SPRAWOZDANIE III

TEORIA WSPÓŁBIEŻNOŚCI

Oczekiwanie na warunek (*condition wait*)

DAWID BIAŁKA

DATA LABORATORIUM 20.10.2020

DATA ODDANIA 27.10.2020

Zad. 1 Dany jest bufor, do którego producent może wkładać dane, a konsument pobierać. Napisać program, który zorganizuje takie
działanie producenta i konsumenta, w którym zapewniona będzie własność bezpieczeństwa i żywotności.
Zrealizować program:

1. przy pomocy metod wait()/notify(). Kod szkieletu
    a. dla przypadku 1 producent/1 konsument
    b. dla przypadku **n1** producentów/**n2** konsumentów (n1>n2, n1=n2, n1<n2)
    c. wprowadzić wywołanie metody sleep() i wykonać pomiary, obserwując zachowanie producentów/konsumentów
2. przy pomocy operacji P()/V() dla semafora:
    a. n1=n2=1
    b. n1>1, n2>1

Zad. 2 Przetwarzanie potokowe z buforem

- Bufor o rozmiarze N - wspolny dla wszystkich procesow!
- Proces A bedacy producentem.
- Proces Z bedacy konsumentem.
- Procesy B, C, ..., Y bedace procesami przetwarzajacymi. Kazdy proces otrzymuje dana wejsciowa od procesu poprzedniego, jego wyjscie zas jest konsumowane przez proces nastepny.
- Procesy przetwarzaja dane w miejscu, po czym przechodza do kolejnej komorki bufora i znowu przetwarzaja ja w miejscu.
- Procesy dzialaja z roznymi predkosciami.

Uwaga:

1. W implementacji nie jest dozwolone korzystanie/implementowanie wlasnych kolejek FIFO, nalezy uzywac tylko mechanizmu monitorow lub semaforow !
2. Zaimplementowac rozwiazanie przetwarzania potokowego (Przykladowe zalozenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesow przetwarzajacych.) Od czego zalezy predkosc obrobki w tym systemie ? Rozwiązanie za pomocą semaforów lub monitorów (dowolnie). **Zrobić sprawozdanie z przetwarzania potokowego**.

Zad 1. a

Koncepcja

W pierwszym przypadku wykorzystujemy obszar synchronized i metody wait() oraz notify(). Program uruchamiamy najpierw dla 1 konsumenta i producenta, później dla przypadku n1>n2 i n2<1 i n1=n2.

Implementacja i wyniki

```
public class Main {

    public static void main(String[] args) {

        int n1 = 1;
        int n2 = 1;

        Buffer buffer = new Buffer(5);

        ExecutorService service = Executors.newFixedThreadPool(n1 + n2);


        for(int i=0; i<n2; i++) {
            service.submit(new Consumer(buffer));
        }

        for(int i=0; i<n1; i++) {
            service.submit(new Producer(buffer));
        }

        service.shutdown();

    }
}
```

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

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

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.put(i);
            try {
                sleep((int) (Math.random() * 70));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

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

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

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.get();
            try {
                sleep((int) (Math.random() * 70));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class Buffer {

   private final int maxSize;
   private List<Integer> bufferList = new ArrayList<Integer>();

   public Buffer(int maxSize) {
      this.maxSize = maxSize;
   }

   public synchronized void put(int product) {
      while(bufferList.size() >= this.maxSize) {
         try {
            System.out.println("Thread " + Thread.currentThread().getId() + " waiting to put items");
            wait();
         } catch(InterruptedException e) {
            e.printStackTrace();
         }
      }

      System.out.println("Thread " + Thread.currentThread().getId() + " adds item" + product);
      bufferList.add(product);

      notify();
   }

   public synchronized int get() {
      while (bufferList.size() <= 0) {
         try {
            System.out.println("Thread " + Thread.currentThread().getId() + " waiting to get items");
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      int lastIndex = bufferList.size() - 1;
      int returnValue = bufferList.get(lastIndex);

      System.out.println("Thread " + Thread.currentThread().getId() + " gets item" + returnValue);
      bufferList.remove(lastIndex);

      notify();

      return returnValue;
   }
}
```

n1 – liczba producentów, n2 – liczba konsumentów

Dla przypadku n1=n2=1 i rozmiaru bufora 5 mamy niewiele wystąpień sytuacji, gdzie producent lub konsument czekają na możliwość wykonania odpowiadającej im operacji.

Dla n1=10 i n2=2 i rozmiaru bufora 30 żaden konsument ani razu nie czekał na dodanie przedmiotów przez producenta. Za to co chwilę pojawiała się sytuacja, że producent czeka na zwolnienie bufora. Ostatecznie program nie kończy się, gdyż producenci łącznie chcą wyprodukować 10 * 100 = 1000 przedmiotów, a konsumenci odebrać tylko 2 * 100 = 200 przedmiotów.

Dla n1=2 i n2=10 i rozmiaru bufora 30 mamy sytuację odwrotną, czyli bardzo często konsumenci czekają na dodanie przedmiotu i ostatecznie program również się nie kończy, gdyż po wyprodukowaniu wszystkich przedmiotów (200 sztuk), nadal mamy konsumentów, którzy chcą pobrać 800 sztuk.

Aby tego uniknąć należy dobrać liczbę iteracji dla każdego producenta i konsumenta tak, aby łączna liczba wyprodukowanych i skonsumowanych przedmiotów była równa. (w szkielecie załączonym przez Pana Doktora w pętli w klasie Producent i Consumer było 100 iteracji, dlatego tak też to pozostawiłem).

Gdy n1=n2=10 i rozmiar bufora 30 to mamy tak samo jak dla sytuacji n1=n2, czyli rzadko występuje sytuacja, aby jakiś konsument lub producent czekał na wykonanie swojej operacji.

Wnioski

Zgodnie z oczekiwaniami, gdy mamy za dużo producentów, to często producenci muszą czekać. Gdy mamy za dużo konsumentów, to konsumenci często czekają. W przypadku, gdy mamy ich po równo wtedy sytuacja czekania występuje dość rzadko. Gdy mamy n1=n2 program kończy się, co znaczy, że działa on prawidłowo ponieważ wszystkie produkty zostały wyprodukowane i każdy produkt został skonsumowany, żaden produkt nie „zniknął” w czasie działania ponieważ w takiej sytuacji wątki konsumenta nadal oczekiwałyby na produkt.

Zad 1. b

Koncepcja

W drugim przypadku wykorzystujemy metody P() i V(). Uruchamiamy program dla n1=n2=1 i n1>1 oraz n2>2.

Implementacja i wyniki

```
public class Main {

    public static void main(String[] args) {

        int n1 = 10;
        int n2 = 2;

        Semaphore consumerSemaphore = new Semaphore(0);

        Semaphore producerSemaphore = new Semaphore(30);

        BinarySemaphore binarySemaphore = new BinarySemaphore();

        Buffer buffer = new Buffer(consumerSemaphore, producerSemaphore, binarySemaphore);

        ExecutorService service = Executors.newFixedThreadPool(n1 + n2);


        for(int i=0; i<n2; i++) {
            service.submit(new Consumer(buffer));
        }

        for(int i=0; i<n1; i++) {
            service.submit(new Producer(buffer));
        }

        service.shutdown();

    }
}
```

```java
public class Buffer {

    private final Semaphore producerSem;
    private final Semaphore consumerSem;
    private final BinarySemaphore binSem;
    private List<Integer> bufferList = new ArrayList<Integer>();

    public Buffer(Semaphore consumerSem, Semaphore producerSem, BinarySemaphore binSem) {
        this.consumerSem = consumerSem;
        this.producerSem = producerSem;
        this.binSem = binSem;
    }

    public void put(int product) {
        producerSem.P(0); //zmniejszamy liczbe wolnych miejsc na nowe przedmioty
        // bo w tym momencie tworzymy przedmiot, w celu wypisania odpowiedniego komunikatu
        // w razie czekania 0 w argumencie oznacza, ze jest to producent
        // 1 to konsument
        binSem.P(); //do modyfikacji tablicy bufferList ma dostep na raz tylko jeden watek

        System.out.println("Thread " + Thread.currentThread().getId() + " puts item" + product);
        bufferList.add(product);

        consumerSem.V(); //zwiekszamy liczbe przedmiotow ktore konsumenci moga pobrac
        binSem.V(); //oddajemy dostep do modyfikacji tablicy bufferList
    }

    public int get() {
        consumerSem.P(1); //zmniejszamy liczbe przedmiotow do pobrania przez konsumentow
        binSem.P(); //do modyfikacji tablicy bufferList ma dostep na raz tylko jeden watek

        int lastIndex = bufferList.size() - 1;
        int returnValue = bufferList.get(lastIndex);

