Jantar dos filósofos - Sistemas Operacionais

João Vitor Yukio Bordin Yamashita

October 10, 2022

1 Jantar dos filosófos

A explicação do problema foi retirada do livro de referência do curso [TB14].

Em 1965, Dijkstra formulou e então solucionou um problema de sincronização que ele chamou de problema do jantar dos filósofos. Desde então, todos os que inventaram mais uma primitiva de sincronização sentiram-se obrigados a demonstrar quão maravilhosa é a nova primitiva exibindo quão elegantemente ela soluciona o problema do jantar dos filósofos. O problema pode ser colocado de maneira bastante simples, como a seguir: cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que um filósofo precisa de dois garfos para comê-lo, podemos ver uma ilustração do problema na Figura 1.

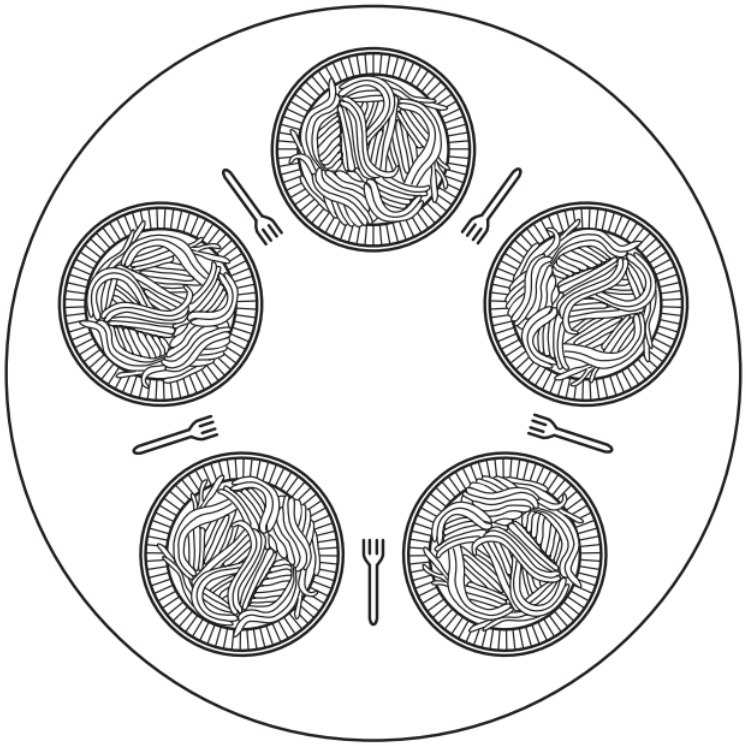


Figure 1: Mesa do problema proposto

A vida de um filósofo consiste em alternar períodos de alimentação e pensamento. (Trata-se de um tipo de abstração, mesmo para filósofos, mas as outras atividades são irrelevantes aqui.) Quando um filósofo fica suficientemente faminto, ele tenta pegar seus garfos à esquerda e à direita, um de cada vez, não importa a ordem. Se for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar.

2 Modelando o problema

Primeiramente vamos propor uma solução usando *mutex*[1] da biblioteca std da linguagem c++. Esse mutex vai representar nossos garfos, ou seja, para o problema proposto teremos cinco mutexes.

Para modelar os nossos filósofos vamos criar uma classe para encapsular o comportamento descrito no problema acima. A vida do filósofo vai ser modelada como uma função.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

using namespace std;

class Filosofo{
    private:
        int id;
        long vezesComeu;
        long vezesPensou;
    public:
        std::mutex* m1; // Garfo da esquerda
        std::mutex* m2;// Garfo da direita
        //Construtor do filosofo
        Filosofo(int id, std::mutex* garfoEsq, std::mutex* garfoDir)
        {
            this->id = id;
            this->m1 = garfoEsq;
            this->m2 = garfoDir;
            vezesComeu = 0;
            vezesPensou = 0;
        }
        void vidaFilosofo()
        {
            /*
                    Vida do filosofo:
                        Pensar,
                        // Depois de um tempo, quando ficar com fome
                        Comer -> pegar um garfo, e depois pegar outro
                        // Depois de satisfeito
                        Devolver os garfos e voltar a pensar
            */
            while(true)
            {
                pensa();
                pegarGarfos();
                come();
                soltaGarfos();
            }
        }
};
```

Listing 1: Modelagem inicial do filosofo

[1] https://cplusplus.com/reference/mutex/mutex/

Com isso podemos criar as funções utilizadas na vida do filósofo:

```
void pensa()
{
    cout << "Filosofo␣" << id << "␣pensando\n";
    vezesPensou++;
    //Tempo que demora para sentir fome
    this_thread::sleep_for(chrono::milliseconds(1000));
}
};
```

Listing 2: Função pensa

Inicialmente essa função foi feita considerando que cada filósofo demora um tempo fixo de 1 segundo pensando. De forma semelhante a função 'come' foi feita considerando que cada filósofo come por um tempo fixo de 1 segundo.

```
void come()
{
    cout << "Filosofo␣" << id << "␣comendo\n";
    vezesComeu++;
    //Tempo que leva para comer
    this_thread::sleep_for(chrono::milliseconds(1000));
}
};
```

Listing 3: Função come

A função 'pegarGarfos' usa o método *lock()*[2] para pegar o garfo caso esteja livre, caso ele não esteja a *thread* que no caso representa o filósofo fica travada esperando o garfo ser liberado.

```
void pegarGarfos()
{
    //Tenta pegar o garfo da direita
    this->m1->lock();
    //Tenta pegar o garfo da esquerda
    this->m2->lock();
}
```

Listing 4: Função pergarGarfos

E por fim, a função soltarGarfos chama o método *unlock()*[3] para soltar os garfos.

```
void pegarGarfos()
{
    //Tenta pegar o garfo da esquerda
    this->m1->unlock();
    //Tenta pegar o garfo da direita
    this->m2->unlock();
}
```

Listing 5: Função soltarGarfos

[2] https://cplusplus.com/reference/mutex/mutex/lock/
[3] https://cplusplus.com/reference/mutex/mutex/unlock/

Agora podemos criar nosso problema de fato, temos a seguinte *main*, onde criamos as *threads*[4] e usamos o método *join()*[5] para sincronizar as threads:

```
int main()
{
    //Garfos
    std::mutex garfo1;
    std::mutex garfo2;
    std::mutex garfo3;
    std::mutex garfo4;
    std::mutex garfo5;

    //Filosofos
    Filosofo f1 = Filosofo(1, &garfo5, &garfo1);
    Filosofo f2 = Filosofo(2, &garfo1, &garfo2);
    Filosofo f3 = Filosofo(3, &garfo2, &garfo3);
    Filosofo f4 = Filosofo(4, &garfo3, &garfo4);
    Filosofo f5 = Filosofo(5, &garfo4, &garfo5);

    //Criacao das threads
    std::thread threadsFilosofos[5] = {
        std::thread(&Filosofo::vidaFilosofo, &f1),
        std::thread(&Filosofo::vidaFilosofo, &f2),
        std::thread(&Filosofo::vidaFilosofo, &f3),
        std::thread(&Filosofo::vidaFilosofo, &f4),
        std::thread(&Filosofo::vidaFilosofo, &f5)
    };

