код Cells1.cs схожий на Cells0.cs, але з додаванням механізмів синхронізації для забезпечення коректного доступу до спільного ресурсу cells.

1. Додаткові Змінні

o lockObj: об'єкт для блокування доступу до масиву cells. Використовується в lock-блоках для забезпечення монопольного доступу до критичних секцій коду.

2. Метод GetCell

o Тепер містить блок lock (lockObj), який забезпечує синхронізований доступ до масиву cells при читанні.

3. Метод MoveParticle

o Огорнутий у блок lock (lockObj), що гарантує, що операції зменшення та збільшення значень у масиві cells виконуються атомарно.

4. Метод PrintSnapshot

o Операції читання масиву cells огорнуті в lock, щоб уникнути можливих гонок даних при виводі стану клітинок.

5. Метод VerifyTotalAtoms

o Обчислення загальної кількості атомів також виконується всередині блоку lock, щоб гарантувати коректність даних.

6. Метод ParticleRun

o Логіка залишилася незмінною, але оскільки метод MoveParticle тепер синхронізований, доступ до масиву cells здійснюється коректно.

Чому Синхронізація Важлива

• Проблема Гонок Даних: У багатопотокових програмах, коли кілька потоків одночасно змінюють спільний ресурс без синхронізації, виникають гонки даних, що призводить до некоректних результатів.

• Рішення: Використання блокування (lock) гарантує, що лише один потік може виконувати код всередині блоку lock в будь-який момент часу.

• Результат: Програма Cells1 забезпечує коректну поведінку, загальна кількість атомів залишається незмінною, і відсутні негативні значення в масиві cells.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Порівняння Cells0 та Cells1

• Без Синхронізації (Cells0):

o Швидша робота через відсутність накладних витрат на блокування.

o Можливі некоректні результати через гонки даних.

o Загальна кількість атомів може змінитися, можуть з'явитися негативні значення в масиві cells.

• З Синхронізацією (Cells1):

o Дещо повільніша робота через використання блокувань.

o Забезпечується коректність даних, відсутність гонок даних.

o Загальна кількість атомів залишається незмінною, всі значення в масиві cells коректні.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Висновок

Обидві програми демонструють важливість використання механізмів синхронізації в багатопотокових додатках. Cells0 показує, які проблеми можуть виникнути при відсутності синхронізації, тоді як Cells1 демонструє, як використання блокування може забезпечити коректність роботи програми, хоча й з деякими накладними витратами на продуктивність.

Ця детальна розбивка коду допоможе краще зрозуміти, як працюють потоки, синхронізація та багатопотокове програмування загалом. Важливо завжди враховувати можливість гонок даних при роботі зі спільними ресурсами та використовувати відповідні механізми для їх запобігання.

Щоб пояснити це просто, можна сказати так:

Уяви собі, що є кілька частинок (атомів), які знаходяться в ряді клітинок, схожих на комірки на папері. Ці частинки постійно рухаються вперед або назад випадковим чином. У кожен момент часу кожна частинка може вирішити, куди піти: вліво або вправо. Це рішення залежить від випадкового числа, яке генерується для кожної частинки.

Метою є змоделювати цей рух частинок за допомогою програми, яка працює на кількох потоках одночасно, щоб одночасно стежити за рухом великої кількості частинок. Частинки не можуть виходити за межі рядка клітинок — якщо вони досягають краю, то відбиваються назад, як від стіни.

Таким чином, ми можемо бачити, як частинки хаотично рухаються по рядку клітинок, і цей процес називається броунівським рухом.

Завдання 1:

Потрібно створити програму, яка показує, як частинки рухаються по одновимірному простору (кристалу). Кожна частинка буде мати свій власний потік (це як окрема "робота" для кожної частинки, яка виконується паралельно). Мета — спостерігати, як змінюється розподіл частинок у кристалі протягом однієї хвилини, показуючи їх стан кожну секунду. Вихідні дані мають бути у вигляді «знімків» стану частинок — де вони знаходяться в певний момент.

Завдання 2:

Необхідно дослідити, як різні підходи до блокування (обмеження доступу до ресурсів для потоків) впливають на швидкість програми. Один варіант — блокувати доступ тільки до однієї клітинки, інший — до всього масиву клітин.

Написання двох версій програми:

Перша версія (некоректна): тут не використовується блокування даних (немає захисту для масиву під час доступу до нього різними потоками).

Друга версія (коректна): тут використовується synchronized для забезпечення того, що тільки один потік може змінювати масив у будь-який момент часу.

У цій версії кожна клітинка має свій власний об'єкт для блокування, що дозволяє потокам одночасно змінювати різні клітинки без взаємного блокування всього масиву.