Synchronization Mechanisms Lab Assignment

December 12, 2019

Student: Brinzan Florinel-Razvan

Calculatoare si tehnologia informatiei
(limba romana)

C.R 3.1 A

Anul 3

1 Problema 1 – Producator-Consumator

1.1 Enuntul problemei

— **Implementați problema Producator-Consumator folosind:**

1. **semafoare**
2. **monitoare**
3. **lacate**

Pe scurt, exista mai multi **producatori si consumatori**. **Fiecare producator va produce elemente (numere) distincte intr-un interval de numere dat si le va plasa intr-un buffer de marime fixa. Rolul consumatorilor este acela de a consuma elemente din buffer.**

—**Probleme ce pot aparea:**

- Producatorul vrea sa adauge un element in buffer in momentul in care acesta este full.
- Consumatorul vrea sa consume un element din buffer in momentul in care acesta este gol.

Pentru a evita aceste 2 probleme trebuie sa blocam producatorul sa mai produca elemente cat timp buffer-ul este plin si sa blocam consumatorul sa mai consume elemente din buffer cat timp acesta este gol.

1.2 Implementarea problemei in Java

1.2.1 Utilizarea semafoarelor

— In implementarea acestei probleme am creat 4 clase: clasa PCSemafor, clasa Producator, clasa Consumator si clasa Main.

1. **Clasa PCSemafor** detine cele **doua semafoare** prin intermediul carora **producatorii pot adauga elemente in buffer** (in limita spatiului disponibil - nu se poate suprascrie un element din buffer daca acesta nu a fost inca consumat) iar **consumtorii pot extrage elemente** (in limita stocului din buffer - nu se poate consuma un element ce a fost consumat anterior). **Un consumator nu poate extrage elemente din buffer daca acesta e "gol"** (are doar elemente ce au fost consumate de firul curent sau alte fire de executie) **deoarece este blocat de "semafor-consumator",iar producatorul nu poate adauga elemente in buffer daca acesta este "plin"** (are doar elemente ce nu au fost consumate inca) **fiindca este blocat de "semafor-producator".**

—**In clasa PCSemafor sunt implementate metodele de producere si consumare a elementelor buffer-ului.**

- **Metoda *void append(int)* – prin aceasta metoda Producatorul adauga elemente in buffer. Cat timp counter-ul (numarul de elemente neconsumate) este mai mic decat capacitatea buffer-ului se vor produce noi elemete** (acestea vor suprascrie elemente deja consumate din buffer si counter-ul ce numara elementele neconsumate va creste). **Daca counter-ul va avea valoarea capacitatii buffer-ului atunci firul de executie va fi blocat pana se va apela metoda *release()*. In momentul in care se adauga un nou element in buffer semaforul consumatorului primeste *release()*.**

PSEUDOCOD

```
cat timp counter == CAPACITATE executa
       blocheaza semafor_producator


buffer[newest] <- element primit ca parametru
newest <- ((newest + 1) % CAPACITATE)
counter <- (counter + 1)
elibereaza semafor_consumator
```

- **Metoda *int take()* – prin aceasta metoda Consumatorul extrage elemente din buffer. Cat timp counter-ul este mai mare decat 0 insemana ca exista elemente neconsumate in buffer, deci consumatorul le poate extrage** (incepand cu cel mai vechi element introdus)**.Daca counter-ul va avea valoarea 0 atunci firul de executie va fi blocat pana se va apela metoda *release()* pentru semafor-consumator. In momentul in care se extrage un element din buffer semaforul producatorului primeste *release()* - elibereaza permis-ul, returnandu-l semaforului.**

PSEUDOCOD

```
element <- 0
    cat timp counter == 0 executa
           blocheaza semafor_consumator

    element <- buffer[oldest];
    oldest <- (oldest + 1) % CAPACITATE;
    counter <- (counter - 1)
    elibereaza semafor_producator
    returneaza element
```

2. **Clasa Producator** extinde clasa **Thread** si **suprascrie metoda *"run()"***.Aceasta clasa are ca membrii de tip data elementele ce vor fi initializate cu valorile date ca parametri in constructor. **Clasa contine un constructor cu parametri, o metoda prin care se returneaza numarul de elemente produse si metoda *run()* ce apeleaza metoda *append(int)* din clasa PCSemafor pentru a adauga elemente in buffer.**

