

# MAC0438 – PROGRAMAÇÃO CONCORRENTE

## Jantar dos Selvagens

Daniel Augusto Cortez – 2960291

### 1 Introdução

Este relatório descreve a implementação proposta para resolver o problema do jantar dos selvagens, bem como apresenta os resultados dos experimentos sugeridos para análise das soluções obtidas.

A localização do código-fonte com as funções exigidas para manipulação do monitor e a forma de utilização do programa estão disponíveis no arquivo `README`.

### 2 Implementação

A implementação foi escrita em Java (1.7.45) no ambiente de desenvolvimento Eclipse (Kepler). O arquivo `DiningSavages.jar` deve ser utilizado para executar o programa.

O monitor implementado na classe `PotMonitor.java` garante o acesso exclusivo ao pote pelos selvagens e pelos cozinheiros. Em alto nível, sem se preocupar com detalhes como o número de vezes que o pote é esvaziado (repetições) e as informações a serem impressas, o código do monitor pode ser escrito como

```
monitor Pot {
    int portions = C;
    cond potEmpty, potFull;

    procedure eatPortion(Savage savage) {
        while (portions == 0) {
            # O primeiro faminto que percebe o pote vazio acorda um cozinheiro
            if (empty(potFull)) signal(potEmpty);
            wait(potFull); # wait(potFull, rank(savage))
        }
        portions = portions - 1;
    }

    procedure makePortions(Cook cook) {
        while (portions > 0) wait(potEmpty);
        portions = C;
        signal_all(potFull);
    }
}
```

Foram utilizadas duas variáveis de condição: `potEmpty` para indicar quando o pote está vazio, e `potFull` para indicar quando o pote está cheio. O contador `portions` conta quantas porções estão

disponíveis no pote. Se `portions = 0` o pote está vazio. Se `portions = C` o pote está cheio. A constante  $C$  é a capacidade do pote informada no arquivo de entrada. Note que apenas o primeiro faminto que observa o pote vazio ao tentar comer dele é o responsável por acordar um cozinheiro. Note também que se um cozinheiro tiver acesso ao monitor e verificar que o pote está vazio, ele completará o pote com novas porções.

O `wait()` com prioridade foi utilizado no caso em que a simulação roda com pesos. Nesse caso, selvagens mais gordos são colocados primeiro na fila da variável `potFull`. Quando essa condição é verdadeira, eles acessam primeiro o pote, e portanto, podem comer mais.

Pela necessidade de se implementar um monitor com prioridades, não se utilizou o métodos padrões `wait()` e `notify()` do Java. A implementação do monitor foi feita com base em semáforos, conforme exposto nos slides da Aula 18. Um *lock* é utilizado dentro do monitor para garantir o acesso exclusivo aos métodos. Cada variável de condição possui um vetor privado de semáforos e uma fila de threads. Cada thread na fila aguarda em um semáforo específico do vetor. A implementação da variável de condição é feita na classe `ConditionVariable.java`. Note que na implementação utilizou-se uma lista ligada para a fila de threads e um `HashMap` para o vetor de semáforos. Quando se deseja colocar uma thread na fila com prioridade, essa thread deve implementar a interface `Rankable`. Com isso, pode-se extrair o valor do *rank* associado à thread e inseri-la em ordem na fila.

O selvagem é implementado na classe `Savage.java`. Ela estende a classe `Thread`. O seu método `run()` chama o procedimento `eatPortion()` do monitor enquanto o número de repetições do pote não se esgotar.

O cozinheiro é implementado na classe `Cook.java`. Ela estende a classe `Thread`. O seu método `run()` chama o procedimento `makePortions()` do monitor enquanto o número de repetições do pote não se esgotar.

A classe `DinigSavages.java` contém o método `main()` do programa. Ele efetua a leitura dos parâmetros e faz a instanciação dos objetos de acordo (selvagens, cozinheiros e o monitor). Em seguida, inicializa as threads rodando a simulação.

O número de repetições, que equivale ao número de vezes que o pote é esvaziado, é controlado pelo monitor. Esse número é lido da linha de comando e passado ao monitor em sua instancionalização. Toda vez que o pote é esvaziado por algum selvagem, o número de repetições é decrementado. Quando esse contador atinge o valor zero, todas as threads são sinalizadas. Ao acordar, a primeira coisa que uma thread faz é verificar se o número de repetições é positivo. Caso negativo, o *lock* é liberado e ela imediatamente sai do procedimento. Os procedimentos dos selvagens e cozinheiros são iterados enquanto o número de repetições for positivo.

### 3 Experimentos

Algumas simulações foram realizadas para testar o correto funcionamento do programa. Elas foram executadas em uma máquina com processador Intel Core i5 (1.6 GHz) com 4Gb de memória RAM, rodando o sistema operacional MacOS 10.9.2.

Foram utilizados dois arquivos de entrada: `simples.txt` e `complexo.txt`. O primeiro arquivo representa um cenário simples com 2 selvagens e 2 cozinheiros. O segundo arquivo representa um cenário complexo com 20 selvagens e 10 cozinheiros. Os números foram escolhidos de forma arbitrária, limitando-se o número de threads ao número de núcleos de CPU no cenário simples. A capacidade do pote utilizada no caso simples foi  $C = 3$  e no caso complexo  $C = 12$ . Cada cenário foi avaliado nas situações uniforme e com pesos através de 1000 repetições. Os dados resultantes das simulações estão disponíveis no arquivo `experimentos.xlsx`.

A Figura 3.1 apresenta os resultados da simulação do cenário simples no caso uniforme. Do lado esquerdo apresenta-se um gráfico (Gráfico 1 do enunciado do EP) em que o eixo horizontal representa o tempo lógico da simulação (número de vezes que o pote foi enchido), e o eixo vertical representa o número médio de vezes que cada cozinheiro acordou. Do lado direito apresenta-se um gráfico (Gráfico 2 do enunciado do EP) em que o eixo horizontal representa os selvagens, e o eixo vertical representa o número de porções que cada selvagem comeu.

A Figura 3.2 apresenta os resultados da simulação do cenário simples no caso com pesos. A descrição dos gráficos apresentados é a mesma que no caso anterior.

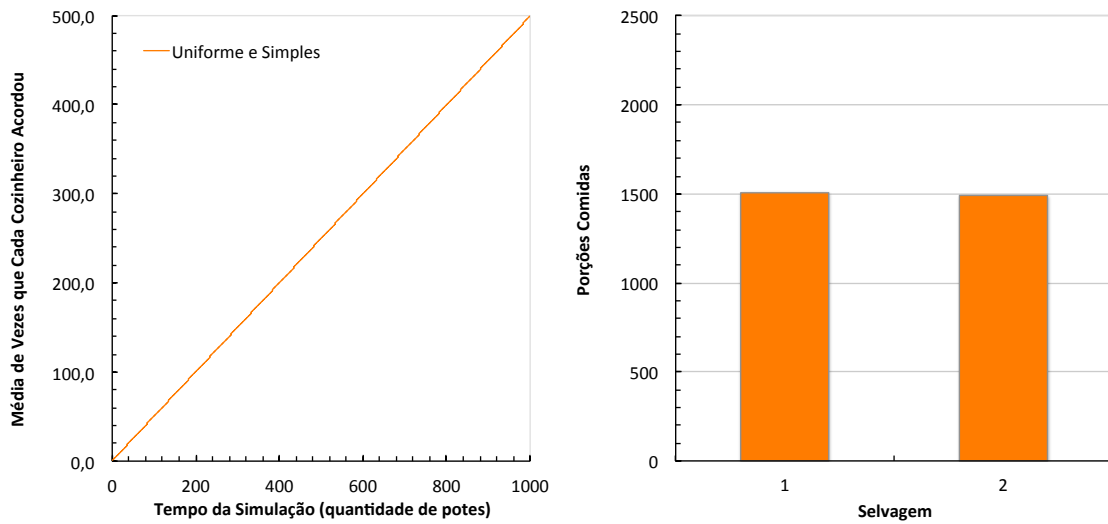


Figura 3.1: Gráficos 1 e 2 para o cenário simples no caso uniforme.

