

# Producer Consumer Coarse/Fine Synchronization

Diaconescu Bogdan Florin

January 2020

## 1 Problem statement

Implement a program to solve the producer-consumer problem using:

- a) coarse synchronization
- b) fine synchronization.

The code must be tested with at least 4 producers and the number of consumers will be given by the number of available virtual CPUs.

## 2 Implementation/Solution

### 2.1 Sincronizare grosiera

Pentru rezolvarea problemei am creat 4 clase:

- **Producer:** clasa ce reprezinta producatorul sub forma unui thread.
  - *number* - variabila de tip int prin intermediul careia este identificat producatorul;
  - *queue* - instanta a clasei *LockBasedQueue* unde sunt puse elementele "prodate" de producatori;
  - *Producer(LockBasedQueue queue, int number)* - constructorul clasei;
  - *run()*: threadul va executa la nesfarsit urmatoarele instructiuni:
    - \* prin intermediul metodei *put(elementValue,number)* producatorul "number" pune in coada elementul "elementValue";
    - \* threadul asteapta 1000 milisecunde.
- **Consumer:** clasa ce reprezinta consumatorul sub forma unui thread.
  - *number* - variabila de tip int prin intermediul careia este identificat consumatorul;
  - *queue* - instanta a clasei *LockBasedQueue* de unde sunt extrase elementele "consumate" de consumatori;
  - *Consumer(LockBasedQueue queue, int number)* - constructorul clasei;