   - **Metoda *int getCounter()* returneaza numarul de elemente produse de firul curent.**

   - **Metoda *void run()* suprascrie metoda "run()" din clasa Thread. Metoda "*run()*" din clasa Producator are rolul de a genera elementul (un nr intreg din intervalul [min, max)) si a-l introduce in buffer prin intermediul metodei *append(int)* din clasa PCSemafor. La final se incrementeaza numarul de elemente produse de firul curent, se afiseaza numarul producatorului si numarul elementului adaugat in buffer.**

   PSEUDOCOD

```
element <- min
    cat timp (element % nr_fireP) != nr_fir executa
            element <- element +1

    cat timp (element < max) executa
            adauga element in buffer
            nr_elemente_produse <- nr_elemente_produse +1
            afiseaza ce producator a produs elementul
            element <- element + nr_fireP
```

3. **Clasa Consumator** extinde clasa **Thread** si **suprascrie metoda "*run()*"**. Aceasta clasa are ca membrii de tip data elementele ce vor fi initializate cu valorile date ca parametri in constructor. **Clasa contine un constructor cu parametri, o metoda prin care se returneaza numarul de elemente consumate si metoda *run()* ce apeleaza metoda *take()* din clasa PCSemafor pentru a extrage elemente din buffer.**

   - **Metoda int getCounter() returneaza numarul de elemente consumate de firul curent.**

   - **Metoda void *run()* suprascrie metoda "*run()*" din clasa Thread. Metoda "*run()*" din clasa Consumator are rolul de a extrage elemente din buffer prin intermediul metodei *take()* din clasa PCSemafor. La final se incrementeaza numarul de elemente consumate de firul curent, se afiseaza numarul consumatorului si numarul elementului consumat din buffer.**

PSEUDOCOD

```
iterator <- min
element <- 0

cat timp(iterator % nr_fireC) != nr_fir executa
       iterator <- iterator + 1

cat timp (iterator < max) executa

       element <- cel mai vechi element din buffer
       nr_elemente_consumate <-
           nr_elemente_consumate + 1
       afiseaza ce consumator a consumat elementul
       iterator <- iterator + nr_fireC
}
```

4. **Clasa Main** contine metoda "***main()***" in care **sunt instantiati producatorii si consumatorii.** Aceasta clasa contine 4 variabile statice - finale ce reprezinta **numarul de producatori (*NR-PRODUCATORI*)**, **numarul de consumatori (*NR-CONSUMATORI*), limita inferioara si superioara a intervalului de numere/produse (*MINN* respectiv *MAXN*)** . **Clasa Main** contine un obiect de tip **PCSemafor** ce reprezinta buffer-ul de elemente care va fi dat ca parametru **tuturor firelor de executie (producatori si consumatori)**.

   - In **metoda *main()*** sunt create thread-urile (producatorii si consumatorii) si este apelata metoda "***start()***" ce pune in executie thread-urile. **Fiecare producator / consumator primeste ca parametrii numarul de producatori/consumatori, numarul sau de identificare, limita inferioara si superioara a intervalului de numere(produse) si buffer-ul de numere(produse).** La final se afiseaza numarul de elemente produse de fiecare producator si numarul de elemente consumate de fiecare consumator.

PSEUDOCOD metoda main()

```
NR_PRODUCATORI <- 10
NR_CONSUMATORI <- 5
MINN <- 0
MAXN <- 50
iterator <- 0
Producator producatori[]
Consumator consumatori[]
PCSemafor buffer
```

```
pentru iterator de la 0 la NR_PRODUCATORI - 1 executa
       producatori[iterator] <- new
           Producator(NR_PRODUCATORI, iterator, MINN,
           MAXN, buffer)
       start producatori[iterator]

pentru iterator de la 0 la NR_CONSUMATORI - 1 executa
       consumatori[iterator] <- new
           Consumator(NR_CONSUMATORI, iterator, MINN,
           MAXN, buffer)
       start consumatori[iterator]

pentru iterator de la 0 la NR_PRODUCATORI - 1 executa
              join producatori[iterator]

pentru iterator de la 0 la NR_CONSUMATORI - 1 executa
              join consumatori[iterator]

pentru iterator de la 0 la NR_PRODUCATORI - 1 executa
       afiseaza Producatorul X a produs Y elemente

pentru iterator de la 0 la NR_CONSUMATORI - 1 executa
       afiseaza Consumatorul X a consumat Y elemente
```

1.2.2 Utilizarea lacatelor

—In implementarea acestei probleme am creat **4 clase**: clasa **PCLock**, clasa **Producator**, clasa **Consumator** si clasa **Main**.