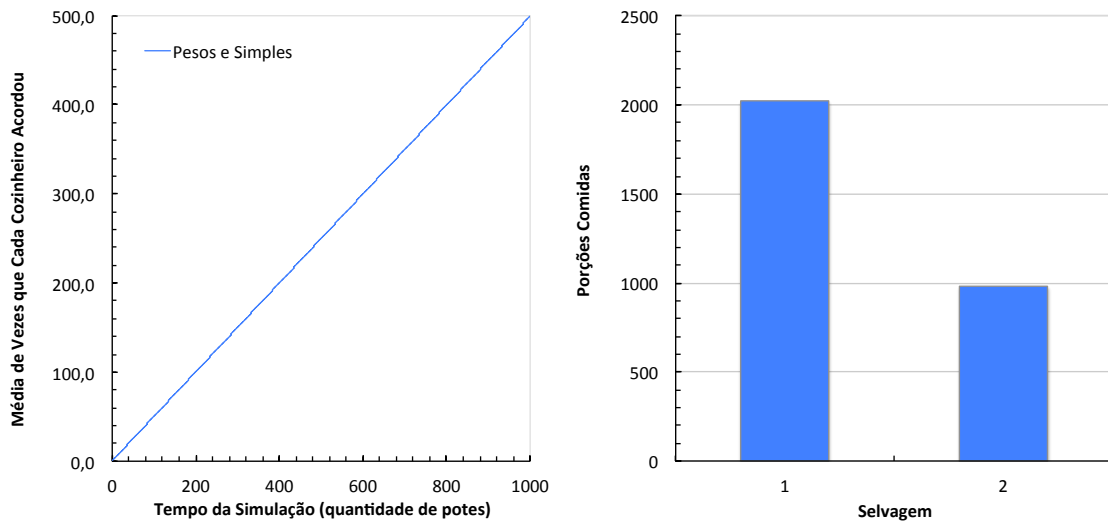


Figura 3.2: Gráficos 1 e 2 para o cenário simples no caso com pesos. Os pesos atribuídos aos selvagens foram 2, 1.

As Figura 3.3 e 3.4 apresentam os resultados da simulação do cenário complexo nos casos uniforme e com pesos, respectivamente. A descrição dos gráficos apresentados é a mesma que nos casos anteriores.

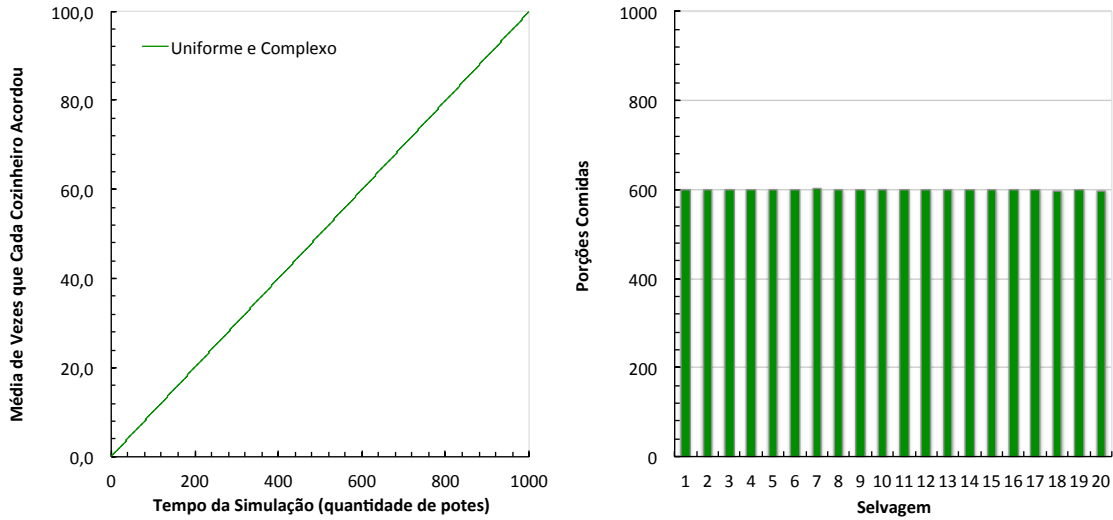


Figura 3.3: Gráficos 1 e 2 para o cenário complexo no caso uniforme.

