Teoria współbieżności
sprawozdanie z ćw. 1
Producenci i konsumenci - problem zagłodzenia

Dominik Adamczyk

15 listopada 2023

1 Wstęp

W tym sprawozdaniu zaprezentuję poprawny kod rozwiązujący problem producentów i konsumentów z możliwością wstawiania losowej porcji w Javie. Pokażę także testy przeprowadzone w celu uzyskania zagłodzenia, a także układy zdarzeń które do owych zagłodzeń i deadlocków prowadzą.

2 Poprawny kod

Program został napisany w Javie. Dzieli się na 4 klasy:

```
public class Main {
    public static int messagesNumber = 2000;
    public static int threadsNumber = 5;
    public static int bufferSize = 10;

    public static Consumer[] consumers = new Consumer[threadsNumber];
    public static Producer[] producers = new Producer[threadsNumber];

    public static Thread[] consumersTh = new Thread[threadsNumber];
    public static Thread[] producersTh = new Thread[threadsNumber];

    public static void main(String[] args) throws InterruptedException {
        Buffer buffer = new Buffer(bufferSize);

        int j = 0;
        while (j < threadsNumber){
            consumers[j] = new Consumer(buffer, bufferSize, j);
            producers[j] = new Producer(buffer, bufferSize, j);
            consumersTh[j] = new Thread(consumers[j]);
            producersTh[j] = new Thread(producers[j]);

            producersTh[j].start();
            consumersTh[j].start();
            j++;
        }

        while(true){
            sleep(300);
            printWaitingLoops();
        }

    }
```

```
    public static void printWaitingLoops(){
        System.out.println("**** Number of productions / consumptions ****");
        for (int i = 0; i < threadsNumber; i++){
            if (producers[i] != null){
                System.out.println("Producer id: " + i + " number of executions: " +
                    producers[i].loops);
            }
        }
        System.out.println("========================================");
        for (int i = 0; i < threadsNumber; i++){
            if (consumers[i] != null) {
                System.out.println("Consumer id: " + i + " number of executions: " +
                    consumers[i].loops);
            }
        }
        System.out.println("**** **** **** **** **** **** **** **** ****");
    }
}
```

Przy okazji widoczna tutaj jest jedna z metod sprawdzania zagłodzenia. Polega ona na wypisywaniu co ustalony czas funkcji printWaitingLoops(). Funkcja ta zbiera dane z wszystkich producentów i konsumentów i wypisuje informację o tym ile razy coś wyprodukowali, albo skonsumowali.

Klasa producenta:

```
public class Producer extends Thread {
    private final Buffer buf;
    private final int M;

    private final int id;

    public int loops;

    public Producer(Buffer buf, int M, int id) {
        this.buf = buf;
        this.M = M;
        this.id = id;
    }

    public void run() {
        while(true) {
            loops++;
            int val;
            if (false){ } // normal distirbution of produce chunks
//          if (this.id == 0) { val = M-1; } // producer id 0 starvation
//          if (this.id < 3){ val = 1; }   // deadlock situation
            else { val = (int) (Math.random()*(M-1)+1); }

            int[] a = new int[val];
            for(int i=0; i<a.length; i++) {
                a[i] = (int)(Math.random() * 10) + 1;
            }
            buf.produce(a, id);

            try {
                sleep(1);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);}}}}
```

Klasa konsumenta:

```
public class Consumer extends Thread {

    private Buffer buf;
    private final int M;
    private final int id;
    public int loops;

    public Consumer(Buffer buf, int M, int id){
        this.buf = buf;
        this.M = M;
        this.id = id;
    }

    public void run(){
        while(true) {
            loops++;
            int val;
            if (false){ } // normal
//          if (this.id == 0) { val = M-1; } // consumer id 0 starvation
//          if (this.id < 3){ val = M-1; } // deadlock situation
            else{ val = (int) (Math.random()*(M-1)+1); }
            buf.consume(val, id);
            try {
                sleep(1);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);}}}}
```

W dwóch powyższych klasach znajdują się zakomentowane instrukcje if. Ich znaczenie jest związane z doprowadzaniem do niekorzystnych sytuacji i zostanie ono wyjaśnione w późniejszych punktach.