—Clasele **Producator**, **Consumator** si **Main** au ramas neschimbate (cu precizarea ca in locul obiectului **PCSemafor** se va folosi un obiect de tip **PCLock**. **Descrierea acestor clase este prezentata mai sus.**

—Clasa **PCLock** detine lacatul, conditiile si cele doua metode (***append*** si ***take***) prin intermediul carora producatorii pot adauga elemente in **buffer** (in limita spatiului disponibil - nu se poate suprascrie un element din buffer daca acesta nu a fost inca consumat) iar consumtorii pot extrage elemente (in limita stocului din **buffer** - nu se poate consuma un element ce a fost consumat anterior). Un consumator nu poate extrage elemente din **buffer** daca acesta e "**gol**" (are doar elemente ce au fost consumate de firul curent sau alte fire de executie) deoarece va intra in bucla "**while**" si va trebui sa astepte pana va primi semnalul dat de metoda ***signalAll().*** Un producator nu poate adauga elemente in buffer daca acesta este "**plin**" (are doar elemente ce nu au fost consumate inca) fiindca va intra in bucla "**while**" si va trebui sa astepte pana va primi semnalul dat de metoda ***signalAll()***. La intrarea in cele 2 metode lacatul se va inchide si va fi deblocat doar la iesirea din metoda.

- **Metoda *void append(int)* – Prin aceasta metoda Producatorul adauga elemente in buffer. Cat timp counter-ul este mai mic decat capacitatea buffer-ului se vor produce noi elemete (acestea vor suprascrie elemente deja consumate din buffer si counter-ul ce numara elementele neconsumate va creste). Daca counter-ul va avea valoarea capacitatii buffer-ului atunci firul de executie va intra in bucla while si va astepta sa fie notificat. In momentul in care se adauga un element in buffer se apeleaza metoda *signalAll()* ce trezeste toate firele de executie. La apelarea acestei metode lacatul se va bloca si va fi deblocat doar la iesirea din metoda.**

PSEUDOCOD

```
blocheaza lacatul
cat timp (counter == CAPACITATE) executa
       asteapta
buffer[newest] <- element
newest <- (newest + 1) % CAPACITATE
counter <- counter + 1
trezeste firele
deblocheaza lacatul
```

- **Metoda *int take()* – Prin aceasta metoda Consumatorul extrage elemente din buffer. Cat timp counter-ul este mai mare decat 0 insemana ca exista elemente neconsumate in buffer, deci consumatorul le poate extrage (incepand cu cel mai vechi element introdus).Daca counter-ul va avea valoarea 0 atunci firul de executie va astepta sa fie notificat. In momentul in care se extrage un element din buffer se apeleaza metoda *signalAll()* ce trezeste toate firele de executie. La apelarea acestei metode lacatul se va bloca si va fi deblocat doar la iesirea din metoda.**

PSEUDOCOD

```
element <- 0
   blocheaza lacatul
   cat timp (counter == 0) executa
          asteapta

   element <- buffer[oldest]
   oldest <- (oldest + 1) % CAPACITATE
   counter <- counter - 1

   trezeste firele
   deblocheaza lacatul
   returneaza element
```

1.2.3 Utilizarea monitoarelor

—In implementarea acestei probleme am creat **4 clase**: clasa **PCMonitor**, clasa **Producator**, clasa **Consumator** si clasa **Main**.

