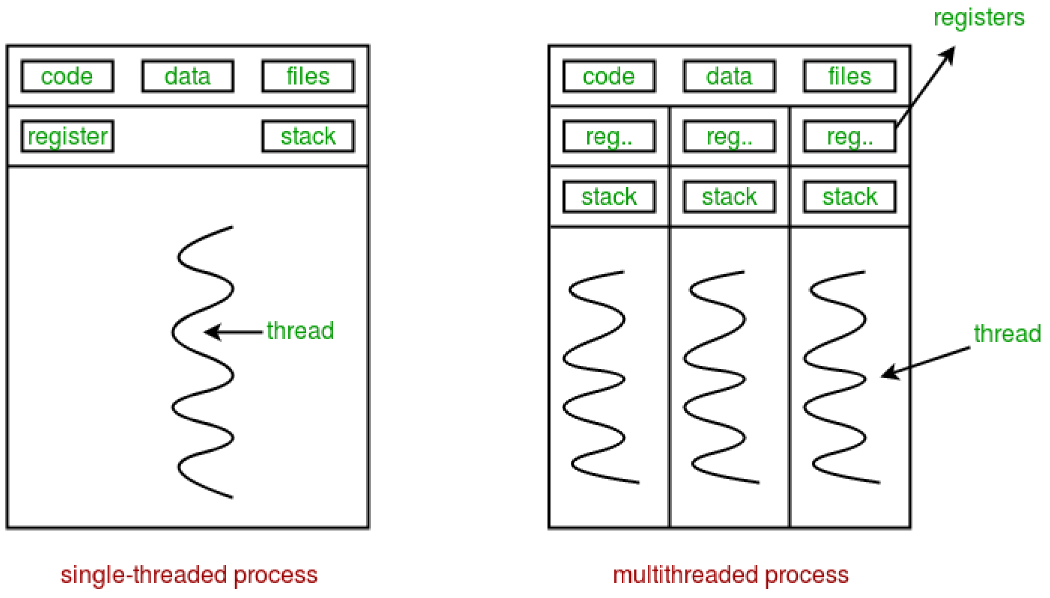
Producer Consumer



1. **Introduction**
2. **Version 1**
3. **Version 2**
4. **Version 3**
5. **Version 4**
6. **Information sources**

1. Introduction

The producer-consumer problem is a classic example where it is necessary to give an independent treatment to a set of data that is generated in a more or less random way or at least in a way in which it is not possible to predict at what moment a data will be generated. To avoid excessive use of computer resources while waiting for data to arrive, the system foresees two types of processes: the producers, in charge of obtaining the data to be processed, and the consumers, specialized in processing the data obtained by the producers.

Look at the solved example of the problem of **producers-consumers**

2. Version 1

We are going to develop the **Producer-Consumer** problem, let's first see what the problem is: we will create two types of threads: a `Producer` that will put some data (for example, an integer) into a given object (we will call it `SharedData`), and a `Consumer` that will get this data. Our `SharedData.java` class is this:

```java
public class SharedData {

    int data;

    public int get() {
        return data;
    }

    public void put(int newData) {
        data = newData;
    }
}
```

and the classes `Producer.java`:

```java
public class Producer extends Thread {

    SharedData data;

    public Producer(SharedData data) {
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            data.put(i);
            System.out.println("Produced number " + i);
            try {
                Thread.sleep(10);
            } catch (Exception e) {
            }
        }
    }
}
```

and `Consumer.java`:

```java
public class Consumer extends Thread {

    SharedData data;

    public Consumer(SharedData data) {
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            int n = data.get();
            System.out.println("Consumed number " + n);
            try {
                Thread.sleep(10);
            } catch (Exception e) {
            }
        }
    }
}
```

The main `Test.java` application will create a `SharedData` object and a thread of each type, and start both.

```java
public class Test {
    public static void main(String[] args) {
        SharedData sd = new SharedData();
        Producer p = new Producer(sd);
        Consumer c = new Consumer(sd);
        p.start();
        c.start();
    }
}
```

In the result we can observe some problems:

```
run:
Consumed number 0
Produced number 0
Consumed number 0
Produced number 1
Consumed number 1
Produced number 2
Produced number 3
Consumed number 3
Produced number 4
```

3. Version 2

We might think that if we just added the `synchronized` keyword to the `get` and `put` methods of the `SharedData` class, it should solve the problem:

```java
public class SharedData {
    int data;
    public synchronized int get() {
        return data;
    }
    public synchronized void put(int newData) {
        data = newData;
    }
}
```

However, if we run the program again, we can notice that it still fails:

```
run:
Consumed number 0
Produced number 0
Consumed number 0
Produced number 1
Produced number 2
Consumed number 1
Produced number 3
Consumed number 3
Produced number 4
```

4. Version 3

In fact, there are two problems that we have to solve. But let's start with the most important: the producer and the consumer have to work in coordination: as soon as the producer puts a number, the consumer can get it, and the producer cannot produce more numbers until the consumer gets the previous ones.

To do this, we need to add some changes to our `SharedData` class. First of all, we need a flag that tells producers and consumers who is next. It will depend on whether there is new data to consume (consumer's turn) or not (producer's turn).

```java
public class SharedData {

    int data;
    boolean available = false;

    public synchronized int get() {
        available = false;
        return data;
    }

    public synchronized void put(int newData) {
        data = newData;
        available = true;
    }
}
```

5. Version 4

Also, we need to make sure that the `get` and `put` methods are called alternately. To do this, we need to use the boolean flag and the `wait` and `notify`/`notifyAll` methods, like so:

```java
public class SharedData {

    int data;
    boolean available = false;

    public synchronized int get() {
        if (!available) {
            try {
                wait();
            } catch (Exception e) {
            }
        }
        available = false;
        notify();
        return data;
    }

    public synchronized void put(int newData) {
        if (available) {
            try {
                wait();
            } catch (Exception e) {
            }
        }
        data = newData;
        available = true;
        notify();
    }
}
```

Notice how we use the `wait` and `notify` methods. Regarding the `get` method (called by the `Consumer`), if nothing is available, we wait. Then we get the number, set the flag to false again, and notify the other thread.

In the `put` method (called by the `Producer`), if something is available, we wait until someone notifies us. Then we `set` the new data, set the flag to `true` again, and notify the other thread.

If both threads try to reach the critical section at the same time, the `Consumer` will have to wait (the flag is set to `false` at the beginning), and the `Producer` will set the first data to be consumed. From then on, the threads will alternate in the critical section, consuming and producing new data each time.

6. Information sources

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
  - https://github.com/ajcpro/psp
  - https://oscarmaestre.github.io/servicios/index.html
  - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
  - https://github.com/pablohs1986/dam_psp2021
  - https://github.com/Perju/DAM
  - https://github.com/eldiegoch/DAM
  - https://github.com/eldiegoch/2dam-psp-public
  - https://github.com/franlu/DAM-PSP
  - https://github.com/ProgProcesosYServicios
  - https://github.com/joseluisgs
  - https://github.com/oscarnovillo/dam2_2122
  - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos