Teoria Współbieżności Lab3 - sprawozdanie

Autor: Michał Flak

Ćwiczenie

Poleceniem zadania było rozwiązanie problemu ograniczonego bufora (producentów - konsumentów).

1. przy pomocy metod wait()/notify()/signal()/await()

1.1 dla przypadku 1 producent / 1 konsument

Zaimplementowano bufor z użyciem kolejki opartej na `LinkedList<Integer>`. Metoda `put(int i)` czeka, aż w kolejce pojawi się wolne miejsce. Metoda `get()` czeka, aż w kolejce pojawi się element.

Klasy producenta i konsumenta są niezmienione względem przykładu. Producent podaje kolejno liczby [0..99], a konsument wypisuje to co dostanie na ekran.

```
class Buffer {
    private Queue<Integer> q;
    int size = 10;

    public Buffer(int size) {
        this.q = new LinkedList<Integer>();
        this.size = size;
    }

    public void put(int i)  {
        synchronized (this) {
            while (q.size() >= size) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            q.add(i);
            notify();
        }
    }

    public int get()  {
        synchronized (this) {
            while (q.size() == 0) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            int value = q.poll();
            notify();
            return value;
        }
    }
}
```

W metodzie `main` tworzymy po jednym wątku producenta i konsumenta:

```
public class Main {
    public static void main(String[] args) {
        Buffer b = new Buffer(10);
        Producer it = new Producer(b);
        Consumer dt = new Consumer(b);

        //it.setPriority(10);
        it.start();
        dt.start();

        try {
            it.join();
            dt.join();
        } catch(InterruptedException ie) { }
    }
}
```

Obserwujemy wyniki:

```
0
1
2
3
...
97
98
99
```

Wyniki są po kolei, program się nie zakleszcza.

1.2. dla przypadku n1 producentów/n2 konsumentów (n1>n2, n1=n2, n1 mniej od n2)

Modyfikujemy metodę main:

```
    public static void main(String[] args) {
        Buffer b = new Buffer(10);

        int n1 = 1;
        int n2 = 20;

        Producer[] ap = new Producer[n1];
        Consumer[] ac = new Consumer[n2];

        Arrays.stream(ap).forEach(p -> {
            p=new Producer(b);
            p.start();
        });
        Arrays.stream(ac).forEach(c -> {
            c=new Consumer(b);
            c.start();
        });

        Arrays.stream(ap).forEach(p -> {
            try {
```

```
                p.join();
            } catch(Exception ie) { }
        });
        Arrays.stream(ac).forEach(c -> {
            try {
                c.join();
            } catch(Exception ie) { }
        });
    }
```