—Clasele **Producator**, **Consumator** si **Main** au ramas neschimbate (cu precizarea ca in locul obiectului **PCSemafor** se va folosi un obiect de tip **PCMonitor**. **Descrierea acestor clase este prezentata mai sus.**

—Clasa **PCMonitor** detine cele **doua metode sincronizate (*append* si *take*) prin intermediul carora producatorii pot adauga elemente in buffer** (in limita spatiului disponibil - nu se poate suprascrie un element din buffer daca acesta nu a fost inca consumat) **iar consumtorii pot extrage elemente** (in limita stocului din buffer - nu se poate consuma un element ce a fost consumat anterior). Un consumator nu poate extrage elemente din buffer daca acesta e "**gol**" (are doar elemente ce au fost consumate de firul curent sau alte fire de executie) deoarece va intra in bucla "**while**" si va fi adormit, iar producatorul nu poate adauga elemente in buffer daca acesta este "**plin**" (are doar elemente ce nu au fost consumate inca) fiindca va intra in bucla "**while**" si va fi adormit. Asteptarea firelor se va incheia in momentul in care va fi apelata metoda ***notifyAll()*** ce **va trezi toate firele care asteapta pe monitorul obiectului**.

- **Metoda *synchronized void append(int)* – Prin aceasta metoda Producatorul adauga elemente in buffer. Cat timp counter-ul este mai mic decat capacitatea buffer-ului se vor produce noi elemete (acestea vor suprascrie elemente deja consumate din buffer si counter-ul ce numara elementele neconsumate va creste). Daca counter-ul va avea valoarea capacitatii buffer-ului atunci firul de executie va intra in bucla while si va astepta sa fie notificat. In momentul in care se adauga un element in buffer se apeleaza metoda *notifyAll()* ce trezeste toate firele care asteapta pe monitorul obiectului**.

PSEUDOCOD

```
cat timp (counter == CAPACITATE) executa
       asteapta
buffer[newest] <- element
newest <- (newest + 1) % CAPACITATE
counter <- counter + 1
trezeste firele
```

- **Metoda *synchronized int take()* – Prin aceasta metoda Consumatorul extrage elemente din buffer. Cat timp counter-ul este mai mare decat 0 insemana ca exista elemente neconsumate in**

buffer, deci consumatorul le poate extrage (incepand cu cel mai vechi element introdus).Daca counter-ul va avea valoarea 0 atunci firul de executie va astepta sa fie notificat. In momentul in care se extrage un element din buffer se apeleaza metoda *notifyAll()* ce trezeste toate firele care asteapta pe monitorul obiectului.

PSEUDOCOD

```
element <- 0
    cat timp (counter == 0) executa
           asteapta

    element <- buffer[oldest]
    oldest <- (oldest + 1) % CAPACITATE
    counter <- counter - 1

    trezeste firele
    returneaza element
```

1.3 Rezultate, Scenarii posibile si descrierea output-ului

—**Sunt 3 scenarii posibile:**

1. **Producatorii** si **Consumatorii** produc si consuma elemente simultan si nu se va ajunge niciodata la limitele buffer-ului (0 elemente respectiv 5 elemente) - **scenariu mai deloc intalnit.**
2. **Producatorii vor aveam momente cand vor vrea sa produca elemente si nu va mai fi spatiu disponibil in buffer** (vor fi doar elemente neconsumate si nu le vor putea suprascrie). In acest caz producatorii trebuie opriti din a mai produce elemente (folosirea metodelor ***acquire(), await(), wait(), lock()***). **Producatorii vor incepe sa produca elemente numai atunci cand bufer-ul are cel putin un element deja consumat ce va putea fi suprascris).**

   - Implementarea solutiilor pentru acest scenariu:

```
public void append(int element) {
    while (counter == CAPACITATE) {
        try {
            semafor_producator.acquire();
        } catch (InterruptedException e) { }
    }
    buffer[newest] = element;
    newest = (newest + 1) % CAPACITATE;
    counter++;
    semafor_consumator.release();
}
```

==========

```
public synchronized void append(int element) {
    while (counter == CAPACITATE) {
        try {
            wait();
        } catch (InterruptedException e){}
    }
    buffer[newest] = element;
    newest = (newest + 1) % CAPACITATE;
    counter++;
    notifyAll(); // signal not zero
}
```