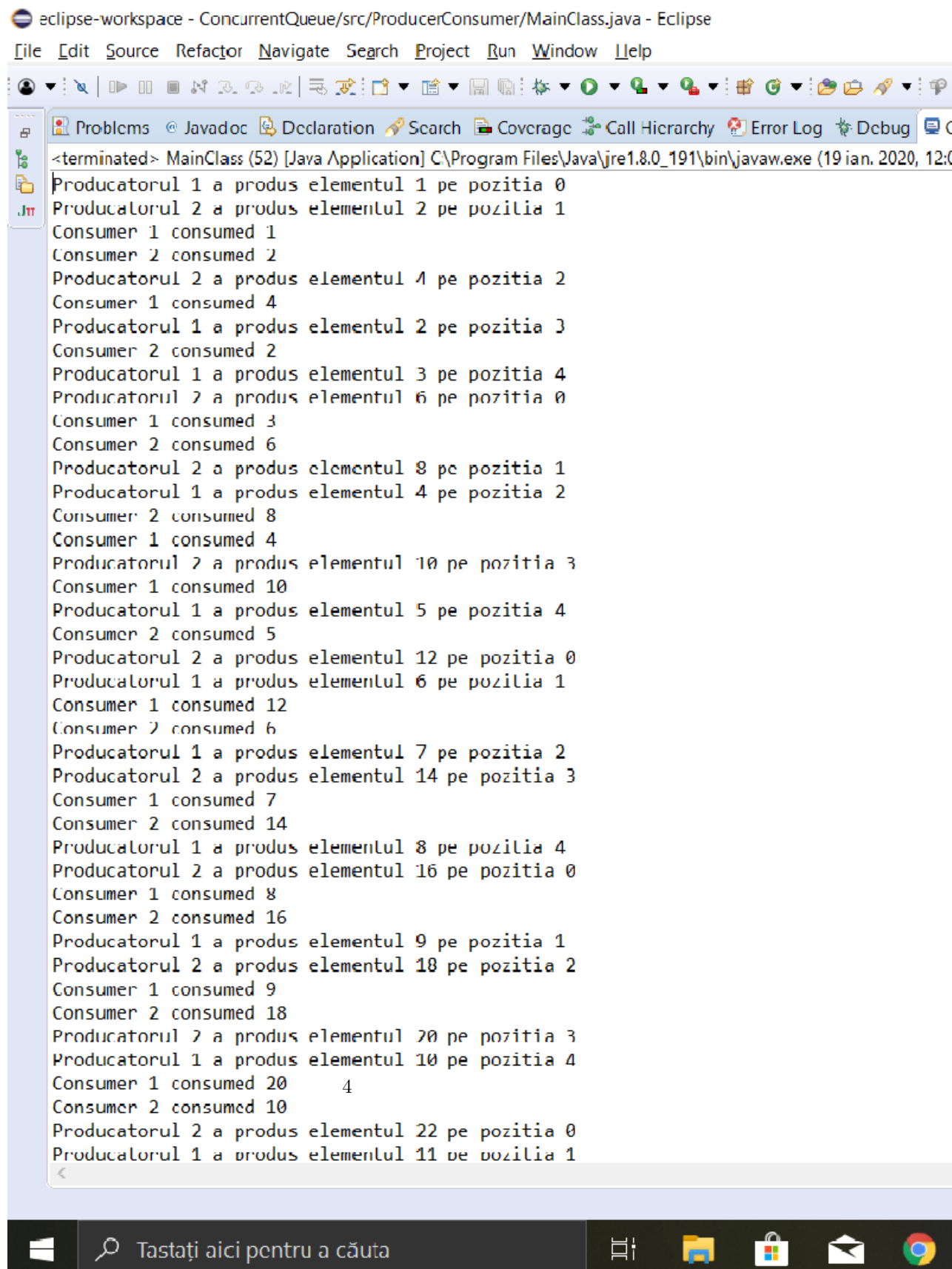
- *run()*: threadul va executa la nesfarsit urmatoarele instructiuni:
  - \* prin intermediul metodei *take()* este extras un element din coada *queue*;
  - \* este afisat elementul;
  - \* threadul asteapta 1000 milisecunde.
- **LockBasedQueue**: clasa ce reprezinta coada in care sunt puse si extrase elementele
  - *head* - variabila de tip *int* ce reprezinta capul cozii;
  - *tail* - variabila de tip *int* ce reprezinta coada cozii;
  - *elementsList* - vector de tipul *int* in care sunt stocate elementele;
  - *queueLock* - zavor folosit pentru accesul la coada;
  - *notFull* si *notEmpty* - cele 2 conditii folosite pentru zavor;
  - *LockBasedQueue(int maximumLength)* - constructorul clasei;
  - *take()* - metoda folosita pentru extragerea unui element din coada. Se realizeaza urmatoarele instructiuni:
    - \* se blocheaza zavorul cozii *queueLock.lock()*;
    - \* cat timp coada este goala se asteapta punerea unui element in coada prin apelarea metodei *await()* pentru conditia *notEmpty*. Aceasta metoda face ca threadul respectiv sa astepte pana cand este notificat de un alt thread;
    - \* se extrage un element din coada, deci coada nu mai este plina si se apeleaza metoda *signal()* care trezeste un thread ce asteapta;
    - \* se deblocheaza zavorul si se returneaza elementul extras.
  - *put(int elementValue, int producerNumber)* metoda folosita pentru a insera elemente in coada. Se realizeaza urmatoarele instructiuni:
    - \* se blocheaza zavorul cozii *queueLock.lock()*;
    - \* cat timp coada este plina se asteapta extragerea unui element din coada prin apelarea metodei *await()* pentru conditia *notFull*. Aceasta metoda face ca threadul respectiv sa astepte pana cand este notificat de un alt thread;
    - \* se adauga un element in coada, deci coada nu mai este goala si se apeleaza metoda *signal()* care trezeste un thread ce asteapta;
    - \* se deblocheaza zavorul cozii.
- **MainClass**: clasa ce contine functia *main* unde este creata coada, sunt creati producatorii si consumatorii si se incepe executia threadurilor.

## 2.2 Sincronizarea fina

Pentru rezolvarea problemei cu ajutorul sincronizarii fine am creat 4 clase:

