Laboratorium 3

Łukasz Wala
AGH, Wydział Informatyki, Elektroniki i Telekomunikacji
Teoria Współbieżności 2022/23

Kraków, 7 listopada 2022

1 Treść zadania 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*.
   (a) dla przypadku 1 producent/1 konsument,
   (b) dla przypadku n_1 producentów/n_2 konsumentów ($n_1 > n_2$, $n_1 = n_2$, $n_1 < n_2$),
   (c) wprowadzić wywołanie metody *sleep* i wykonać pomiary, obserwując zachowanie producentów/konsumentów,
2. przy pomocy operacji P/V dla semafora:
   (a) $n_1 = n_2 = 1$
   (b) $n_1 > 1$, $n_2 > 1$.

2 Implementacja - *wait* i *notify*

Poniżej implementacja z użyciem *wait* i *notify*.

```
class Producer extends Thread {
    private Buffer buf;
    private int iters;

    public Producer(Buffer buf, int iters) {
        this.buf = buf;
        this.iters = iters;
    }
```

```
    public void run() {
        for (int i = 0; i < iters; ++i) {
        buf.put(i);
        }
    }
}

class Consumer extends Thread {
    private Buffer buf;
    private int iters;

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

    public void run() {
        for (int i = 0; i < iters; ++i) {
        System.out.println(buf.get());
        }
    }
}

class Buffer {
    private List<Integer> buf = new ArrayList<Integer>();
    private int size;

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

    public synchronized void put(int i) {
        while (buf.size() >= size) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }

        buf.add(i);
        notify();
    }

    public synchronized int get() {
```

```
        while (buf.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }
        int index = new Random().nextInt(buf.size());
        int returnVal = buf.get(index);
        buf.remove(index);
        notify();
        return retVal;
    }
}

public class PKmon {
    public static void main(String[] args) throws InterruptedException{
        Buffer buf = new Buffer(50);
        int iterations = 100;

        Consumer consumer = new Consumer(buf, iterations);
        Producer producer = new Producer(buf, iterations);

        producer.start();
        consumer.start();
        producer.join();
        consumer.join();
    }
}
```

Dla jenego producenta i jednego konsumenta implementacja działa poprawnie, konsumer wypisuje wszystkie wartości zapisane w bufferze przez producenta.

Teraz można rozważyć przypadki dla licz productenów i konsumerów różnych od 1 (klasy *Consumer*, *Producer* oraz *Buffer* pozostają bez zmian):

```
public class PKmon {
    public static void main(String[] args) throws InterruptedException {
        Buffer buf = new Buffer(100);
        int n1 = 5;
        int n2 = 5;

        int iters1 = 100;
        int iters2 = 100;

        if (n1*iters1 != n2*iters2) throw new RuntimeException("Invalid parameters");
```

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

        for(int i=0; i<n1; ++i) {
            service.submit(new Producer(buf, iters1));
        }

        for(int i=0; i<n2; ++i) {
            service.submit(new Consumer(buf, iters2));
        }

        service.shutdown();
    }
}
```

Gdy liczba konsumentów jest mniejsza od liczby producentów, buffer szybko się przepełnia i producenci muszą czekać. Gdy natomist liczna producentów jest mniejsza, część z konsuntów nie musi wykonywać żadnej pracy. Zasoby są wykorzystane najbardziej optymalnie, gdy liczna konsumentów i producentów jest zbliżona.

Do implementacji można dodać kilka wywołań *sleep* i zmierzyć czas wykonania przy takiej samej sumarycznej iteracji produkowania, ale zmieniając proporcę producentów do konsumentów:

```
class Producer extends Thread {
    private Buffer buf;
    private int iters;

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

    public void run() {
        for (int i = 0; i < iters; ++i) {
        buf.put(i);
        try {
            sleep(100);
        } catch (InterruptedException e) {
            System.exit(0);
        }
        }
    }
}
class Consumer extends Thread {
    private Buffer buf;
    private int iters;
```

```
    public Consumer(Buffer buf, int iters) {
        this.buf = buf;
        this.iters = iters;
    }

    public void run() {
        for (int i = 0; i < iters; ++i) {
        System.out.println(buf.get());
        try {
            sleep(100);
        } catch (InterruptedException e) {
            System.exit(0);
        }
        }
    }
}
public class PKmon {
    public static void main(String[] args) throws InterruptedException {
        Buffer buf = new Buffer(100);
        int n1 = 3;
        int n2 = 3;

        int iters1 = 70;
        int iters2 = 70;

        if (n1*iters1 != n2*iters2) throw new RuntimeException("Invalid parameters");