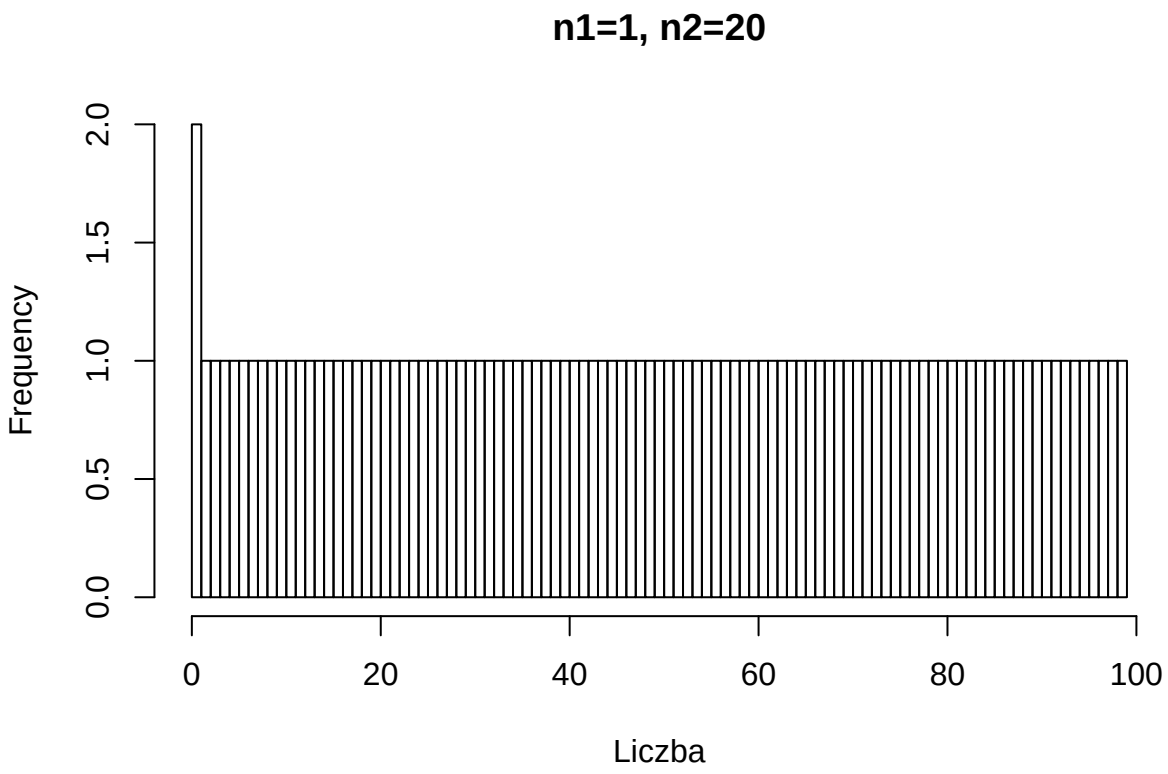
Dla:

```
n1=1
n2=20
```

Wynikiem są liczby [0..99] w losowej kolejności:

```
0
1
2
5
...
42
41
39
37
```

```
data <- read.table("n1-1_n2-20.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=1, n2=20")
```



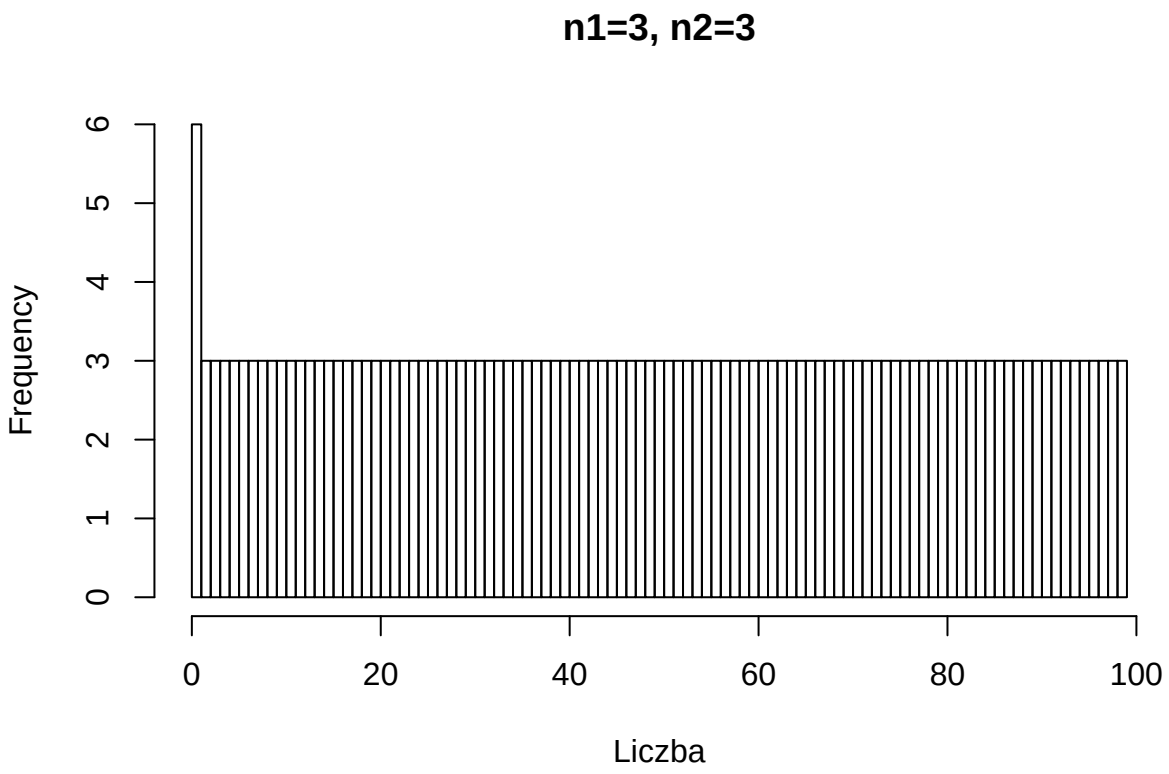
Każda z liczb jest wyświetlona tylko raz. Program wymaga ręcznego zakończenia po wypisaniu wszystkich liczb, bo konsumenci czekają w swoich pętlach na elementy w buforze, które nie nadejdą.