        System.out.println("Thread " + Thread.currentThread().getId() + " gets item" + returnValue);
        bufferList.remove(lastIndex);

        producerSem.V(); //zwiekszamy liczbe miejsc na nowe przedmioty bo wlasnie pobralismy przedmiot
        binSem.V(); //oddajemy dostep do modyfikacji tablicy bufferList

        return returnValue;
    }
}
```

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

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

   public void run() {
      for (int i = 0; i < 100; ++i) {
         _buf.put(i);
         try {
            sleep((int) (Math.random() * 70));
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

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

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

   public void run() {
      for (int i = 0; i < 100; ++i) {
         _buf.get();
         try {
            sleep((int) (Math.random() * 70));
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

```
public class Semaphore {
   int counter;

   public Semaphore(int startingValue) {
      this.counter = startingValue;
   }

   public synchronized void P(int who) {
      while (counter <=0) {
         try {
            if(who == 0) {
               System.out.println("Thread " + Thread.currentThread().getId() + " waiting to put items");
            }
            else {
               System.out.println("Thread " + Thread.currentThread().getId() + " waiting to get items");
            }
            wait();
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      this.counter--;
   }

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

```
public class BinarySemaphore {
   private boolean _stan = true;

   public BinarySemaphore() {
   }

   public synchronized void P() {
      while(!_stan) {
         try {
            wait();
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      _stan = false;

   }

   public synchronized void V() {
      _stan = true;
      notify();
   }

}
```

Dla n1=n2=1 podobnie jak w punkcie a nie ma za wielu sytuacji, aby producent lub konsument musieli czekać. Dla n1=10 i n2=2 również jest tak jak w poprzednim punkcie, często występuje sytuacja, że producent musi czekać.

Wnioski

Przy użyciu semaforów również uzyskujemy poprawne wyniki i również dla przypadku, kiedy jest więcej producentów niż konsumentów często występuje sytuacja, że producent musi czekać na zwolnienie miejsca.

Zad 2

Koncepcja

Jak w poprzednich zadaniach, tworzymy bufor (tablica) w której tym razem będziemy przechowywać obiekty specjalnej klasy Item. Będziemy w niej zapisywać, ile razy dany przedmiot został przetworzony, aby odpowiednie procesy mogły w odpowiedniej kolejności przetwarzać przedmiot. Na samym końcu również konsument wykorzysta tą informację, że dany przedmiot jest gotowy do odebrania. Producent będzie na kolejnych, wolnych miejscach tworzył nowe przedmioty. Każdy proces odpowiadający za dany stopień przetwarzania ma zapisaną ostatnią pozycję, którą przetwarzał. Gdy proces przetworzy przedmiot to przesuwa się o jedną pozycję w prawo i czeka, aż pojawi się tam przedmiot z odpowiednim poziomem przetworzenia. Gdy dojdziemy na koniec tablicy, przesuwamy się na sam początek. Klasa Procesor odpowiada za przetwarzanie. Im wyższy poziom przetwarzania, tym proces ma ustawianą wyższą wartość argumentu sleep().