Klasa Bufora:

```
public class Buffer {
    private final LinkedList<Integer> buffer;
    public final int bufferCapacity;
    private final ReentrantLock lock;
    private final Condition firstProducer;
    private final Condition firstConsumer;
    private final Condition restProducers;
    private final Condition restConsumers;

    private int countFirstProducer = 0;
    private int countFirstConsumer = 0;
    private int countRestProducers = 0;
    private int countRestConsumers = 0;
    boolean hasFirstProducer = false;
    boolean hasFirstConsumer = false;

    public Buffer(int m){
        this.buffer = new LinkedList<>();
        this.bufferCapacity = m * 2;
        this.lock = new ReentrantLock();
        this.firstConsumer = lock.newCondition();
        this.firstProducer = lock.newCondition();
        this.restConsumers = lock.newCondition();
        this.restProducers = lock.newCondition();
    }
```

```
public void produce(int[] toProduce, int id){
    lock.lock();
    try {
        countRestProducers++;
        while(hasFirstProducer){

             System.out.println("PRODUCER id: " + id +
                    " waiting on restProducers | number of waiting processes: " +
                        countRestProducers);
            restProducers.await();
        }
        countRestProducers--;
        countFirstProducer++;
        hasFirstProducer = true;
        while (buffer.size() + toProduce.length > bufferCapacity){

             System.out.println("PRODUCER id: " + id +
                    " waiting on firstProducer | number of waiting processes: " +
                        countFirstProducer);

            firstProducer.await();
        }
        countFirstProducer--;
        hasFirstProducer = false;

        for (int j : toProduce) buffer.add(j);

        System.out.println("PRODUCER id: " + id +
                 " added " + toProduce.length + " to buffer");

        restProducers.signal();
        firstConsumer.signal();


    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally{
        lock.unlock();
    }
}

public void consume(int toConsume, int id){
    lock.lock();
    try {
        countRestConsumers++;
        while(hasFirstConsumer){

             System.out.println("CONSUMER id: " + id +
                    " waiting on restConsumers | number of waiting processes: " +
                        countRestConsumers);

            restConsumers.await();
        }
        countRestConsumers--;
        countFirstConsumer++;
        hasFirstConsumer = true;
        while (buffer.size() - toConsume < 0){
             System.out.println("CONSUMER id: " + id +
                    " waiting on firstConsumer | number of waiting processes: " +
                        countFirstConsumer);
```

```
                firstConsumer.await();
            }
            countFirstConsumer--;
            hasFirstConsumer = false;

            for (int i=0; i < toConsume; i++) buffer.removeFirst();
            System.out.println("CONSUMER id: " + id +
                    " consumed " + toConsume + " from buffer");

            restConsumers.signal();
            firstProducer.signal();

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally{
            lock.unlock();
        }
    }
}
```

Klasa bufora jest najważniejsza do dla algorytmu i najbardziej się różni pomiędzy różnymi wersjami algorytmu. Dodatkowe instrukcje wypisanie pomagają w sprawdzaniu poprawnego działania poszczególnych kolejek warunków.

Rozwiązanie z 2 condition modyfikuje powyższy kod pozbywając się dwóch pętli while i dwóch condition, natomiast rozwiązanie z hasWaiters() ignoruje zmienne typu bolean i używa odpowiednich warunków hasWaiters() w pierwszych pętlach while.

3 Uzyskanie zagłodzenia i zakleszczenia - zliczanie wykonań funkcji

W klasach producenta i konsumenta znajdują się zakomentowane instrukcje if. Służą one zaprezentowaniu niekorzystnych warunków dla producentów i konsumentów w różnych konfiguracjach działania buforu. Dla poprawnego algorytmu niezależnie od wybranej instrukcji if nie nastąpi zagłodzenia lub deadlock. Dla rozwiązania z dwoma warunkami łatwo zauważalne jest zagłodzenie, gdy odpowiedni if zostanie odkomentowany. Dla rozwiązania z hasWaiters można zobaczyć zarówno zagłodzenia jak i zakleszczenie.