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

        final long startTime = System.currentTimeMillis();

        for(int i=0; i<n1; ++i) {
            service.submit(new Producer(buf, iters1));
        }

        for(int i=0; i<n2; ++i) {
            service.submit(new Consumer(buf, iters2));
        }

        service.shutdown();
        while (!service.awaitTermination(24L, TimeUnit.HOURS))
            ;
        final long endTime = System.currentTimeMillis();
        System.out.println(endTime - startTime);
    }
```

```
}
```

Poniżej krótka tabela średnich czasów wykonania dla poszczególnych konfiguracji:

Producenci (l. wątków × l. iteracji)	Konsumerzy (l. wątków × l. iteracji)	Czas wykonania (ms)
7×30	7×30	3017
3×70	7×30	7114
7×30	3×70	7025
3×70	3×70	7028

Jak widać, pomimo tego że w obu przypadkach, gdy liczna wątków konsumentów i producentów jest nierówna, czas wykonania jest prawie taki sam jak gdyby użyć mniejszej liczby wątków dla obu grup wątków. Dopiero gdy zwiększona zostanie zarówno liczna producentów jak i konsumentów, program działa szybciej, co powierdza wcześniejsze obserwacje.

3 Implementacja - semafory

Poniżej implementacja z użyciem samafor zaimplementowanych w sprawozdaniu do laboratorium 2:

```
class Buffer {
    private List<Integer> buf = new ArrayList<Integer>();
    private CountingSemaphore empty;
    private CountingSemaphore full;

    public Buffer(int size) {
        this.empty = new CountingSemaphore(size);
        this.full = new CountingSemaphore(0);
    }

    public synchronized void put(int i) {

        empty.P();
        buf.add(i);
        full.V();
    }

    public synchronized int get() {

        full.P();
        int index = new Random().nextInt(buf.size());
        int returnVal = buf.get(index);
```

```
        buf.remove(index);
        empty.V();
        return retVal;
    }
}
```

Implementacja ta używa dwóch semafor licznikowych (i bloku *synchronized*, który też możnaby zastąpić semaforą): jedną do sprawdzania czy buffer jest pusty, a drugi do sprawdzania czy jest pełny. Dla różnych liczb producentów i konsumentów występują tu podobne zjawiska, co w poprzedniej implementacji.

4 Treść zadania 2

Zaimplementować rozwiązanie przetwarzania potokowego (przykładowe założenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesów przetwarzających). Od czego zależy prędkość obróbki w tym systemie? Rozwiązanie za pomocą semaforów lub monitorów (dowolnie).

5 Implementacja

W tym zadaniu nieoptymalnym byłoby używanie jednego semafora blokującego cały bufor. Z tego powodu dla każdego elementu bufora tworzyony jest osobny semafor/monitor, dzięki temu jeden wątek przetwarzający element pod danym indeksem bufora nie blokuje innych wątków jednocześnie uniemożliwiając im dostęp do elementu pod danym indeksem.

```
class Buffer {
    private List<Integer> buf = new ArrayList<Integer>();
    private CountingSemaphore empty;
    private CountingSemaphore full;
    private List<BinarySemaphore> indexSemaphores = new ArrayList<BinarySemaphore>();

    public Buffer(int size) {
        this.empty = new CountingSemaphore(size);
        this.full = new CountingSemaphore(0);
        for (int i=0; i<size; ++i) {
            indexSemaphores.add(new BinarySemaphore());
        }
    }

...
public synchronized void putIndex(int index, int value) {
    buf.set(index, value);

    indexSemaphores.get(index).V();
```

```
}

public synchronized int getIndex(int index) {
    indexSemaphores.get(index).P();

    return buf.get(index);
}

}
```

W owym rozwiązaniu wątek przetwarzający blokuje cały bufor tylko podczas odczytywania i zapisywania. Gdy odczytuje od element pod danym indeksem, blokuje go, przez co wątek kolejny w kolejce do przetwarzania czeka, aż przetworzona wartość zostanie zapisana przez wątek poprzedni. Z tego też powodu w takim rozwiązaniu prędkość całego programu jest uzależniona od najwolniejszego z wątków przetwarzających.

6 Wnioski

Semafory oraz monitory są prostym i skutecznym sposobem na zsynchronizowanie wątków operujących na współdzielonym zasobie, w tym przypadku buforze. Zastosowanie semafor liczących pozwala również na zapewnienie, ażeby pewnę warunki potrzebne do wykonania operacji na zasobie były spełnione (w tym przypadku nie wyciąganie z pustego bufora oraz nie przepełnienie go). Warto jednak uważać, gdyż niekiedy, pomimo zapewnienia bezpieczeństwa, łatwo niepotrzebnie blokować zasób, kiedy jest to nie potrzebne, znacznie spowalniając działanie programu (np. blokując cały bufor podczas przetwarzania w przetwarzaniu potokowym z buforem).

7 Bibliografia

1. Dokumentacja języka Java - docs.oracle.com
2. C.A.R. Hoare; 1974; *Monitors: An Operating System Structuring Concept*