Implementacja i wyniki

```
public class Main {

    public static void main(String[] args) {

        int n1 = 1;
        int n2 = 1;
        int n3 = 5;


        Buffer buffer = new Buffer(30);

        ExecutorService service = Executors.newFixedThreadPool(n1 + n2 + n3);

        for(int i=0; i<n1; i++) {
            service.submit(new Producer(buffer));
        }

        for(int i=0; i<n2; i++) {
            service.submit(new Consumer(buffer));
        }

        for(int i=0; i<n3; i++) {
            service.submit(new Processor(buffer, i, 30+i*6));
        }

        service.shutdown();

    }
}
```

```
public class Buffer {

   private List<Item> bufferList = new ArrayList<>();
   private final int maxSize;
   private int numOfItems = 0;
   private int lastFreeSlot = 0;

   public Buffer(int maxSize) {
      this.maxSize = maxSize;
      for(int i=0; i<maxSize; i++) {
         bufferList.add(null);
      }
   }

   public synchronized void put(Item item) {
      while(numOfItems >= this.maxSize) {
         try {
            System.out.println("Thread " + Thread.currentThread().getId() + " waiting to put items");
            wait();
         } catch(InterruptedException e) {
            e.printStackTrace();
         }
      }
      for(int i=lastFreeSlot ;; i = (i+1) % maxSize) {
         if(bufferList.get(i) == null) {

            bufferList.set(i, item);
            lastFreeSlot = i;

            numOfItems++;

            System.out.println("Thread " + Thread.currentThread().getId() + " adds item of value " +
item.getValue() + " at position " + i);
            break;
         }
      }
      notify();
   }

   public synchronized Item get(Consumer consumer){
      while (numOfItems <= 0) {
         try {
            System.out.println("Thread " + Thread.currentThread().getId() + ".Number of items = 0 in
consumer");
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      int lastPosition = consumer.getLastPosition();
      Item item = bufferList.get(lastPosition);

      try {
         while (item == null || item.getStageOfProduction() != 5) {
            System.out.println("Thread " + Thread.currentThread().getId() + ".Item not fully processed or not
present");
```

```
            wait();
        }

        bufferList.set(lastPosition, null);
        System.out.println("Thread " + Thread.currentThread().getId() + " gets item with processed value of
" + item.getValue() + "at position " + lastPosition);
        consumer.setLastPosition((lastPosition + 1) % maxSize);
        numOfItems--;

        notify();

      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      return null;
    }

    public synchronized void processItem(Processor processor) throws Exception {
      while (numOfItems <= 0) {
        try {
          System.out.println("Thread " + Thread.currentThread().getId() + ".Number of items = 0 in buffer:
proccesor");
          wait();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }

      int lastPosition = processor.getLastPosition();
      Item item = bufferList.get(lastPosition);

      while (item == null || item.getStageOfProduction() != processor.getStageOfProductionID()) {
        wait();
      }
      System.out.println("Thread " + Thread.currentThread().getId() + "procceses item " + item.getValue() +
" at position " + lastPosition + ". Stage " +
          "of production " + processor.getStageOfProductionID());

      item.process();

      System.out.println("Thread " + Thread.currentThread().getId() + "proccesed item at position" +
lastPosition + ".New value" +
          " " + item.getValue());
      processor.setLastPosition((lastPosition + 1) % maxSize);
      notify();
    }
}
```

```
public class Processor extends Thread{
    private Buffer _buf;
    private final int stageOfProductionID;
    private int lastPosition = 0;
    private int sleepTime;

    public Processor(Buffer buffer, int ID, int sleepTime) {
        this._buf = buffer;
        this.stageOfProductionID = ID;
        this.sleepTime = sleepTime;
    }

    public int getLastPosition(){
        return lastPosition;
    }

    public int getStageOfProductionID() {
        return stageOfProductionID;
    }

    public void setLastPosition(int lastPosition) {
        this.lastPosition = lastPosition;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            try {
                _buf.processItem(this);
            } catch (Exception e) {
                e.printStackTrace();
            }
            try {
                sleep((int) (Math.random() * sleepTime));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class Item {
    private int value;
    private int stageOfProduction = 0;

    public Item(int val) {
        this.value = val;
    }

    public int getValue() {
        return value;
    }

    public int getStageOfProduction() {
        return stageOfProduction;
    }

    public synchronized void process() throws Exception{
        if(stageOfProduction == 0) {
            this.value += 5;
        } else if(stageOfProduction == 1) {
            this.value *= 6;
        } else if(stageOfProduction == 2) {
            this.value += 21;
        } else if(stageOfProduction == 3) {
            this.value -= 3;
        } else if(stageOfProduction == 4) {
            this.value /= 2;
        } else throw new Exception("Invalid stage of production");

        stageOfProduction++;
    }
}
```

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

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

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.put(new Item(i));
            try {
                sleep((int) (Math.random() * 70));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
class Consumer extends Thread {
    private Buffer _buf;
    private int lastPosition = 0;
    private int counter = 0;

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

    public int getLastPosition(){
        return lastPosition;
    }

    public void setLastPosition(int lastPosition) {
        this.lastPosition = lastPosition;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.get(this);
            try {
                sleep((int) (Math.random() * 70));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Na samym końcu mamy na konsoli wywołanie, że konsument pobrał przedmiot o wartości 321, czyli tyle, ile wyjdzie z przejścia przez wszystkie stany przetwarzania dla liczby 99, czyli ostatniego dodanego przedmiotu.

Wnioski

Na podstawie powyższej obserwacji możemy stwierdzić, że algorytm działa poprawnie. Wszystkie etapy przetwarzania wykonują się w odpowiedniej kolejności. Szybkość działania programu będzie zależeć od tego, w jakiej kolejności będą budzone wątki. Gdy np. mamy sytuację, że przedmiot nie był jeszcze przetwarzany to gdy wątki obudzą się w kolejności ich poziomu przetwarzania to żaden nie będzie czekał tylko od razu będzie przetwarzał przedmiot. Ale możemy mieć sekwencję, że pod rząd obudzą się wątki, które przetwarzają przedmiot na wyższym albo niższym poziomie niż się on aktualnie znajduje (mogą też się obudzić kilka razy te same zanim obudzi się ten, który powinien przetwarzać dany przedmiot) to czas trwania będzie dłuższy.