Dla:

```
n1=3
n2=3
```

Wynikiem jest 300 liczb [0.99] w losowej kolejności, z których każda powtarza się 3 razy. Histogram (z niewiadomego mi powodu 0 ma częstotliwość 6 - w wynikach występuje tylko 3 razy):

```
data <- read.table("n1-3_n2-3.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=3, n2=3")
```



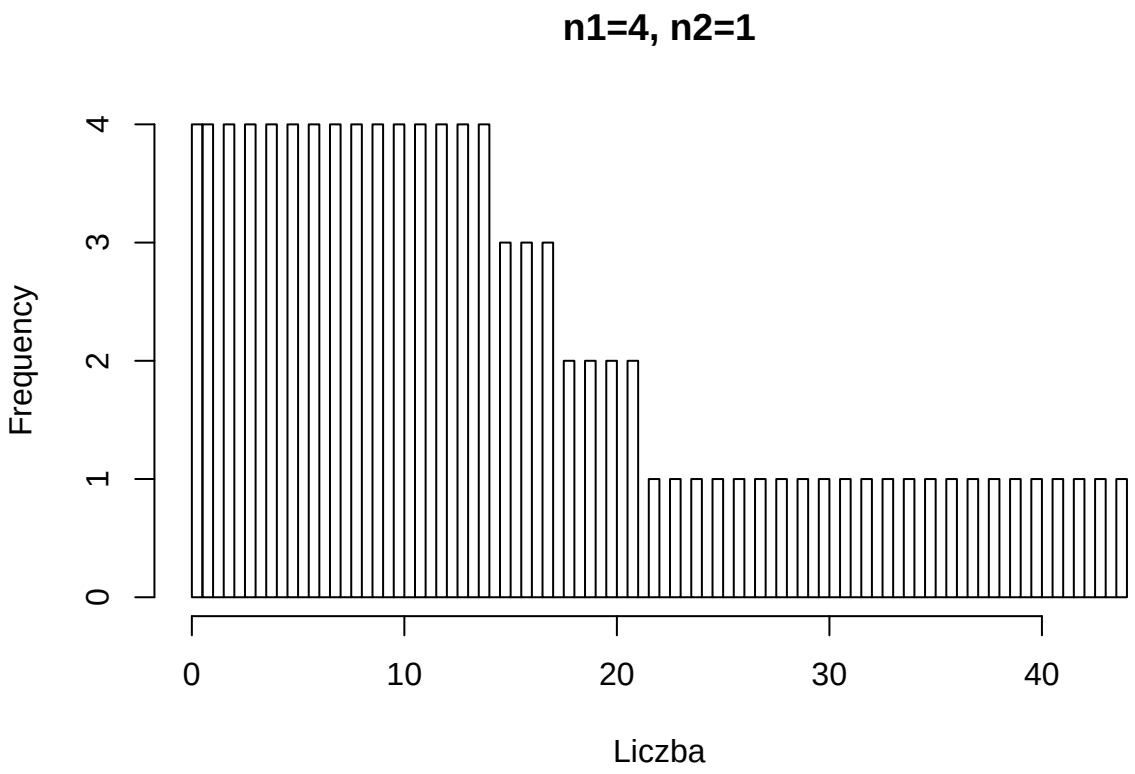
Program tym razem zakańcza się sam - wszystkie liczby skonsumowane.