Do zagłodzenia w błędnych algorytmach najłatwiej jest doprowadzić poprzez zwiększenie rozmiaru konsumpcji / produkcji dla jednego z wątków. Wątki te powinny mieć zauważalnie mniejszą liczbę gdy wypisujemy statystyki w funkcji printWaitingLoops().

Do zakleszczenia dla rozwiązania z hasWaiters() doprowadziłem ustawiając około połowę wątków konsumentów, aby konsumowały minimalną ilość i połowę wątków producentów aby produkowały maksymalną ilość. Promuje to prawdopodobieństwo wystąpienia ciągu zdarzeń prowadzącego do zakleszczenia. Podobny ciąg zostanie zaprezentowany w dalszej części.

Poniżej prezentuję zagłodzenie wyniki funkcji PrintWaitingLoops() dla każdej z 3 metod. Dla każdej z nich użyte zostały niekorzystne rozmiary konsumpcji / produkcji dla wątku zerowego. Do testów użyto 5 wątków producentów i 5 wątków konsumentów.

```
4 conditions bool:

**** Number of productions / consumptions ****
Producer id: 0 number of executions: 7142
Producer id: 1 number of executions: 6987
Producer id: 2 number of executions: 7164
Producer id: 3 number of executions: 6971
Producer id: 4 number of executions: 7144
========================================
Consumer id: 0 number of executions: 6009
Consumer id: 1 number of executions: 6212
Consumer id: 2 number of executions: 6011
Consumer id: 3 number of executions: 6136
Consumer id: 4 number of executions: 5999
**** **** **** ****  ****  **** **** **** ****
```

Rysunek 1: Rozwiązanie poprawne - liczba wykonań produce / consume dla każdego z wątków

```
4 conditions hasWaiters()
**** Number of productions / consumptions ****
Producer id: 0 number of executions: 175940              <--- zagłodzenie
Producer id: 1 number of executions: 302261
Producer id: 2 number of executions: 293772
Producer id: 3 number of executions: 303904
Producer id: 4 number of executions: 291728
========================================
Consumer id: 0 number of executions: 286504              <--- zagłodzenie
Consumer id: 1 number of executions: 409573
Consumer id: 2 number of executions: 411319
Consumer id: 3 number of executions: 403160
Consumer id: 4 number of executions: 403269
**** **** **** ****  ****  **** **** **** ****
```

Rysunek 2: Rozwiązanie z hasWaiters() - liczba wykonań produce / consume dla każdego z wątków

```
2 conditions
**** Number of productions / consumptions ****
Producer id: 0 number of executions: 120158              <--- zagłodzenie
Producer id: 1 number of executions: 221162
Producer id: 2 number of executions: 220870
Producer id: 3 number of executions: 223403
Producer id: 4 number of executions: 225707
========================================
Consumer id: 0 number of executions: 120004              <--- zagłodzenie
Consumer id: 1 number of executions: 222359
Consumer id: 2 number of executions: 222427
Consumer id: 3 number of executions: 226654
Consumer id: 4 number of executions: 219640
**** **** **** ****  ****  **** **** **** ****
```

Rysunek 3: Rozwiązanie z 2 condition - liczba wykonań produce / consume dla każdego z wątków

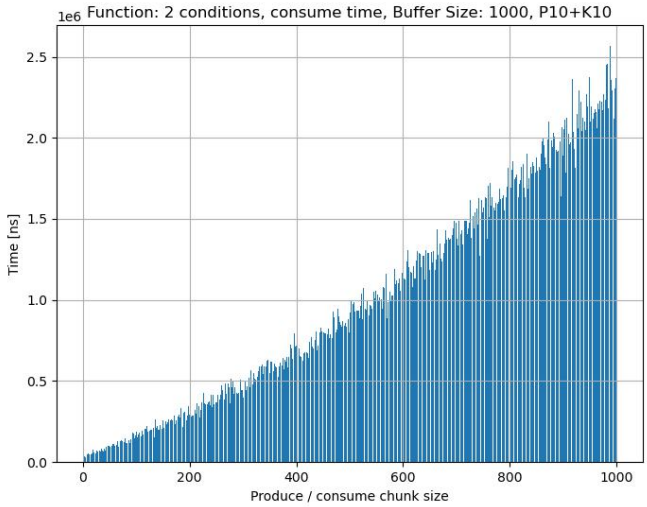
Dla rozwiązania poprawnego nie zostało wykazane zagłodzenie, które jest widoczne w pozostałych rozwiązaniach.

