MC504 – PROJETO 1

FILÓSOFOS FAMINTOS: ALGORITIMOS E PARALELISMO

DANIEL SCOCCO

Metodologia

```
while(true)
    think()
    getForks()
    eat()
    putForks()
```

Diferentes algoritimos

Metodologia

- CPU: Intel i3 4-Cores @ 2.93GHz
- OS: Ubuntu 12.04 32-bits
- Sleep Interval (Think e Eat): 100 a 1000 ms
- Cada algoritimo testado 10 vezes até 1000 refeições, com 9 filósofos na mesa.
- Calculada a média de refeições/minuto e de filósofos comendo ao mesmo tempo

Algoritimo 1- BusyWait 1

```
getForks1(){
    while(forksArray[rightFork]==0);
    forksArray[rightFork] = 0;

    while(forksArray[leftFork]==0);
    forksArray[leftFork] = 0;
}
```

Problemas:

- Não garante exclusão mutua (raro)
- Não garante ausência de deadlock (freq.)
- Não garante ausência de starvation (raro)

Algoritimo 1 – BusyWait 1

Trial 1:	Meals/sec = 3.32	Concurrency = 1.80
Trial 2:	Meals/sec = 4.48	Concurrency = 2.43
Trial 3:	Meals/sec = 4.43	Concurrency = 2.41
Trial 4:	Meals/sec = 4.06	Concurrency = 2.40
Trial 5:	Meals/sec = 4.69	Concurrency = 2.57
Trial 6:	Meals/sec = 3.75	Concurrency = 2.11
Trial 7:	Meals/sec = 3.95	Concurrency = 2.31
Trial 8:	Meals/sec = 4.52	Concurrency = 2.39
Trial 9:	Meals/sec = 3.98	Concurrency = 2.14
Trial 10:	Meals/sec = 4.03	Concurrency = 2.17
Average:	**Meals/sec = 4.12**	**Concurrency = 2.07**

Algoritimo 2 – BusyWait 2

```
getForks2(){
    if(id%2==0){
        while(forksArray[rightFork]==0);
        forksArray[rightFork] = 0;
        while(forksArray[leftFork]==0);
        forksArray[leftFork] = 0;
    }
    else{
        while(forksArray[leftFork]==0);
        forksArray[leftFork] = 0;
        while(forksArray[rightFork]==0);
        forksArray[rightFork] = 0;
    }
}
```

Resolve problema de deadlock, aumenta eficiência, porém continua não garantindo exclusão mutua e ausência de deadlock

Algoritimo 1 – BusyWait 2

Trial 1:	Meals/sec = 5.92	Concurrency = 3.19
Trial 2:	Meals/sec = 5.91	Concurrency = 3.24
Trial 3:	Meals/sec = 5.90	Concurrency = 3.17
Trial 4:	Meals/sec = 5.96	Concurrency = 3.23
Trial 5:	Meals/sec = 5.89	Concurrency = 3.26
Trial 6:	Meals/sec = 5.99	Concurrency = 3.25
Trial 7:	Meals/sec = 5.81	Concurrency = 3.22
Trial 8:	Meals/sec = 5.84	Concurrency = 3.21
Trial 9:	Meals/sec = 5.82	Concurrency = 3.17
Trial 10:	Meals/sec = 5.77	Concurrency = 3.26
Average:	**Meals/sec = 5.88**	**Concurrency = 3.22**

Algoritimo 3 – Semaphores 1

```
getForks3(){
    if(id%2==0){
        forksSem[rightFork].acquire();
        forksSem[leftFork].acquire();
    }
    else{
        forksSem[leftFork].acquire();
        forksSem[rightFork].acquire();
    }
}
```

Garante exclusão mutua, ausência de deadlock e de starvation (Java Semaphores tem um atributo *fairness*).

