Laboratorio 2 - Programación concurrente, condiciones de carrera, esquemas de sincronización, colecciones sincronizadas y concurrentes

Andrés Felipe Arias Ajiaco

Cesar David Amaya Gómez

Johan Sebastián Gracia Martínez

Sebastián David Blanco Rodríguez

Universidad Escuela Colombiana de ingeniería Julio Gravito

Arquitecturas de software

Ing. Javier Iván Toquica Barrera

9 de febrero de 2024

Introducción

El laboratorio propone dos partes distintas que involucran el manejo de hilos en entornos concurrentes.

En la primera parte, se aborda el control de hilos utilizando mecanismos de sincronización como wait y notify. Se propone una modificación a un programa existente llamado PrimeFinder, el cual calcula números primos concurrentemente distribuyendo la búsqueda entre varios hilos. La modificación consiste en detener todos los hilos cada cierto tiempo y mostrar el número de primos encontrados hasta ese momento. Se hace hincapié en el uso de los mecanismos de sincronización proporcionados por Java, como wait, notify y notifyAll.

En la segunda parte, se analiza un juego llamado SnakeRace. Se analiza el código para entender cómo se utiliza la concurrencia y los hilos para crear un comportamiento autónomo de las serpientes en el juego. Se pide identificar posibles condiciones de carrera, uso inadecuado de colecciones concurrentes. Además, se ajusta para permitir que el juego pueda ser iniciado, pausado y reanudado a través de botones en la interfaz, mostrando información relevante como la serpiente viva más larga y la primera serpiente en morir.

Desarrollo del laboratorio

PARTE 1 - Control de hilos con wait/notify.

En esta parte se sobre escribe el método de run poniéndole una nueva funcionalidad para detener los hilos además imprime la cantidad de números encontrados por cada hilo, después de un tiempo establecido el programa se detiene muestra los resultados encontrados hasta le momento y para continuar el usuario tiene que oprimir la tecla “Enter” para lograr esto se utilizó el syncrhonized y notifyall.

```
public void run() {

    for(PrimeFinderThread thread :pft){
        thread.start();
    }
    Timer timer = new Timer();
    timer.schedule(() -> {
            for (PrimeFinderThread thread : pft) {
                thread.stopThread(true);
            }
            for (PrimeFinderThread thread : pft) {
                System.out.println(
                        "El hilo: " + thread.getName() + " Encontro " + thread.getPrimesQuantity() + " Primos.");
            }
            System.out.println("Presione enter para continuar. ");
            String read;
            Scanner scanner = new Scanner(System.in);
            read = scanner.nextLine();
            if (read != null) {
                scanner.close();
                System.out.println("Continuando Busqueda...");
                synchronized (lock) {
                    for (PrimeFinderThread thread : pft) {
                        thread.stopThread(false);
                    }
                    lock.notifyAll();
                }
            }
    },TMILISECONDS);
```



```
    },TMILISECONDS);

    try {
        for (PrimeFinderThread thread : pft) {
            thread.join();
        }
        for (int j = 0; j < pft.length; j++) {
            PrimeFinderThread thread = pft[j];
            System.out.println("La cantidad de numero primos encontrados por el hilo:" + thread.getName() + "fue:" + thread.getPrimesQuantity());
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
  }
}
```

Se sobre escribió el método run de la clase primeFinderTrhead agregándole una funcionalidad que notifica a los hilos que se deben detener usando syncrhonized y wait.

```java
    @Override
public void run(){
        for (int i= a;i < b;i++){
            if (isPrime(i)){
                primes.add(i);
                totalPrimes.incrementAndGet();
                //System.out.println(i);
            }
            if(stop){
                synchronized (lock){
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        }
}
```

Aquí podemos observar los resultados que nos da el programa con 3 hilos y un numero de 0 a 30.000.000 y el correcto funcionamiento del tiempo dado.

```
El hilo: Thread-1 Encontro 371886 Primos.
El hilo: Thread-2 Encontro 170143 Primos.
El hilo: Thread-3 Encontro 124748 Primos.
Presione enter para continuar.

Continuando Busqueda...
La cantidad de numero primos encontrados por el hilo:Thread-1fue:664579
La cantidad de numero primos encontrados por el hilo:Thread-2fue:606028
La cantidad de numero primos encontrados por el hilo:Thread-3fue:587252
```

```
Cantidad de números primos en el rango de 0 a 30 millones: 1857859
```

PARTE 2 – SnakeRace

2.1 Las serpientes están definidas como un vector de hilos, por lo cual se pueden generar n cantidad de serpientes con movimientos independientes

relacionando un identificador como lo puede ser en este caso la Cabeza de la serpiente y un método como randomMovement el cual genera una dirección

aleatoria para cada serpiente desde el constructor.