- **SharedQueue**: clasa ce reprezinta coada in care se vor pune elementele si de unde vor fi extrase.
  - *maxLength* - dimensiunea cozii;
  - *elementsList* - lista in care se pun si din care se extrag elementele;
  - *cellLock* - un vector de zavoare pentru accesul la fiecare celula a listei;
  - *queueLock* - zavor pentru accesul la coada;
  - *notFull* si *notEmpty* - cele 2 conditii pentru cazul in care coada este plina sau goala;
  - *SharedQueue()* - constructorul clase;
  - *put(int value)* - functie ce adauga elementul *value* in coada. Mai intai se blocheaza zavorul pentru celula. Daca lista nu este plina (*elementsList.size() != maxLength*) se adauga elementul, iar daca lista este plina (*elementsList.size() == maxLength*) se asteapta pana cand se elibereaza un loc (*notFull.await()*) apoi se insereaza elementul si se elibereaza zavorul pentru celula.
  - *take()* - functie folosita pentru extragerea unui element din coada. Mai intai se blocheaza zavorul pentru celula, apoi se verifica daca lista are elemente in ea. In caz afirmativ se extrage elementul, iar in cazul in care nu exista niciun element in lista se asteapta pana se gaseste unul. La final, se elibereaza zavorul pentru celula.
  - **Producer**: clasa ce reprezinta producatorul si implementeaza interfata Runnable.
    - \* Fiecare producator este identificat prin variabila *number*;
    - \* In functia *run()* se insereaza un element in coada cu ajutorul functiei *put(elementValue)* iar apoi se asteapta 1000 de milisekunde.
  - **Consumer**: clasa ce reprezinta consumatorul si implementeaza interfata Runnable.
    - \* Fiecare consumator este identificat prin variabila *number*;
    - \* In functia *run()* se extrage un element din coada cu ajutorul functiei *take()* iar apoi se asteapta 1000 de milisekunde.
  - **ProducerExecutor**: clasa executor in care se incepe executia producatorului.
  - **ConsumerExecutor**: clasa executor in care se incepe executia consumatorului.
  - **MainClass**: clasa ce contine functia *main* de unde se incepe executia programului. Exista un numar de 4 producatori, iar numarul consumatorilor reprezinta numarul procesoarelor virtuale disponibile.

### 3 Experimental data



```
<terminated> MainClass (52) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (19 ian. 2020, 12:00)
Producatorul 1 a produs elementul 1 pe pozitia 0
Producatorul 2 a produs elementul 2 pe pozitia 1
Consumer 1 consumed 1
Consumer 2 consumed 2
Producatorul 2 a produs elementul 4 pe pozitia 2
Consumer 1 consumed 4
Producatorul 1 a produs elementul 2 pe pozitia 3
Consumer 2 consumed 2
Producatorul 1 a produs elementul 3 pe pozitia 4
Producatorul 2 a produs elementul 6 pe pozitia 0
Consumer 1 consumed 3
Consumer 2 consumed 6
Producatorul 2 a produs elementul 8 pe pozitia 1
Producatorul 1 a produs elementul 4 pe pozitia 2
Consumer 2 consumed 8
Consumer 1 consumed 4
Producatorul 2 a produs elementul 10 pe pozitia 3
Consumer 1 consumed 10
Producatorul 1 a produs elementul 5 pe pozitia 4
Consumer 2 consumed 5
Producatorul 2 a produs elementul 12 pe pozitia 0
Producatorul 1 a produs elementul 6 pe pozitia 1
Consumer 1 consumed 12
Consumer 2 consumed 6
Producatorul 1 a produs elementul 7 pe pozitia 2
Producatorul 2 a produs elementul 14 pe pozitia 3
Consumer 1 consumed 7
Consumer 2 consumed 14
Producatorul 1 a produs elementul 8 pe pozitia 4
Producatorul 2 a produs elementul 16 pe pozitia 0
Consumer 1 consumed 8
Consumer 2 consumed 16
Producatorul 1 a produs elementul 9 pe pozitia 1
Producatorul 2 a produs elementul 18 pe pozitia 2
Consumer 1 consumed 9
Consumer 2 consumed 18
Producatorul 2 a produs elementul 20 pe pozitia 3
Producatorul 1 a produs elementul 10 pe pozitia 4
Consumer 1 consumed 20
Consumer 2 consumed 10
Producatorul 2 a produs elementul 22 pe pozitia 0
Producatorul 1 a produs elementul 11 pe pozitia 1
<
```

## 4 Results & Conclusions

Am rezolvat problema "producer-consumer" utilizand sincronizarea grosiera cat si cea fina. Pentru sincronizarea fina a trebuit sa folosesc zavoare pentru accesul la fiecare element din coada, precum si un executor pentru producator, respectiv consumator.

In ambele metode am creat o coada in care sunt puse si din care sunt extrase elementele. In cazul sincronizarii grosiere, atat numarul producatorilor cat si cel al consumatorilor a fost 2. La sincronizarea fina, au existat 4 producatori, iar numarul consumatorilor a fost dat de numarul de procese virtuale disponibile.