Algoritimo 3 – Semaphores 1

Trial 1:	Meals/sec = 5.96	Concurrency = 3.23
Trial 2:	Meals/sec = 5.88	Concurrency = 3.23
Trial 3:	Meals/sec = 5.77	Concurrency = 3.28
Trial 4:	Meals/sec = 5.89	Concurrency = 3.23
Trial 5:	Meals/sec = 5.99	Concurrency = 3.30
Trial 6:	Meals/sec = 5.90	Concurrency = 3.20
Trial 7:	Meals/sec = 5.90	Concurrency = 3.20
Trial 8:	Meals/sec = 5.91	Concurrency = 3.27
Trial 9:	Meals/sec = 5.93	Concurrency = 3.25
Trial 10:	Meals/sec = 5.89	Concurrency = 3.27
Average:	**Meals/sec = 5.90**	**Concurrency = 3.25**

Algoritimo 4 – Semaphores 2

```
getForks4(){
    placesSem.acquire();
    forksSem[rightFork].acquire();
    forksSem[leftFork].acquire();
}
```

Solução do livro: semáforo extra para controlar quantos filósofos podem comer ao mesmo tempo. Funciona, mas perde eficiência.

Algoritimo 4 – Semaphores 2

Trial 1:	Meals/sec = 4.23	Concurrency = 2.36
Trial 2:	Meals/sec = 3.62	Concurrency = 2.00
Trial 3:	Meals/sec = 4.72	Concurrency = 2.58
Trial 4:	Meals/sec = 4.01	Concurrency = 2.14
Trial 5:	Meals/sec = 3.58	Concurrency = 1.97
Trial 6:	Meals/sec = 3.25	Concurrency = 1.81
Trial 7:	Meals/sec = 4.78	Concurrency = 2.58
Trial 8:	Meals/sec = 4.83	Concurrency = 2.64
Trial 9:	Meals/sec = 4.30	Concurrency = 2.33
Trial 10:	Meals/sec = 4.05	Concurrency = 2.28
Average:	**Meals/sec = 4.18**	**Concurrency = 2.27**

Algoritimo 5 – BusyWait 3

```
getForks5(){
    while(philosophersBlock[id]!=0);
    philosophersBlock[leftNeighbor]++;
    philosophersBlock[rightNeighbor]++;
}
```

Filósofo só pode pegar garfos se ambos estiverem disponíveis. Sem deadlock e bem eficiente. Problema de exclusão mutua (que pode ser resolvido com semáforos nos garfos, sacrificando um pouco da eficiência).

Algoritimo 5 – BusyWait 3

Trial 1:	Meals/sec = 6.09	Concurrency = 3.37
Trial 2:	Meals/sec = 6.23	Concurrency = 3.37
Trial 3:	Meals/sec = 6.06	Concurrency = 3.37
Trial 4:	Meals/sec = 6.18	Concurrency = 3.47
Trial 5:	Meals/sec = 6.03	Concurrency = 3.35
Trial 6:	Meals/sec = 6.22	Concurrency = 3.40
Trial 7:	Meals/sec = 6.17	Concurrency = 3.51
Trial 8:	Meals/sec = 6.30	Concurrency = 3.40
Trial 9:	Meals/sec = 6.24	Concurrency = 3.45
Trial 10:	Meals/sec = 6.08	Concurrency = 3.32
Average:	**Meals/sec = 6.16**	**Concurrency = 3.37**

Comparação

Algoritimo BusyWait 1

Meals/sec = 4.12   Concurrency = 2.07

Algoritimo BusyWait 2

Meals/sec = 5.88   Concurrency = 3.22

Algoritimo Semaphores 1

Meals/sec = 5.90 Concurrency = 3.25

Algoritimo Semaphores 2

Average: Meals/sec = 4.18   Concurrency = 2.27

Algoritimo BusyWait 3

Meals/sec = 6.16   Concurrency = 3.37

Pra Se Pensar

1. Nem sempre nível de paralelismo andou junto ao desempenho geral do algoritimo (e.g., algoritimo 5 tem 10% mais paralelismo que algoritimo 1, mas somente 1% mais desempenho). Qual a melhor forma de se medir?

2. Mesmo algoritimo implementado com BusyWait e com semáforos teve quase o mesmo desempenho. Ciclos desperdiçados do BusyWait equivalem ao overhead dos semáforos? Resultado da função eat() somente dormir ao invez de realizar algo?