    //Iniciacao das threads
    for(int i = 0; i<5; i++)
    {
        threadsFilosofos[i].join();
    }

    return 0;
}
```

Listing 6: Função main

Nesse caso, o problema de *starvation* não ocorrerá, já que quando um filósofo pega o garfo e não consegue pegar o segundo ele espera até que o segundo seja liberado com o primeiro garfo na mão. Um outro problema que, nesse caso, poderá acontecer é o *deadlock*, que ocorre caso todos os filósofos peguem os seus garfos da esquerda, o que irá acarretar numa espera infinita para que o seu vizinho devolva o garfo, como pode ser visto na figura 2.

[4] https://cplusplus.com/reference/thread/thread/
[5] https://cplusplus.com/reference/thread/thread/join/

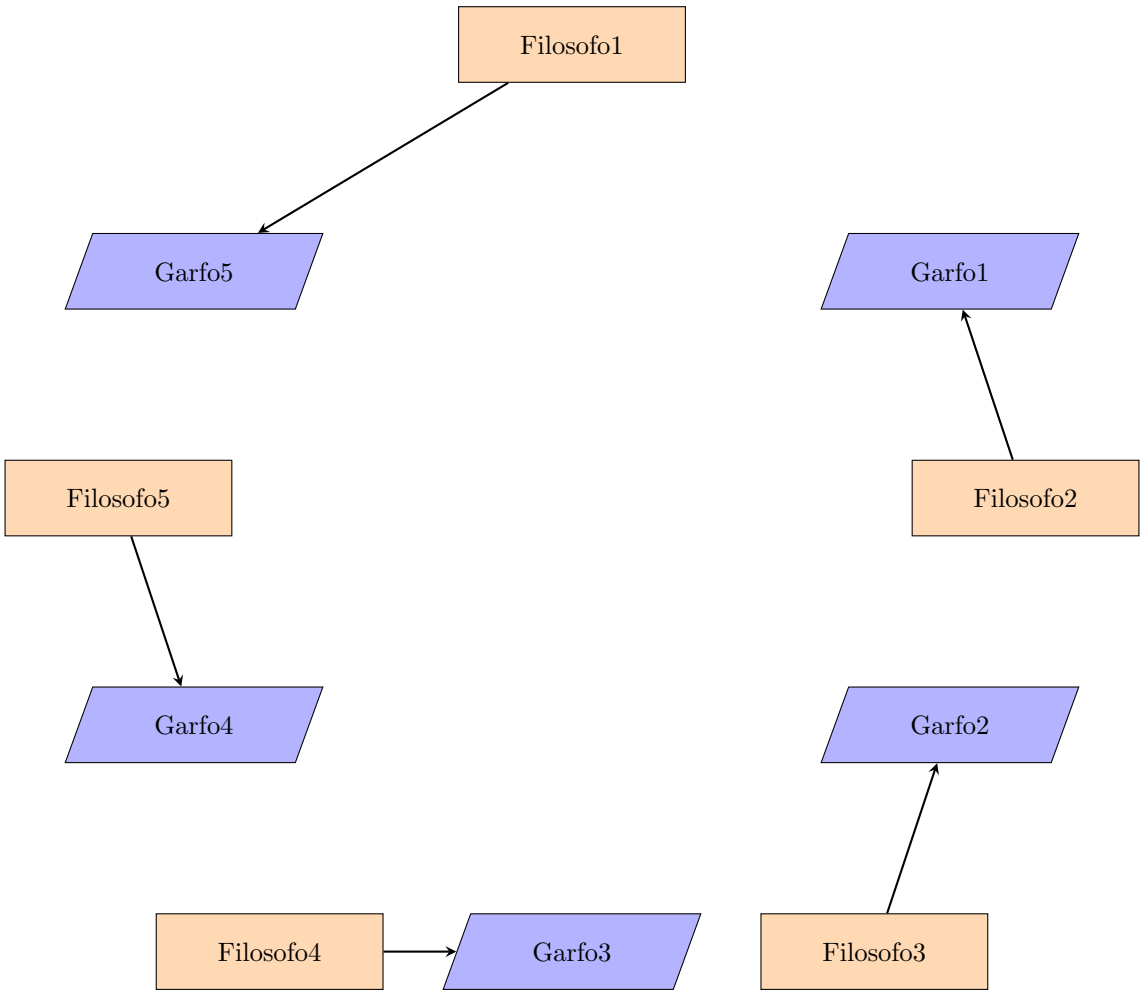


Figure 2: Deadlock

Para resolver isso vamos quebrar a circularidade do problema, fazendo com que um filósofo seja 'canhoto', ou seja, comece pegando o garfo da sua esquerda primeiro. Para isso basta trocarmos a ordem dos garfos de um filósofo, fazendo isso para o último filósofo temos:

```
Filosofo f1 = Filosofo(1, &garfo5, &garfo1);
Filosofo f2 = Filosofo(2, &garfo1, &garfo2);
Filosofo f3 = Filosofo(3, &garfo2, &garfo3);
Filosofo f4 = Filosofo(4, &garfo3, &garfo4);
Filosofo f5 = Filosofo(5, &garfo5, &garfo4);
```

Listing 7: Deadlock

Teríamos a seguinte situação:

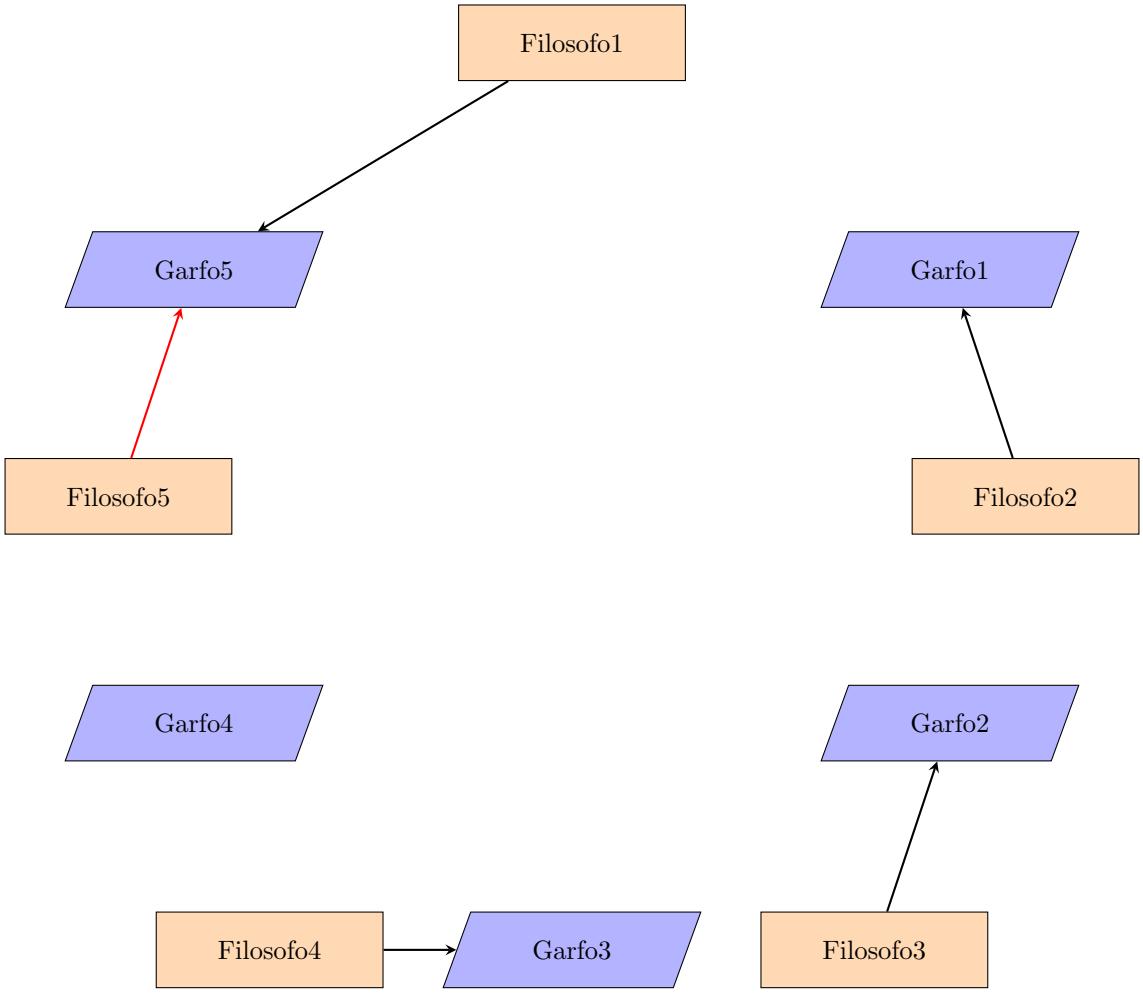


Figure 3: Problema sem Deadlock

Temos então o seguinte código final:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

using namespace std;

class Filosofo{
    private:
        int id;
        long vezesComeu;
        long vezesPensou;
    public:
        std::mutex* m1; // Garfo da esquerda
        std::mutex* m2;// Garfo da direita
        //Construtor do filosofo
        Filosofo(int id, std::mutex* garfoEsq, std::mutex* garfoDir)
        {
            this->id = id;
```