2.2

- Condiciones de carrera:

    a. Las serpientes al tener movimiento a una dirección aleatoria se debe evitar que estas puedan moverse sobre su propio cuerpo ya que esto no es posible en la práctica

    b. Es posible que 2 o más serpientes se encuentren en direcciones iguales y alcancen al mismo tiempo un objeto dentro del tablero lo que causaría una colisión y se debe definir a qué serpiente se le asigna la ventaja de la celda

- Uso inadecuado de colecciones:

    El "cuerpo" de la serpiente se encuentra agrupado en un Linkedlist lo que dificulta su desplazamiento

- Uso innecesario de esperas activas:

    El estado de una serpiente se ve definido por medio de un condicional dentro del método de ejecución del hilo, lo cual hace que el método realice comprobaciones indeterminadas hasta que se cumpla esta condición.

3

- Es necesario poner un estado sobre las celdas que se encuentran ocupadas para que así las serpientes no puedan moverse sobre su propio cuerpo

- con el uso de bloques sincronizados se logra definir un "orden de llegada" para cada hilo que representan las serpientes, con el fin de asignarle la ventaja a solo una serpiente

4

En esta parte se implementaron los botones de “Iniciar”, “Parar” y “reanudar” los cuales realizan sus respectivas acciones correctamente, se definen los tres métodos

```
public void stop(){
    int snakeLenght = 0;
    Snake largestSnake = null;
    for(Snake s: snakes){
        s.setStop(true);
        int actualSnake = s.getBody().size();
        if( actualSnake > snakeLenght){
            snakeLenght = actualSnake;
            largestSnake = s;
        }
    }
    board.largestSnake = largestSnake;
    board.setStop(true);
    board.repaint();
}

1 usage  new *
public void resume(){
    synchronized (lock){
        for(Snake s: snakes){
            s.setStop(false);
        }
        lock.notifyAll();
    }
    board.setStop(false);
}

1 usage  new *
public void start(){
    for(Thread t: thread){
        t.start();
    }
}
```

```
1 usage  new *
public void prepareActionsMenu(){

    JPanel actionsBPabel=new JPanel();
    GridLayout gridLayout = new GridLayout( rows: 1, cols: 3);

    JButton start = new JButton( text: "Start");
    start.addActionListener(e -> {
        start();
        start.setEnabled(false);
    });

    JButton stop = new JButton( text: "Stop");
    stop.addActionListener(e -> stop());

    JButton resume = new JButton( text: "Resume");
    resume.addActionListener(e -> resume());

    actionsBPabel.setLayout(gridLayout);
    actionsBPabel.add(start);
    actionsBPabel.add(stop);
    actionsBPabel.add(resume);

    frame.add(actionsBPabel,BorderLayout.SOUTH);
}
```

Se sobre escribe el método de run para que la función de “stop” funcione utilizando el syncrhonized y wait

```
@Override
public void run() {
    while (!snakeEnd) {

        snakeCalc();

        //NOTIFY CHANGES TO GUI
        setChanged();
        notifyObservers();

        try {
            if (hasTurbo == true) {
                Thread.sleep( millis: 500 / 3);
            } else {
                Thread.sleep( millis: 500);
            }
            if(stop){
                synchronized (lock){
                    lock.wait();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

    fixDirection(head);
}
```

Se modifico el método de paintComponent que se ejecuta continuamente para poder representar la serpiente más larga y la primera que se estrella