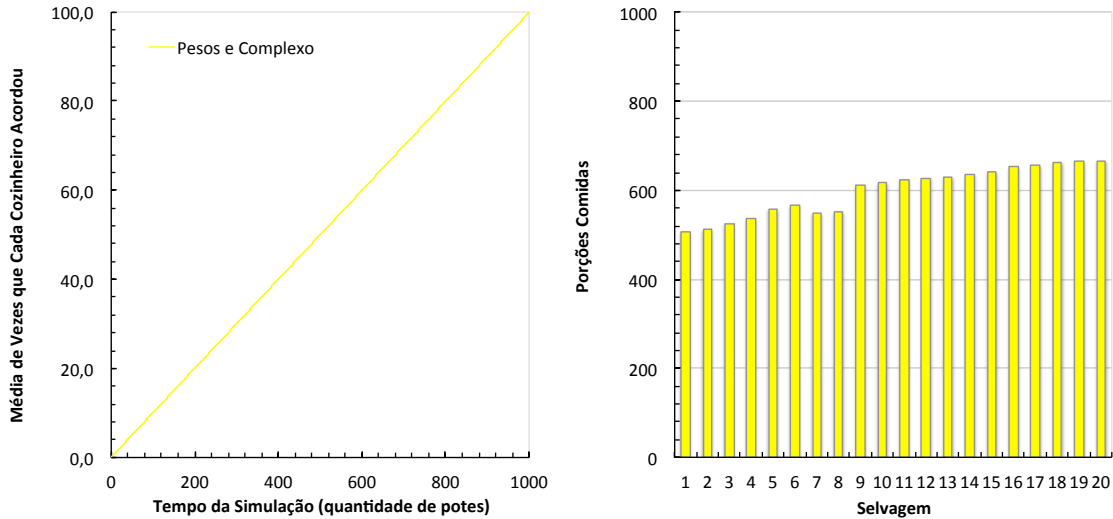


Figura 3.4: Gráficos 1 e 2 para o cenário complexo no caso com pesos. Os pesos atribuídos aos selvagens foram 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20.

## 4 Conclusões

O comportamento linear do gráfico 1 encontrado em todos os casos foi o esperado, indicando que os mecanismos de sincronização funcionaram corretamente. De fato, se a quantidade de potes é  $Q$ , então a soma de vezes que cada cozinheiro encheu o pote é  $Q$ , de forma que a média do número de vezes que cada cozinheiro acordou é  $Q/M$  ( $M$  é o número de cozinheiros). Isso explica as retas obtidas com coeficiente angular  $1/2$  no cenário simples (Figuras 3.1 e 3.2,  $M = 2$ ), e as retas obtidas com coeficiente angular  $1/10$  no cenário complexo (Figuras 3.3 e 3.4,  $M = 10$ ).

O comportamento do gráfico 2 em todos os casos também foi o esperado. Observe como nos casos uniforme (Figuras 3.1 e 3.3) a quantidade de vezes que cada selvagem comeu é praticamente a mesma, uma vez que as barras verticais são quase do mesmo tamanho. Já no caso com pesos (Figuras 3.2 e 3.4), as barras ficaram mais ou menos proporcionais aos pesos dos selvagens, indi-

cando que selvagens mais gordos comeram mais do que selvagens mais magros. Isso indica que o mecanismo de inserir os selvagens na fila da variável de condição `potFull` com prioridade funcionou corretamente.

Nas simulações escolhemos valores de  $C$  (a capacidade do pote) que melhor ilustrassem os comportamentos esperados. Entretanto, observou-se que o comportamento é relativamente sensível a escolha desse parâmetro, sem entretanto, afetar o seu caráter qualitativo esperado.

Para finalizar, gostaríamos de destacar que a saída do programa está de acordo com a lógica implementada. Em particular, observa-se que sempre primeiro um selvagem nota o pote vazio, depois é feita a impressão dos totais solicitados no enunciado, e por fim o cozinheiro acordado enche o pote. A saída típica é como no trecho abaixo, extraído durante a simulação do cenário simples no caso uniforme:

```
(Selvagem S2 notou o pote vazio)
Selvagem S1 comeu 1484 vezes
Selvagem S2 comeu 1486 vezes
Cozinheiro C1 encheu 496 vezes
Cozinheiro C2 encheu 494 vezes
[Cozinheiro C2 encheu o pote]
(Selvagem S1 notou o pote vazio)
Selvagem S1 comeu 1485 vezes
Selvagem S2 comeu 1488 vezes
Cozinheiro C1 encheu 496 vezes
Cozinheiro C2 encheu 495 vezes
[Cozinheiro C1 encheu o pote]
(Selvagem S2 notou o pote vazio)
Selvagem S1 comeu 1487 vezes
Selvagem S2 comeu 1489 vezes
Cozinheiro C1 encheu 497 vezes
Cozinheiro C2 encheu 495 vezes
[Cozinheiro C2 encheu o pote]
```

A situação em que são impressas duas (ou mais) informações de que um cozinheiro encheu o pote, sem a impressão da informação de que um selvagem notou o pote vazio também pode ocorrer. Por exemplo:

```
[Cozinheiro C2 encheu o pote]
[Cozinheiro C1 encheu o pote]
(Selvagem S1 notou o pote vazio)
Selvagem S1 comeu 1485 vezes
Selvagem S2 comeu 1488 vezes
Cozinheiro C1 encheu 496 vezes
Cozinheiro C2 encheu 495 vezes
```

Essa situação ocorre quando um cozinheiro é escalonado e tem acesso ao monitor justamente quando o pote está vazio. Nesse caso ele não precisa de nenhum selvagem para acordá-lo, e ele entra no monitor preparando novas porções.