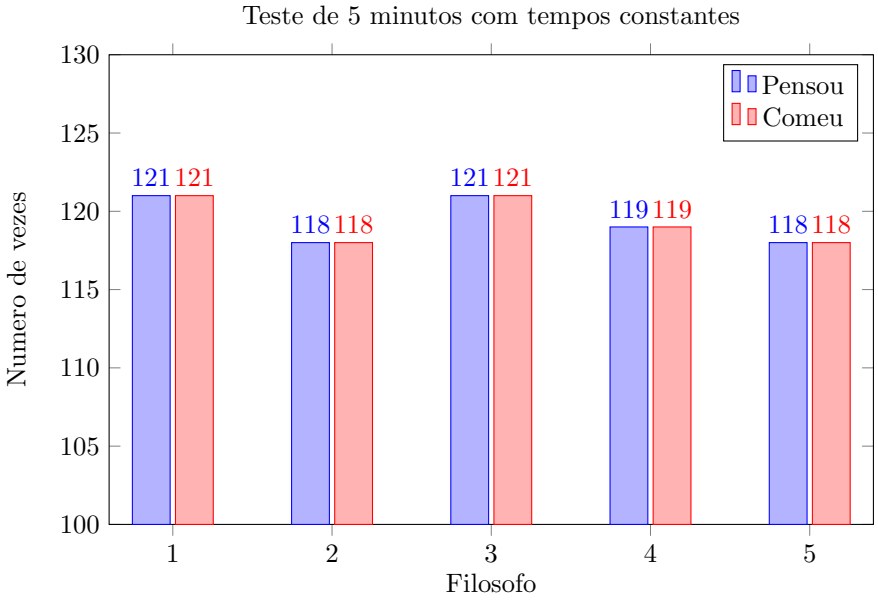
```
            this->m1 = garfoEsq;
            this->m2 = garfoDir;
            vezesComeu = 0;
            vezesPensou = 0;
        }
        void vidaFilosofo()
        {
            /*
                    Vida do filosofo:
                        Pensar,
                        // Depois de um tempo, quando ficar com fome
                        Comer -> pegar um garfo, e depois pegar outro
                        // Depois de satisfeito
                        Devolver os garfos e voltar a pensar
            */
            while(true)
            {
                pensa();
                pegarGarfos();
                come();
                soltarGarfos();
            }
        }
        void pensa()
        {
            cout << "Filosofo␣" << id << "␣pensando\n";
            vezesPensou++;
            this_thread::sleep_for(chrono::milliseconds(1000));
        }
        void come()
        {
            cout << "Filosofo␣" << id << "␣comendo\n";
            vezesComeu++;
            //Tempo que leva para comer
            this_thread::sleep_for(chrono::milliseconds(1000));
        }
        void pegarGarfos()
        {
            //Tenta pegar o garfo da direita
            this->m1->lock();
            //Tenta pegar o garfo da esquerda
            this->m2->lock();
        }
        void soltarGarfos()
        {
            //Solta os garfos
            this->m1->unlock();
            this->m2->unlock();
        }
};

int main()
{
    std::mutex garfo1;
    std::mutex garfo2;
    std::mutex garfo3;
```

```
    std::mutex garfo4;
    std::mutex garfo5;

    Filosofo f1 = Filosofo(1, &garfo5, &garfo1);
    Filosofo f2 = Filosofo(2, &garfo1, &garfo2);
    Filosofo f3 = Filosofo(3, &garfo2, &garfo3);
    Filosofo f4 = Filosofo(4, &garfo3, &garfo4);
    Filosofo f5 = Filosofo(5, &garfo5, &garfo4);

    std::thread threadsFilosofos[5] = {
        std::thread(&Filosofo::vidaFilosofo, &f1),
        std::thread(&Filosofo::vidaFilosofo, &f2),
        std::thread(&Filosofo::vidaFilosofo, &f3),
        std::thread(&Filosofo::vidaFilosofo, &f4),
        std::thread(&Filosofo::vidaFilosofo, &f5)
    };

    for(int i = 0; i<5; i++)
    {
        threadsFilosofos[i].join();
    }

    return 0;
}
```