Dla:

```
n1=4
n2=1
```

Wynikiem jest 100 liczb z przedziału [0.99] w losowej kolejności, z których każda powtarza się *do* 4 razy. Histogram:

```
data <- read.table("n1-4_n2-1.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=4, n2=1")
```



Program zawiesza się - producenci czekają na wyczyszczenie bufora.

1.3. wprowadzić wywołanie metody sleep() i wykonać pomiary, obserwując zachowanie producentów/konsumentów

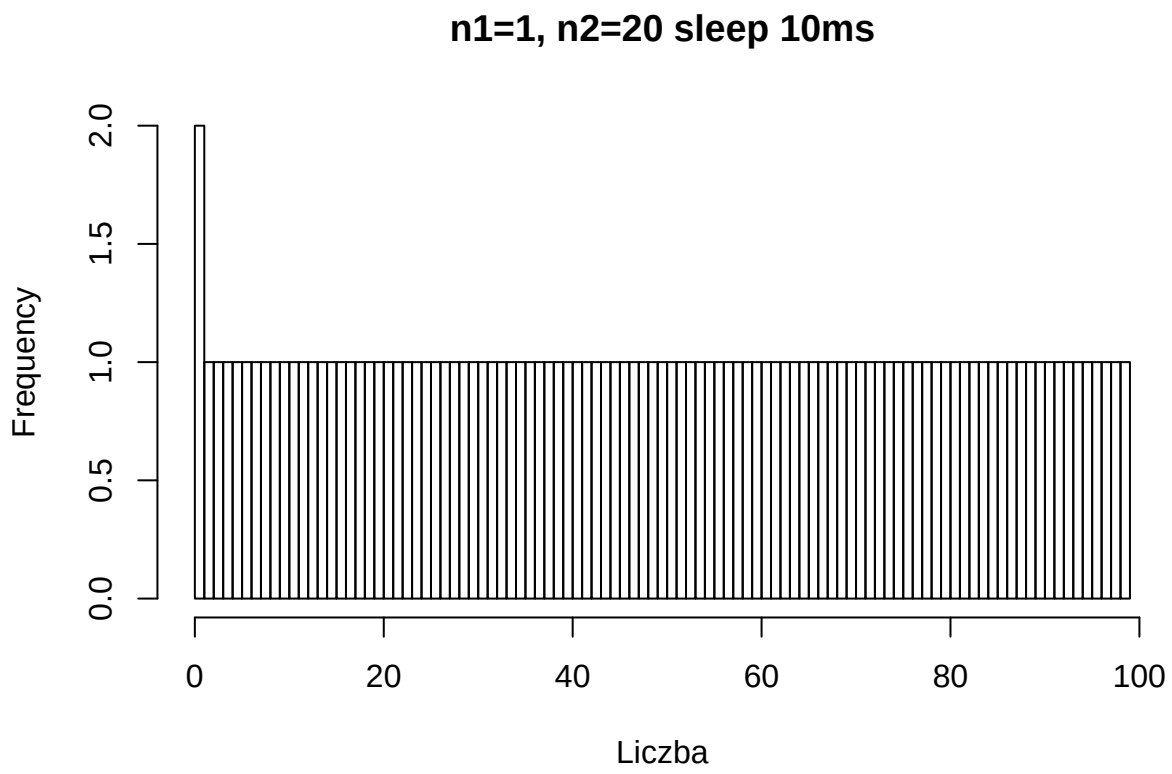
Dodajemy 10 milisekund `Thread.sleep()` do pętli producenta i konsumenta:

```
class Producer extends Thread {
    private Buffer _buf;

    public Producer(Buffer _buf) {
        this._buf = _buf;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.put(i);
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Consumer extends Thread {
```

```
    private Buffer _buf;

