Process Synchronization

Problemas:

1. Pode acontecer de o processo Produtor produzir novos valores mesmo quando a caixa já está cheia, sobrescrevendo o que já está lá.
2. Pode acontecer de os processos consumidores tentarem consumir o que está na caixa mesmo ela estando vazia.
3. Não está sendo que garantido que um **Consumidor Par** consuma apenas **números pares** e nem que um **Consumidor Ímpar** consuma apenas **números ímpares**.

Solução:

O que precisamos fazer primeiro é sincronizar os processos:

1. Garantir que o produtor produza apenas um valor por vez até algum processo consumí-lo.
2. Garantir que sempre haja algum valor na caixar antes que algum processo tente consumir.

Para isso criamos duas variáveis no arquivo “Dropbox.java”:

```
public int p = 0;
public int c = 0;
```

- Sempre que o processo produtor produzir um valor, a variável *p* será incrementada.
- Sempre que um processo consumidor consumir um valor, a variável *c* será incrementada.

A idéia é que um processo consumidor só vai tentar consumir quando o valor de *p* for maior que *c*

(p > c). Além disso, como queremos que o produtor não produza mais que um valor por vez, ele só será capaz de produzir quando um processo já tiver consumido o que ele produziu, ou seja, quando a variável *p* for igual a *c (*p == c).

```
while (true) {
            // < await (p == c); >
            while (dropbox.p > dropbox.c){
                continue;
            }

            int number = random.nextInt(10);

            try {
                Thread.sleep(random.nextInt(100));
                dropbox.put(number);
                dropbox.p += 1;
            } catch (InterruptedException e) { }
        }
```

O código acima está no arquivo “Producer.java”.

Neste código, usamos o loop para atrasar o produtor até que o valor de p seja igual ao valor de c.

Após isso, o processo produz o valor, armazena-o na caixa e incrementa a variável p.

```
while (true) {

            // < await (p > c);
            while (dropbox.p == dropbox.c){
                continue;
            }

            dropbox.take(even);
            dropbox.c += 1;

            try {
               Thread.sleep(random.nextInt(100));
```

```
            } catch (InterruptedException e) { }
        }
```

O código acima está no arquivo “Consumer.java”.

Neste código, o loop está atrasando o processo até que o valor de p seja maior que c, ou seja, está esperando que tenha algo na caixa. Após a condição ser satisfeita, ele consome o que está na caixa e incrementa a variável c. Assim, o produtor está liberado para poder produzir outro valor, já que *p* será igual a *c* novamente.

No entanto, um problema ainda não foi resolvido. Ainda não está sendo garantido que um **Consumidor Par** consuma apenas valores **pares** e que um **Consumidor Ímpar** consuma apenas valores **ímpares.** Para resolver isso, basta verificar se o número tem a mesma paridade do consumidor antes que ele tente consumir. Abaixo segue a versão final com a verificação de paridade.

```
while (true) {

            // < await (p > c);
            while (dropbox.p == dropbox.c){
                continue;
            }

            // verificação de paridade
            if (dropbox.getEvenNumber() == this.even){
                dropbox.take(even);
                dropbox.c += 1;
            }

            try {
               Thread.sleep(random.nextInt(100));
            } catch (InterruptedException e) { }
        }
```