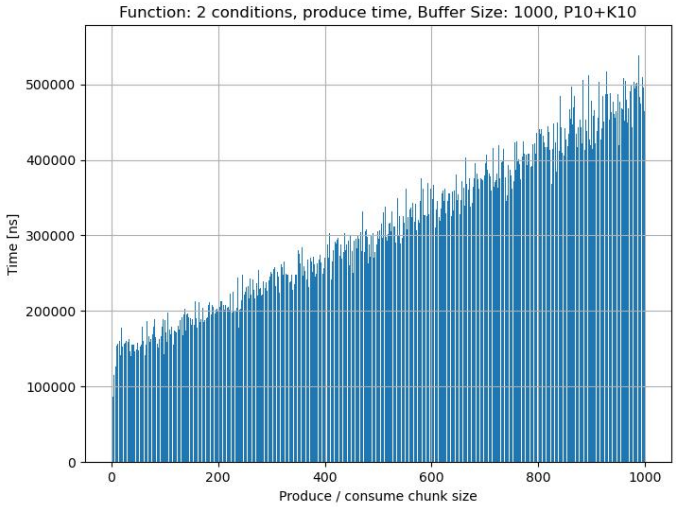
4 Uzyskanie zagłodzenia i zakleszczenia - mierzenie czasów

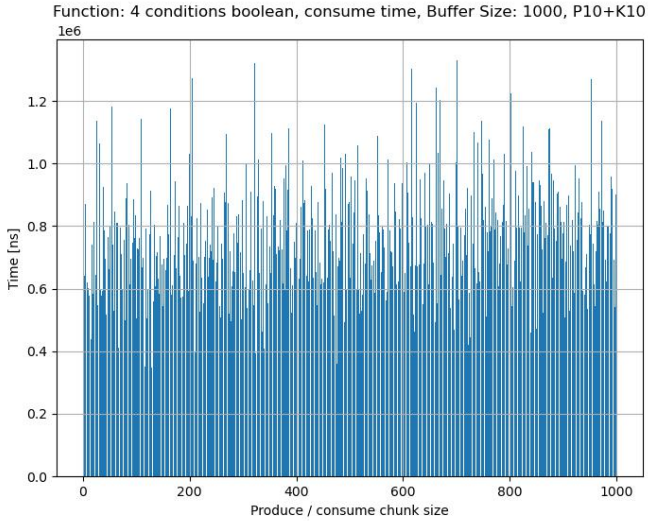
Drugim zaimplementowanym podejściem wykazującym zagłodzenie jest obliczanie średniego czasu wykonania się funkcji produkcji lub konsumpcji zależnie od długości wielkości wstawianej / wyciąganej z bufora. W tym celu zmodyfikowane zostały klasy Producera i Consumera (żeby przechowywały listę z czasami wykonań funkcji zależnie od wielkości interakcji z buforem) oraz klasa Main w której została dodana funkcja zbierająca dane ze wszystkich wątków i wyciągająca z nich średnie. Funkcja ta zostaje wykonana 60s po włączeniu programu. Odpowiednie dane zostały zebrane do pliku .csv a później opracowane w Pythonie.

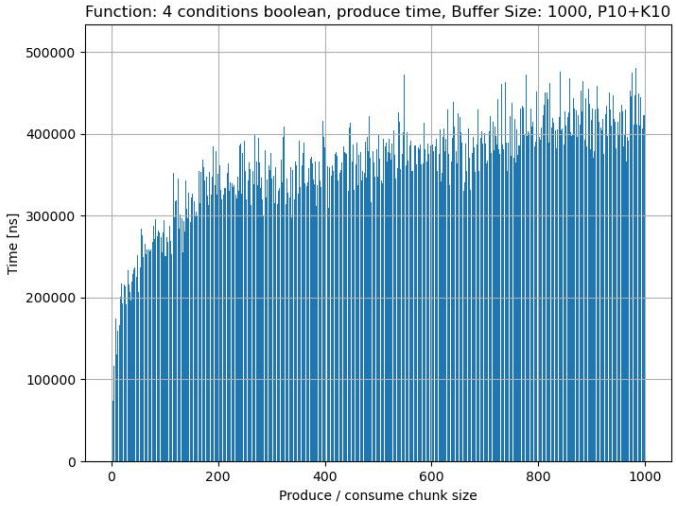
Zmianiony kod Javy:

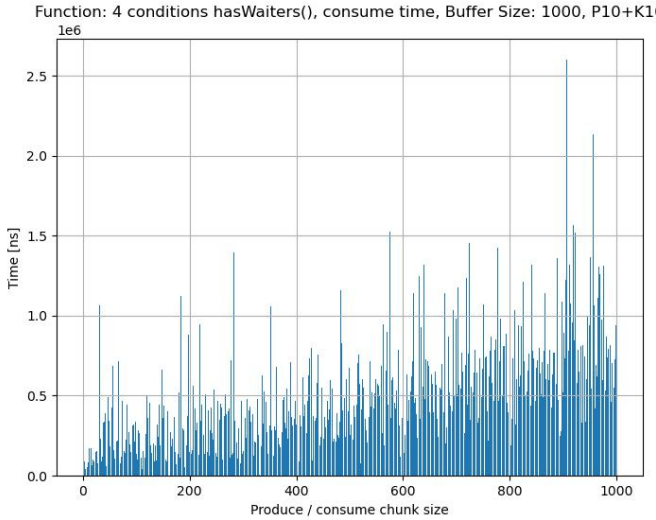
```
class Main:
    public static void printAverageTimes() throws IOException {
        System.out.println("******");
        List<List<Long>> getTimes = new ArrayList<>();
        List<List<Long>> putTimes = new ArrayList<>();
        for (int i = 0; i <= bufferSize; i++){
            getTimes.add(new ArrayList<>());
            putTimes.add(new ArrayList<>());}
        for (int i = 0; i < threadsNumber; i++){
            for (int j = 0; j <= bufferSize; j++){
                getTimes.get(j).addAll(consumers[i].timesList.get(j));
                putTimes.get(j).addAll(producers[i].timesList.get(j));}}
        List<Double> getMeans = new ArrayList<>();
        List<Double> putMeans = new ArrayList<>();
        for (int i = 0; i <= bufferSize; i++){
            long getSum = 0;
            long putSum = 0;
            for (Long val : getTimes.get(i)){getSum += val;}
            for (Long val : putTimes.get(i)){putSum += val;}
            double getMean = (double) (!getTimes.get(i).isEmpty() ? getSum /
                getTimes.get(i).size() : 0);
            double putMean = (double) (!putTimes.get(i).isEmpty() ? putSum /
                putTimes.get(i).size() : 0);
            getMeans.add(getMean);
            putMeans.add(putMean);
        }
        BufferedWriter writer = new BufferedWriter(new FileWriter("times2.csv", true));
        for (int i = 1; i < bufferSize; i++){
            writer.write("get4condbool,"+bufferSize+","+threadsNumber+","+i+","+getMeans.get(i)+"\n");
            System.out.println("get4condbool,"+bufferSize+","+threadsNumber+","+i+","+getMeans.get(i));
        }
        for (int i = 1; i < bufferSize; i++){
               writer.write("put4condbool,"+bufferSize+","+threadsNumber+","+i+","+putMeans.get(i)+"\n");
            System.out.println("put4condbool,"+bufferSize+","+threadsNumber+","+i+","+putMeans.get(i));
        }
        writer.close();
    }
class Consumer{
    public void run(){
        while(true) {
            loops++;

            int val = (int) (Math.random()*(M-1)+1);
            long start = System.nanoTime();
            buf.consume(val, id);
            long elapsed = System.nanoTime() - start;
            timesList.get(val).add(elapsed);}}}
// analogicznie dla producenta
```

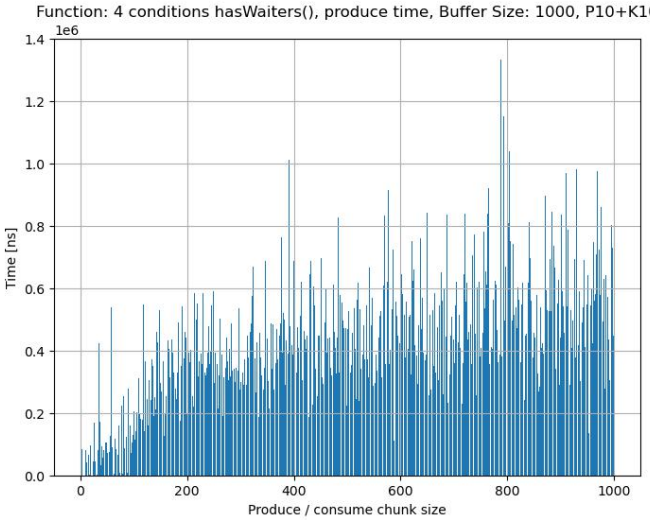

Function: 2 conditions, consume time, Buffer Size: 1000, P10+K10
1e6
Time [ns]
Produce / consume chunk size


Function: 2 conditions, produce time, Buffer Size: 1000, P10+K10
Time [ns]
Produce / consume chunk size


Function: 4 conditions boolean, consume time, Buffer Size: 1000, P10+K10
1e6
Time [ns]
Produce / consume chunk size


Function: 4 conditions boolean, produce time, Buffer Size: 1000, P10+K10
Time [ns]
Produce / consume chunk size


Function: 4 conditions hasWaiters(), consume time, Buffer Size: 1000, P10+K10
1e6
Time [ns]
Produce / consume chunk size


Function: 4 conditions hasWaiters(), produce time, Buffer Size: 1000, P10+K10
1e6
Time [ns]
Produce / consume chunk size

Powyższe wykresy wyraźnie pokazują występowanie zagłodzenia dla programu z dwoma warunkami. Im większy rozmiar wstawiania / zabierania z bufora tym dłuższy czas oczekiwania funkcji. Problem ten w mniejszym stopniu uwidacznia się też w rozwiązaniu z hasWaiters(). Rozwiązanie poprawne ma w przybliżeniu stały czas dla każdego rozmiaru. Wyjątkiem są małe liczby gdzie funkcji jest łatwiej ominąć pętlę while dla FirstProducer/FirstConsumer (ponieważ jest większa szansa na wstawienie od razu małej wartości do bufora).

5 Ciągi sytuacji prowadzące do zagłodzeń i zakleszczenia

Poniżej prezentuję ciągi sytuacji prowadzące do uzyskania zagłodzeń i zakleszczeń w błędnych rozwiązaniach.

5.1 Zagładzanie w rozwiązaniu z dwoma warunkami

Rozważmy problem na przykładzie zagłodzienia producenta. Jeśli trafi się producent który chce włożyć stosunkowo dużą liczbę na bufor to może nie mieć zbyt często na to okazji. Gdy tylko bufor się zwolni staje się on jednym z kandydatów aby na ten bufor coś dodać, jeżeli nie będzie mógł dodać nic na bufor (bo bufor by się przepełnił) to zostanie ponownie zawieszony. Tymczasem producenci, którzy mają mniejsze wartości do położenia na bufor nie będą się tak często wieszały.

```
Przykładowy układ zdarzeń:
Bufor ma rozmiar 10 (czyli możemy na nim umieszczać wartości 1-5)
maxBuff = 10
użyte wątki: P1, P2, C1, C2

\\ sytuacja początkowa

beforeLockQueue: {C1(2), C2(2)}
buffState = 10
producers = {P1(5), P2(1)}
consumers = {}

\\ konsumenci wchodzą i zjadają. Wybudzają obydwu konsumentów

beforeLockQueue: {P1(5), P2(1), C1(2), C2(2)}
buffState = 6
producers = {}
consumers = {}

\\ producent P2 i P1 mogą wejść, jednak jedynie P2 włoży na bufor, P1 się znowu zawiesi

beforeLockQueue: {P2(3), C1(3), C2(1)}
buffState = 6
producers = {P1(5)}
consumers = {}

\\ producent P2 może włożyć na bufor

beforeLockQueue: {P2(4), C1(3), C2(1)}
buffState = 9
producers = {P1(5)}
consumers = {}
```

```
\\ konsument C1 może zjeść, wybudzi P1

beforeLockQueue: {P1(5), P2(4), C1(4), C2(1)}
buffState = 6
producers = {}
consumers = {}


\\ P1 dostaje locka ale znow się od razu zawiesi

beforeLockQueue: {P2(4), C1(4), C2(1)}
buffState = 6
producers = {P1(5)}
consumers = {}

Jak widać wątek producenta mający dużą wartość ma mniejszą szansę dodania jej do kolejki
```

5.2 Zagłodzenie w rozwiązaniu z hasWaiters()

```
    Przykład zagłodzenia dla producenta:
Musimy dojść do sytuacji, w której mamy kilka wątków zawieszonych na first producer.
↪  Dzieje się tak gdy np:

maxBuff = 10

\\sytuacja początkowa, jeden producent na firstProducer

buffState = 9
beforeLockQueue: {P3(3)}     \\P1(5) oznacza producenta 1 chcącego położyć 5 na
↪  bufor
firstProducer: {P1(5)}
restProducers: {P2(4)}

\\ jakiś konsument wchodzi i zjada 1, w rezultacie budzi producenta który  teraz
↪  oczekuje na przyznanie lock

buffState = 8
beforeLockQueue: {P3(5), P1(5)}
firstProducer: {}
restProducers: {P2(4)}


\\ lock dostaje inny czekający producent i wiesza się na firstProducer

buffState = 8
beforeLockQueue: {P1(5)}
firstProducer: {P3(5)}
restProducers: {P2(4)}

\\ lock zostaje przyznany czekającemu producentowi, który nie może wstawić na bufor,
↪  a dodatkowo powiesi się na firstProducer (bo stamtąd został obudzony)

buffState = 8
beforeLockQueue: {}
firstProducer: {P3(5), P1(5)}
restProducers: {P2(4)}
```

```
W tym przykładzie dochodzimy do sytuacji gdy więcej niż 1 producent jest powieszony
↪  na firstProducer. Co więcej za każdym razem gdy konsumujemy budzimy tylko
↪  jednego producenta z firstProducer. Oznacza to, że będzie dochodziło do
↪  zagładzania wątków które akurat będą miały mniejszą możliwość dodania rzeczy na
↪  bufor, tak samo jak miało to miejsce w przypadku rozwiązania na 2 warunki (tak
↪  właściwie bufor firstProducers w tym przypadku staje się analogią buforu
↪  Producers z zadania na 2 warunki).

W ten sam sposób jak pokazany wyżej na firstProducer może trafić więcej niż 2 wątki
↪  producentów, a nawet wszystkie z nich. Analogiczne rozumowanie można
↪  przeprowadzić dla warunku firstConsumer, gdzie zawieszonych na nim będzie więcej
↪  1 wątek.
```

6 Zakleszczenie w rozwiązaniu z hasWaiters()

```
Przygotowanie rozumowania
Użyte wątki: P1, P2, P3, P4 C1, C2, C3
maxBuff = 10 (Zakładam że wcześniej uzupełniły go wątki producentów)

\\ stan początkowy
buffState = 10
beforeLockQueue: {P2(5), P3(5), P4(5), C1(1), C2(5), C3(5)}
firstProducer: {P1(2)}
restProducers: {}
firstConsumer: {}
restConsumers: {}

\\ zjadamy konsumentem, przez co budzimy P1

buffState = 9
beforeLockQueue: {P1(2), P2(5), P3(5), P4(5), C1(1), C2(5), C3(5)}
firstProducer: {}
restProducers: {}
firstConsumer: {}
restConsumers: {}

\\ P1 czeka na locku, wpuszczamy P2, który widzi pustą kolejkę firstProducer, nie
↪  może wstawić więc wisi na firstProducer

buffState = 9
beforeLockQueue: {P1(4), P3(5), P4(5), C1(1), C2(5), C3(5)}
firstProducer: {P2(5)}
restProducers: {}
firstConsumer: {}
restConsumers: {}

\\ zjadamy konsumentem, budzimy P2

\\ P2 czeka na locku, wpuszczamy P3, który widzi pustą kolejkę

buffState = 8
beforeLockQueue: {P1(4), P2(5), P4(5), C1(1), C2(5), C3(5)}
firstProducer: {P3(5)}
restProducers: {}
firstConsumer: {}
restConsumers: {}
```

```
\\ zjadamy konsumentem, budzimy P3

\\ P3 czeka na locku, wpuszczamy P4, który widzi pustą kolejkę

buffState = 7
beforeLockQueue: {P1(4), P2(5), P3(5), C1(2), C2(5), C3(5)}
firstProducer: {P4(5)}
restProducers: {}
firstConsumer: {}
restConsumers: {}

\\ wpuszczamy P1, P2, P3, wszystkie wznowią się w pętli z lockiem firstProducer

buffState = 7
beforeLockQueue: {C1(2), C2(5), C3(5)}
firstProducer: {P1(4), P2(5), P3(5), P4(5)}
restProducers: {}
firstConsumer: {}
restConsumers: {}

\\ zjadamy konsumentem 2 razy, budzimy producenta 2 razy

buffState = 0
beforeLockQueue: {P1(4), P2(5), C1(5), C2(5), C3(5)}
firstProducer: {P3(5), P4(5)}
restProducers: {}
firstConsumer: {}
restConsumers: {}

\\ wprowadzamy wątki konsumentów które wieszają się

buffState = 0
beforeLockQueue: {P1(4), P2(5), C2(5)}
firstProducer: {P3(5), P4(5)}
restProducers: {}
firstConsumer: {C1(5)}
restConsumers: {C3(5)}

\\ wprowadzamy producenta P1, który wybudza C1

buffState = 4
beforeLockQueue: {C1(5), P1(2), P2(5), C2(5)}
firstProducer: {P3(5), P4(5)}
restProducers: {}
firstConsumer: {}
restConsumers: {C3(5)}

\\ wprowadzamy konsumentów. Pierwsze C2 (bo będzie widział puste firstConsumer), a
↪  później C1 (który wznowi się i zawiesi na firstConsumer)

buffState = 4
beforeLockQueue: {P1(2), P2(5)}
firstProducer: {P3(5), P4(5)}
restProducers: {}
```

```
firstConsumer: {C1(5), C2(5)}
restConsumers: {C3(5)}

\\ wprowadzamy P1, które zawiesi się na restProducers (bo wcześniej zwyczajnie
↪  wyszło z buforu, a teraz zobaczy że są wątki w firstProducer). Wprowadzamy też
↪  P2, które doda do buforu i wybudzi C2, które zje z buforu . C2 wybudzi też P3 z
↪  firstProducer który ponownie doda do buforu.


buffState = 9
beforeLockQueue: {P2(5), C2(5), P3(5)}
firstProducer: {P4(5)}
restProducers: {P1(2)}
firstConsumer: {C1(5)}
restConsumers: {C3(5)}

\\ Wszystkie wątki w kolejce do locka będą zaczynać swoje funkcje put i take od
↪  początku. Zobaczą że istnieją już wątki w pierwszych konsumentach/producentach
↪  więc zawieszą się na pozostałych

buffState = 9
beforeLockQueue: {}
firstProducer: {P4(5)}
restProducers: {P2(5), P1(2), P3(5)}
firstConsumer: {C1(5)}
restConsumers: {C3(5), C2(5)}

\\ Co powoduje deadlocka
```