    public Consumer(Buffer _buf) {
        this._buf = _buf;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            System.out.println(_buf.get());
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

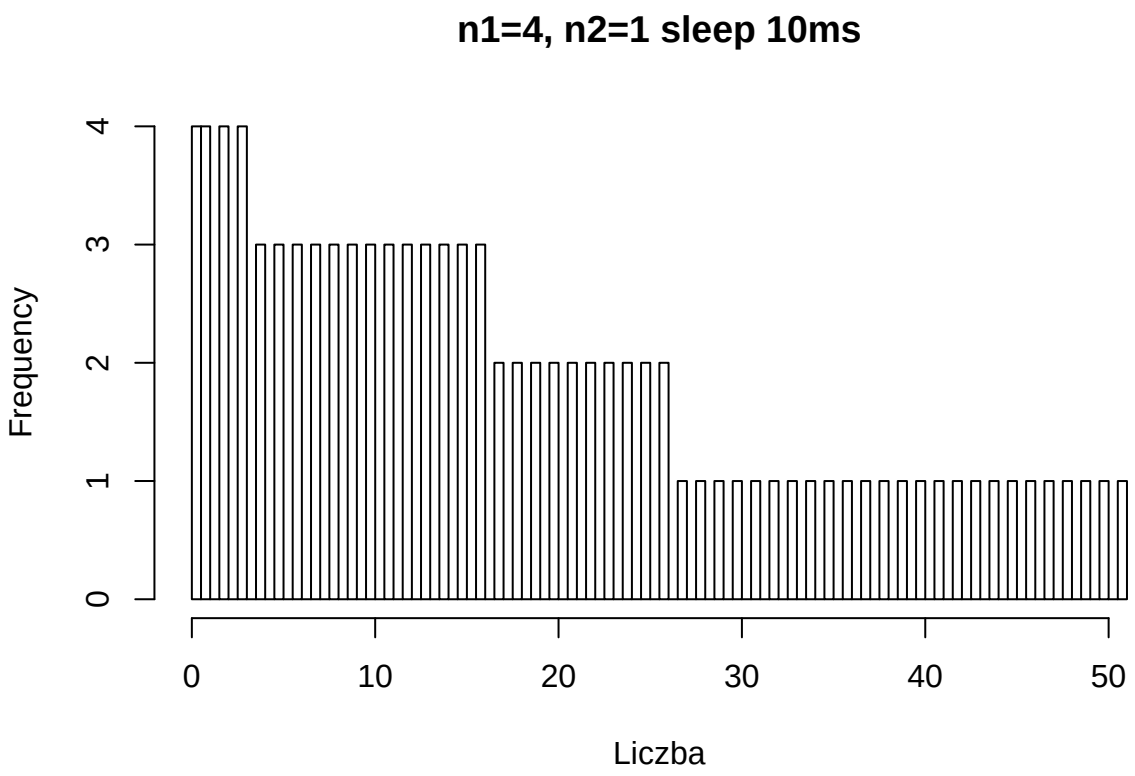
Uruchamiamy ponownie z tymi samymi parametrami co w poprzednim punkcie:

```
data <- read.table("n1-1_n2-20_sleep.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=1, n2=20 sleep 10ms")
```




```
data <- read.table("n1-3_n2-3_sleep.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=3, n2=3 sleep 10ms")
```


```
data <- read.table("n1-4_n2-1_sleep.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=4, n2=1 sleep 10ms")
```



Rozkład jest podobny. Różnicę zaobserwowano w tym, że wyniki są teraz posortowane - wygląda na to, że wątki uruchamiały się sekwencyjnie.