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    drawGrid(g);
    drawSnake(g);
    drawFood(g);
    drawBarriers(g);
    drawJumpPads(g);
    drawTurboBoosts(g);
    if(stop){
        drawSnakeBody(g);
    }
    getFirstDiedSnake();
    if(crashed && stop){
        drawCrashedSnaked(g);
    }
}
```

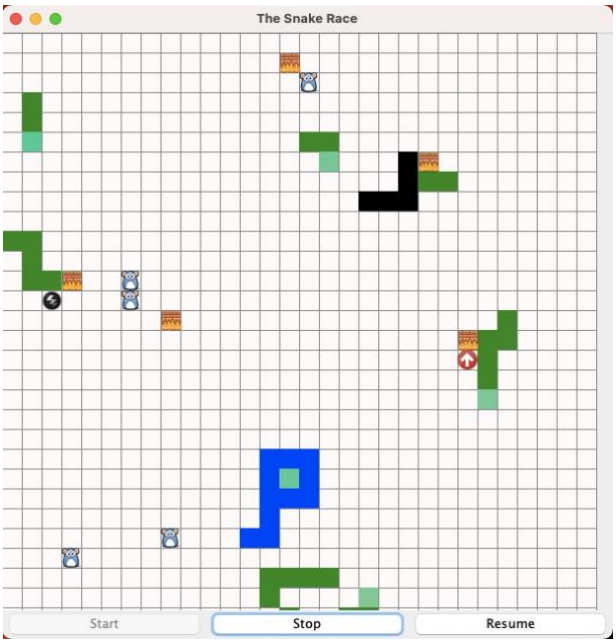
Se crean los métodos para poder representar las serpientes, en color azul la más larga y en color negro la primera que muere.

```
1 usage  new *
public Snake getFirstDiedSnake(){
    Snake snakes[] =  SnakeApp.getApp().snakes;
    if(firstCrashedSnake[0] == null){
        for(Snake s: snakes){
            if(s.isSnakeEnd()){
                firstCrashedSnake[0] = s;
                crashed = true;
            }
        }
    }
    return firstCrashedSnake[0];
}
```

```
    1 usage  new *
    private void drawSnakeBody(Graphics g){
        LinkedList<Cell> body = largestSnake.getBody();
        if(!body.isEmpty()){
            for(int i = 1; i < body.size();i++){
                Cell p = body.get(i);
                g.setColor(new Color( r: 0, g: 0, b: 255));
                g.fillRect( x: p.getX() * GridSize.WIDTH_BOX,  y: p.getY()
                            * GridSize.HEIGH_BOX, GridSize.WIDTH_BOX,
                        GridSize.HEIGH_BOX);
            }
        }
    }

    1 usage  new *
    private void drawCrashedSnaked(Graphics g){
        LinkedList<Cell> body = firstCrashedSnake[0].getBody();
        if(!body.isEmpty()){
            for(int i = 1; i < body.size();i++){
                Cell p = body.get(i);
                g.setColor(new Color( r: 0, g: 0, b: 0));
                g.fillRect( x: p.getX() * GridSize.WIDTH_BOX,  y: p.getY()
                            * GridSize.HEIGH_BOX, GridSize.WIDTH_BOX,
                        GridSize.HEIGH_BOX);
            }
        }
    }
```

A continuación, vemos un ejemple cuando se detiene el juego con el botón de “stop” y podemos observar que la primera serpiente que murió esta de color negro y la más larga de color azul.



Conclusiones

- Se puede identificar durante el desarrollo del laboratorio que el uso de los estados de los hilos es fundamental para la eficiencia en cuanto a disponibilidad de memoria en los procesos para cada hilo lógico que se posea, es importante reconocer la funcionalidad, requerimientos, estructura y manejo de cada tipo de operador para los estados de los hilos, teniendo en cuenta el orden en que son invocados y la lógica que viene detrás de su uso, contando herramientas como: métodos sincronizados, bloques sincronizados, condiciones de guarda, operadores con hilos. Se hace eficiente la distribución de procesos por lo que hay menor consumo y mayor eficiencia en las tareas propuestas.

- Para garantizar la integridad de los datos y ofrecer una experiencia de usuario sin contratiempos, es importante detectar y solucionar situaciones de carrera en aplicaciones concurrentes como SnakeRace1. En otras palabras, abordar estas condiciones de carrera es fundamental para asegurar que los datos se manejen correctamente y que los usuarios disfruten de una interacción fluida con la aplicación.

- Si analizamos el código de SnakeRace podemos observar la importancia de aplicar estrategias de manejo de concurrencia que se ajusten a las necesidades específicas de la aplicación, seleccionar cuidadosamente las estructuras de datos y algoritmos para que los hilos no tengan conflictos y optimizar la eficiencia del programa en un entorno multi-hilo

Bibliografía

Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., & Holmes, D. (2006). Java Concurrency in Practice. http://ci.nii.ac.jp/ncid/BA78043355

ChatGPT. (s. f.). https://chat.openai.com/