|  |
| --- |
| Programowanie współbiezne |
| Laboratium 1: Semafory |
| Figura fraktalna w L-System z semaforami |
|  |
| **Tomasz Trębski – 165535**  **Karol Górecki - 165405** |
| **10/21/2013** |

|  |
| --- |
| **Sprawozdanie na laboratoria z przedmiotu Programowanie współbieżne** |

Table of Contents

[Wprowadzenia 2](#_Toc370066800)

[Teoria 2](#_Toc370066801)

[Współbieżność 2](#_Toc370066802)

[Semafory 2](#_Toc370066803)

[Fraktal 3](#_Toc370066804)

[L-System 3](#_Toc370066805)

[Praktyka 3](#_Toc370066806)

[Wnioski 3](#_Toc370066807)

# Wprowadzenia

Celem zadania było zaimplementowanie współbieżnego programu generującego figurę fraktalną z użyciem semaforów. Do wykonania zadania zostały użyte mechanizmy udostępnione przez standardową bibliotekę Javy dla mechanizmu semaforów, generowanie ścieżki fraktalnej w systemie L-System oraz bibliotekę generującą obraz w podejściu TurtleGraphics (znanym z programu Logo).

# Teoria

## Współbieżność

Współbieżność odnosi się do własności systemów operacyjnych, gdzie możliwe jest równoległe wykonywanie obliczeń i wzajemnych interakcji z miedzy nami. W przypadku procesorów wielordzeniowych możliwe jest uzyskania prawdziwej współbieżności, kiedy to kolejne procesy/wątki w ramach tych procesów, wykonany są na innych rdzeniach procesora. W przypadku procesorów starszej generacji nie było to możliwe, ale takie programy dało się pisać. W tym przypadku współbieżność była emulowana, a czas procesora był dzielony między procesy/wątki.

## Semafory

Semafor jest obiektem, który posiada pewną pulę zezwoleń, która odnosi się do tego, jak wiele razy semafor może zostać „podniesiony”/”opuszczony”. Tak samo jak monitory umożliwiają budowanie sekcji krytycznych – wątek, który uruchomi na semaforze operację P w momencie, gdy semafor nie posiada żadnego pozwolenia zostanie wstrzymany do czasu aż pozwolenie zostanie zwrócone do puli. W Javie semafory reprezentowane są przez obiekty klasy *java.util.concurrent.Semaphore*, a odpowiednikami operacji P i V są odpowiednio metody *acquire()*  oraz *release()*.

W ogólności semafory mogą posiadać dowolną liczbę pozwoleń (stąd określenie semafor ogólny) jednak najczęstszym typem semaforów są semafory posiadające tylko jedno pozwolenie – są to tzw. semafory binarne inaczej zwane mutexami.

Cechy semaforów:

* Reentrant locking[[1]](#footnote-1) – brak
* Timed locking – tak, z użyciem metody **tryAcquire(long timeout, TimeUnit unit)** blokującą wątek do momentu uzyskania blokady, maksymalnie przez pewien okres czasu
* Lock polling – tak, dzięki metodom **availablePermits()**, która zwraca ilość dostępnych jeszcze pozwoleń oraz metody **tryAquire()**
* Condition locking – nie zapewniają
* Shared / exclusive locking – nie zapewniają
* Lock ownership – nie zapewniają – semafory mogą być podnoszone i opuszczane przez różne wątki

## Fraktal

Najprościej opisać można fraktal, jako obiekt **samopodobny**, czyli taki, który w dowolnym miejscu wygląda identycznie.

## L-System

L-System jest to specjalny sposób na przepisywania i zamianę określonych części łańcuchów znakowych w określonym porządku, który określony jest przez alfabet, zbiór stałych i zmiennych. W przykładzie wygenerowaliśmy **Trójkąt Sierpińskiego**, który charakteryzuje się następującymi konfiguracją:

|  |  |
| --- | --- |
| Typ | Wartość |
| Stała | **+** – skręć w lewo o zadany kąt  **-** – skręt w prawo o zadany kąt |
| Zmienne | F – rysuj w przód  L – rysuj w przód |
| Axiom | F |
| Reguły | F>L-F-L  L>F-L-F |
| Kąt |  |
| Zmień wartość kąta na przeciwną, co iterację ? | Tak |

# Praktyka

Część praktyczna została zaimplementowana w języku Java. Za bazę posłużyło abstrakcyjna klasa **FSemaphore**, będąca implementacją interfejsu Runnable[[2]](#footnote-2), w której znajdował się kod pozwalający na uzyskania pozwolenia na dostęp do semafora i dostępu do dzielonego zasobu. Zależnie od implementacji wykonywany był kod obliczający następną ścieżkę lub ja rysujący, odpowiednio klasy **CFPSemaphore** oraz **DFPSemaphore**. W obu przypadkach zasobem, który był dzielony między wieloma instancji wątków była **Double-Ended**-**Queue**.

Dzięki takiemu rozwiązaniu w:

* **Wątku liczącym** – pobierany, ale niezdejmowany, był ostatni element kolejek, stanowiący ziarno do wygenerowania nowej ścieżki, a nowa ścieżka była umieszczona na końcu kolejki.
* **Wątek rysujący** – w momencie uzyskania dostępu do zasobu, pobierał elementy z kolejki, do momentu jej opróżnienia, sprawdzając czy nie pojawił się duplikat. Ważna było, aby wątek rysujący, po wykorzystaniu wszystkich elementów do wygenerowania obrazka pozostawił w kolejce ścieżkę (niebędącą duplikatem), jako ziarno do generowania kolejnej ścieżki.

# Wnioski

Okazało się, że najtrudniejszym elementem nie było uzyskania blokowania dostępu do zasobu, który był dzielony między wątkami, ale generowanie ścieżki w trybie wielowątkowym, gdzie najczęściej do tego celu stosuje się program jednowątkowy (iteracyjny bądź rekursywny). Uzyskanie synchronizacji okazało się znacząco uproszczone dzięki dokumentacji klasy Semaphore[[3]](#footnote-3) w Java, która posłużyła, jako baza do zaprojektowania sekcji krytycznych.

1. Reentrant locking pojawia się wtedy jeśli możliwe jest założenie blokady przez wątek, który już ją posiada [↑](#footnote-ref-1)
2. <http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html> [↑](#footnote-ref-2)
3. <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html> [↑](#footnote-ref-3)