```
0
0
0
1
1
1
2
2
...
```

2. przy pomocy operacji P()/V() dla semafora:

Zaimplementowano semafor zliczający:

```
public class Semafor {
    private int _liczba = 0;

    public Semafor(int initial) {
        _liczba = initial;
    }

    public synchronized void P() {
        while (_liczba == 0) {
            try{
```

```
                wait();
            } catch (InterruptedException ex) {

            }
        }
        _liczba--;
    }

    public synchronized void V() {
        _liczba++;
        notify();
    }
}
```

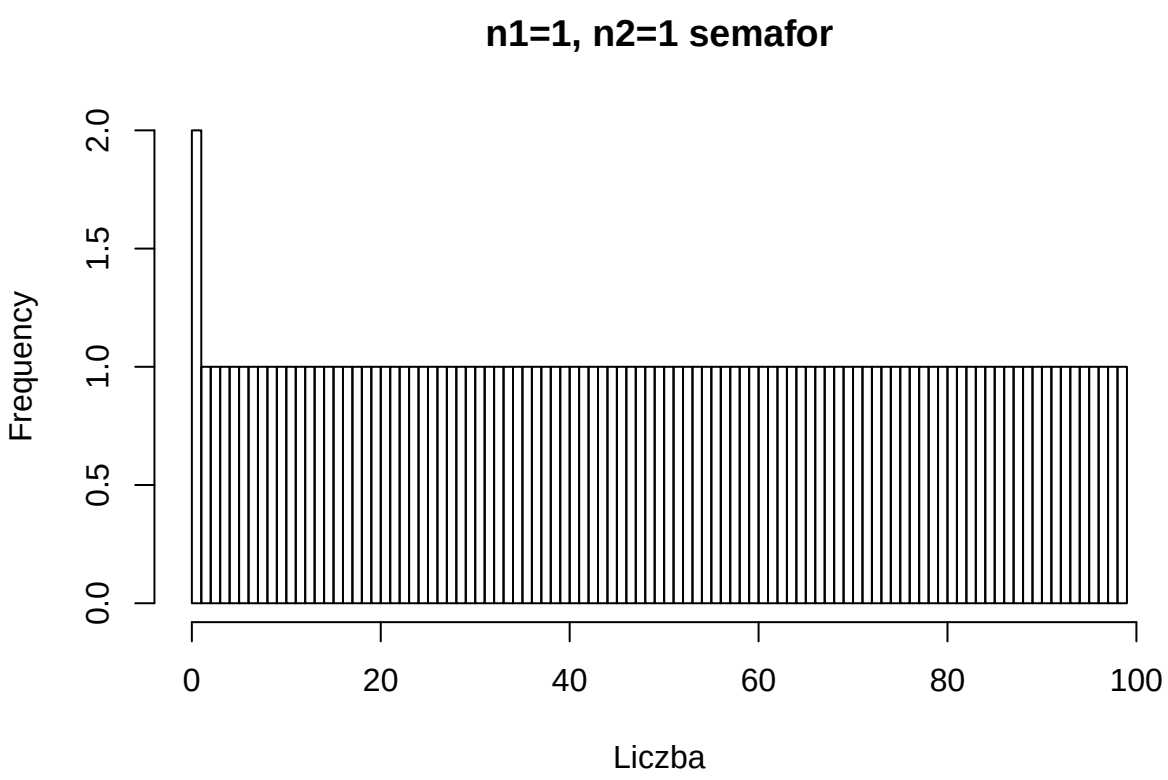
Zmodyfikowano klasę `Buffer`, tak aby używała dwóch semaforów zliczających:

```
class BufferSem implements Buffer  {
    private Queue<Integer> q;
    private int size = 10;
    private Semafor fillCount;
    private Semafor emptyCount;

    public BufferSem(int size) {
        this.q = new LinkedList<Integer>();
        this.size = size;
        fillCount = new Semafor(0);
        emptyCount = new Semafor(size);
    }

    public void put(int i)  {
        emptyCount.P();
        q.add(i);
        fillCount.V();
    }

    public int get()  {
        fillCount.P();
        int value;
        try {
            value = q.poll();
        } catch (Exception ex) {
            value = -1;
        }
        emptyCount.V();
        return value;
    }
}
```

2.1. n1=n2=1

Uzyskano posortowane liczby od 0 do 99.

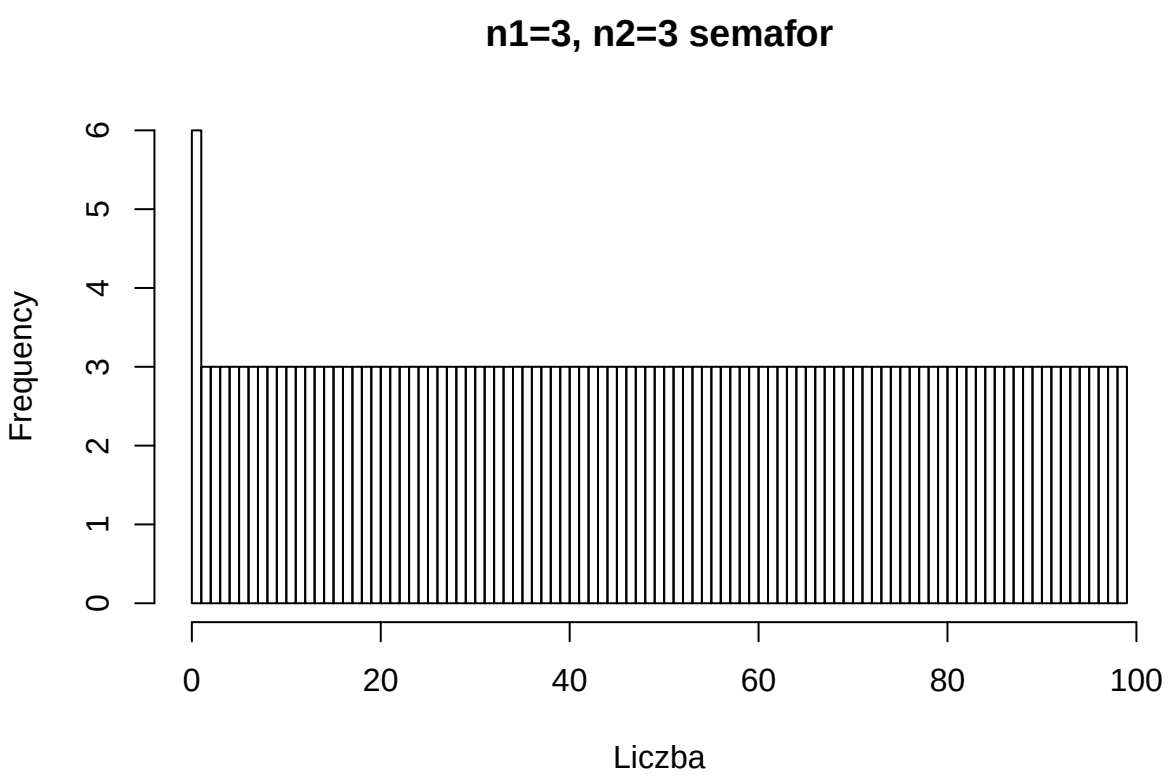
```
data <- read.table("n1-1_n2-1_sem.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=1, n2=1 semafor")
```



2.2. n1>1, n2>1

Wybrano n1 = 3, n2 = 3. Uzyskano liczby z zakresu [0..99], każda w 3 egzemplarzach.

```
data <- read.table("n1-3_n2-3_sem.txt", header=F)
hist(data[,1], 100, xlab="Liczba", main="n1=3, n2=3 semafor")
```



Wnioski

Obie metody skutecznie rozwiązują ten problem. W kodzie produkcyjnym użyłbym jednak wbudowanych narzędzi - na przykład `BlockingQueue`.