Listing 8: Problema jantar dos filósofos

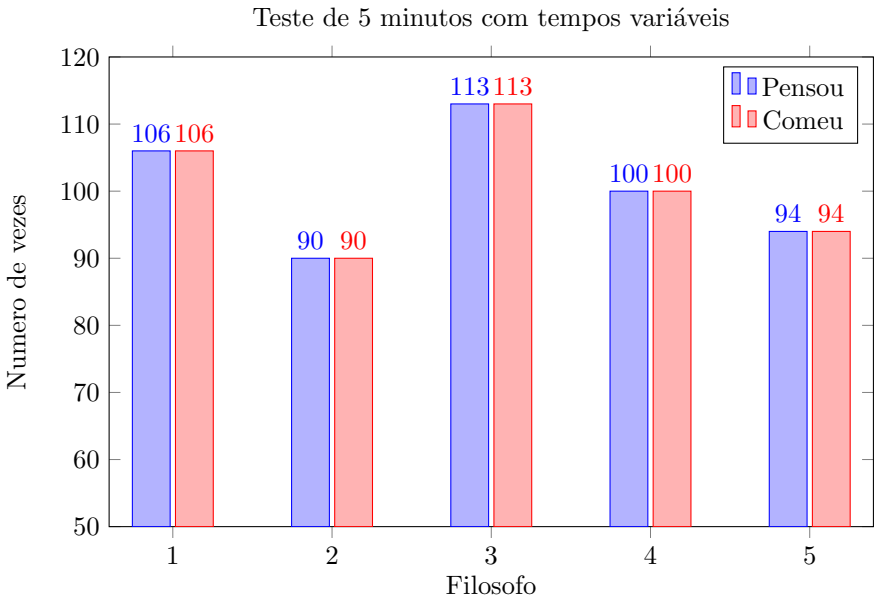
3 Testes

Foram realizados alguns testes, para o primeiro teste realizado, o programa teve uma duração de 5 minutos e todos os filósofos demoravam um segundo para comer e para pensar, tivemos o seguinte número de vezes que os filósofos comeram e pensaram 3:



Conseguimos perceber que todos os filósofos comeram e pensaram na mesma quantidades e todos fizeram numa quantidade bem semelhante.

Para o segundo teste foi utilizado um tempo para pensar e comer variável, podendo ser um valor entre 0 e 1999 ms, tivemos os seguintes resultados 3:



Não houve uma grande diferença entre o número de vezes em que cada filósofo comeu ou pensou. Ou seja, a solução do problema do Jantar dos Filósofos através de semáforos além de completa garante uma política de escalonamento justa na qual nenhum filósofo é privilegiado.

References

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Boston, MA, 4 edition, 2014.