Laboratorium 12 - CSP

Piotr Karamon

27.01.2025.

Treść zadania

Zaimplementuj w Javie z użyciem JCSP rozwiązanie problemu producenta i konsumenta z buforem N-elementowym tak, aby każdy element bufora był reprezentowany przez odrębny proces (taki wariant ma praktyczne uzasadnienie w sytuacji, gdy pamięć lokalna procesora wykonującego proces bufora jest na tyle mała, że mieści tylko jedną porcję). Uwzględnij dwie możliwości:

1. kolejność umieszczania wyprodukowanych elementów w buforze oraz kolejność pobierania nie mają znaczenia.
2. pobieranie elementów powinno odbywać się w takiej kolejności, w jakiej były umieszczane w buforze
3. proszę wykonać pomiary wydajności kodu dla obu przypadków, porównać wydajność z własną implementacją rozwiązania problemu.
4. napisać sprawozdanie dotyczące rozwiązania problemu oraz przeprowadzić analizę wyników eksperymentów.

Rozwiązanie

Wariant gdy kolejność nie ma znaczenia

- Każda komórka bufora posiada trzy kanały:
  - Kanał producenta - do zapisu danych
  - Kanał konsumenta - do odczytu danych
  - Kanał "jeszcze" - sygnalizujący dostępność
- Każda komórka działa w nieskończonej pętli, w której:
  1. Sygnalizuje przez kanał "jeszcze", że jest wolna.
  2. Czeka aż:
     (a) Producent zapisze do niej dane (przez swój kanał).
     (b) Konsument odczyta dane (przez swój kanał)
- Producent:
  1. Używa klasy `Alternative` aby wybrać obojętnie którą wolną komórkę. Jeśli nie ma wolnych to czeka.
  2. Po uzyskaniu dostępu:

(a) Oznacza komórkę jako zajętą.
(b) Zapisuje do komórki dane

- Konsument:
  1. Jak producent używa `Alternative` aby znaleźć komórkę, która ma dane do oczytu. Jeśli nie ma to czeka.
  2. Po odczytaniu oznacza komórkę jako wolną.

Kod komórki bufora:

```
class Buffer implements CSProcess {
    private final One2OneChannelInt in;
    private final One2OneChannelInt out;
    private final One2OneChannelInt jeszcze;

    public Buffer(One2OneChannelInt in,
                  One2OneChannelInt out,
                  One2OneChannelInt jeszcze) {
        this.out = out;
        this.in = in;
        this.jeszcze = jeszcze;
    }

    public void run() {
        while (true) {
            jeszcze.out().write(0);
            out.out().write(in.in().read());
        }
    }
}
```

Kod producenta:

```
class Producer implements CSProcess {
    private final One2OneChannelInt[] out;
    private final One2OneChannelInt[] jeszcze;
    private final int N;

    public Producer(One2OneChannelInt[] out, One2OneChannelInt[] jeszcze, int N) {
        this.out = out;
        this.jeszcze = jeszcze;
        this.N = N;
    }

    public void run() {
        var guards = Arrays.stream(jeszcze).map(c -> c.in()).toArray(Guard[]::new);
        var alternative = new Alternative(guards);
        for (int i = 0; i < N; i++) {
            var index = alternative.select();
            jeszcze[index].in().read();
            var item = (int) (Math.random() * 100) + 1;
            out[index].out().write(item);
        }
    }
}
```

Kod konsumenta:

```
class Consumer implements CSProcess {
    private final One2OneChannelInt[] in;
```

```
    private final int N;

    public Consumer(final One2OneChannelInt[] in, int n) {
        this.in = in;
        this.N = n;
    }

    public void run() {
        var start = System.currentTimeMillis();

        var guards = Arrays.stream(in).map(c -> c.in()).toArray(Guard[]::new);
        var alt = new Alternative(guards);
        for (int i = 0; i < N; i++) {
            int index = alt.select();
            int item = in[index].in().read();
        }

        var end = System.currentTimeMillis();
        System.out.printf("Time elapsed: %d ms\n", end - start);
        System.exit(0);
    }
}
```

Aby uruchomić program:

- Określamy liczbę elementów do wyprodukowania na 15000.
- Określamy długość bufora, czyli ilość komórek na 15.
- Tworzymy odpowiednie kanały, które następnie przekazujemy do klas procesów.
- Wszystkie procesy uruchamiamy równolegle.

```
public class CSP {
    public static void main(String[] args) {
        int bufferLength = 15;
        int itemsToProduce = 15000;

        var channelIntFactory = new StandardChannelIntFactory();
        var producerChannels = channelIntFactory.createOne2One(bufferLength);
        var consumerChannels = channelIntFactory.createOne2One(bufferLength);
        var jeszczeChannels = channelIntFactory.createOne2One(bufferLength);

        var procList = new CSProcess[bufferLength + 2];
        procList[0] = new Producer(producerChannels, jeszczeChannels, itemsToProduce);
        procList[1] = new Consumer(consumerChannels, itemsToProduce);
        for (int i = 0; i < bufferLength; i++) {
            procList[i + 2] = new Buffer(producerChannels[i], consumerChannels[i],
            ↪  jeszczeChannels[i]);
        }

        new Parallel(procList).run();
    }
}
```

Wynik:

```
Time elapsed: 578 ms
```

Wariant z uwzględnieniem kolejności

- Ten wariant implementuje przetwarzanie potokowe w ścisłej kolejności. Każda komórka bufora jest połączona w łańcuch, gdzie:
  - Każda komórka przechowuje referencję do następnej w kolejności.
  - Ostatnia komórka wskazuje na konsumenta.
- Mechanizm działania:
  - Producent umieszcza dane tylko w pierwszej komórce łańcucha.
  - Pierwsza komórka przekazuje dane do drugiej (jeśli ta jest wolna).
  - Druga do trzeciej itd.
  - Ostatnia komórka przekazuje dane konsumenta.

Kod komórki bufora:

```
class Buffer implements CSProcess {
    private final One2OneChannelInt in;
    private final One2OneChannelInt out;

    public Buffer(One2OneChannelInt in,
                  One2OneChannelInt out) {
        this.out = out;
        this.in = in;
    }

    public void run() {
        while (true) {
            out.out().write(in.in().read());
        }
    }
}
```

Kod producenta:

```
class Producer implements CSProcess {
    private final One2OneChannelInt out;
    private final int N;

    public Producer(One2OneChannelInt out, int n) {
        this.out = out;
        this.N = n;
    }

    public void run() {
        for (int i = 0; i < N; i++) {
            var item = (int) (Math.random() * 100) + 1;
            out.out().write(item);
        }
    }
}
```

Kod konsumenta:

```
class Consumer implements CSProcess {
    private final One2OneChannelInt in;
    private final int N;

    public Consumer(final One2OneChannelInt in, int n) {
```

```
        this.in = in;
        this.N = n;
    }

    public void run() {
        var start = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            int item = in.in().read();
        }

        var end = System.currentTimeMillis();
        System.out.printf("Time elapsed: %d ms%n", end - start);
        System.exit(0);
    }
}
```

Kod uruchamiający, analogiczny do poprzedniej wersji:

```
public class CSPSeq {

    public static void main(String[] args) {
        int bufferLength = 15;
        int itemsToProduce = 15000;

        var channelIntFactory = new StandardChannelIntFactory();
        var channels = channelIntFactory.createOne2One(bufferLength + 1);

        var procList = new CSProcess[bufferLength + 2];
        procList[0] = new Producer(channels[0], itemsToProduce);
        procList[1] = new Consumer(channels[bufferLength], itemsToProduce);
        for (int i = 0; i < bufferLength; i++) {
            procList[i + 2] = new Buffer(channels[i], channels[i + 1]);
        }

        new Parallel(procList).run();
    }
}
```

Wynik:

```
Time elapsed: 626 ms
```

Wnioski

- Wersja problemu producenta i konsumenta, która zachowuje kolejność przetwarzania, jest nieco wolniejsza od wersji bez tej gwarancji.
- Biblioteka JCSP umożliwia nam implementowanie programów współbieżnych w stylu CSP. Język Java natywnie słabo wspiera modelowanie współbieżności w tym stylu. Dzięki zastosowaniu biblioteki nie musimy ręcznie zarządzać wątkami lub blokadami.

Bibliografia

- Communicating Sequential Processes for JavaTM (JCSP)