==========

```
public void append(int element) {

    lock.lock();
    try {

        while (counter == CAPACITATE) {
            conditie1.await();
        }

        buffer[newest] = element;
        newest = (newest + 1) % CAPACITATE;
        counter++;

        conditie2.signalAll();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }finally {
        lock.unlock();
    }
}
```

3. **Consumatorii vor avea momente cand vor dori sa consume elemente din buffer dar buffer-ul s-ar putea sa fie gol** (are doar elemente deja consumate). In acest caz consumatorii trebuie opriti din a mai consuma elemente, iar acest lucru este posibil prin apelarea metodelor ***acquire(), await(), wait(), lock()***. **Consumatorii vor incepe sa consume din nou elemente cand cel putin un producator va pune in buffer un elemnt nou (atunci vor fi "treziti/deblocati" consumatorii ce asteapta).**

- Implementarea solutiilor pentru acest scenariu:

```java
public int take() {
    int element;
    while (counter == 0) {
        try {
            semafor_consumator.acquire();
        } catch (InterruptedException e) { }
    }
    element = buffer[oldest];
    oldest = (oldest + 1) % CAPACITATE;
    counter--;
    semafor_producator.release();
    return element;
}
```

==========

```java
public synchronized int take() {
    int element;
    while (counter == 0) {
        try {
            wait();
        } catch (InterruptedException e){}
    }
    element = buffer[oldest];
    oldest = (oldest + 1) % CAPACITATE;
    counter--;
    notifyAll(); // signal not full
    return element;
}
```

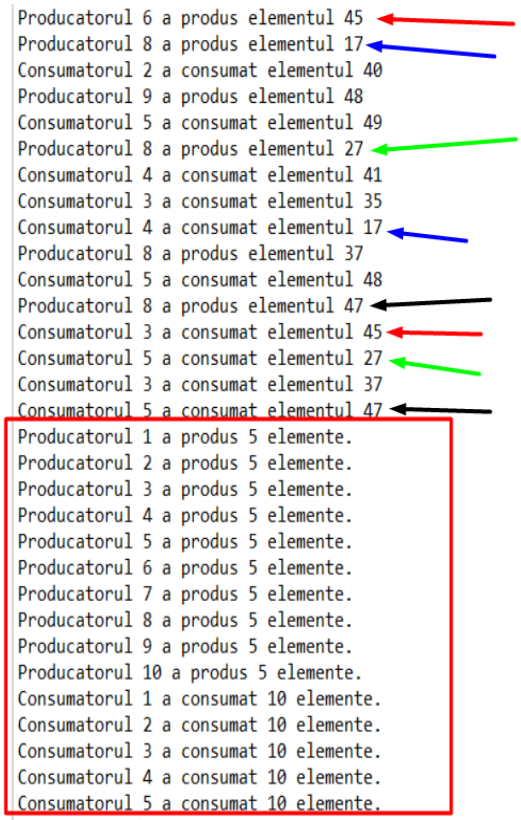
==========

```java
public int take() {

    int element = 0;

    lock.lock();
    try {
        while (counter == 0) {
            conditie2.await();
        }
        element = buffer[oldest];
```

- **In consola se afiseaza numarul producatorului si ce element a produs precum si numarul consumatorului si ce element a consumat.**

- **La final se calculeaza totalul de elemente produse de fiecare producator si totalul de elemente consumate de fiecare consumator.**

.



1.4 Concluzii

- Implementand aceasta problema mi-am fixat mai bine cunostintele legate de programarea concurenta, am invatat lucruri noi despre comportamentul firelor de executie cat si despre restrictiile ce ar putea fi aplicate asupra firelor astfel incat rezultatul sa fie cel dorit.

- Utilizarea mecanismelor de sincronizare este extrem de importanta in programarea concurenta. Studiind aceste mecanisme mi-am imbunatatit abilitatile de programare concurenta.

- Consider ca aceste tipuri de exercitii au fost foarte bine alese pentru partea aceasta de studiu a mecanismelor de sincronizare ale firelor de executie deoarece implementand solutiile unor cerinte de acest tip ajungi sa intelegi cu adevarat riscurile executiei unui program pe mai multe fire, stii la ce sa te astepti si cum ai putea rezolva anumite "conflicte" dintre firele de executie.