

# TPC1: Filósofos Jantando

Alunos: Gustavo Willian Martins da Silva e Lorenzo Duarte More

Para o trabalho sobre sistemas concorrentes, solicitou-se a apresentação de duas soluções para o problema dos filósofos. As implementações escolhidas pelo grupo baseiam-se em Dijkstra e hierarquia de recursos. Os códigos apresentados foram modificados para seguir o modelo sequencial disponibilizado: alterou-se o tempo para pensar e comer e limitou-se o número de rodadas em 5. A seguir, serão descritas as principais características dessas implementações. A página 2 contém Figuras com a execução dos códigos, enquanto estes se encontram a partir da página 3.

A solução baseada em Dijkstra com alterações de Tanenbaum, na linguagem utilizada (C++20), tem o objetivo de evitar que dois filósofos peguem um garfo que já esteja sendo usado. Para isso, utiliza um array de semáforos, 1 para cada filósofo, que indica se o filósofo pode comer ou não. Os filósofos se sentam em um círculo em qual eles pensam, tentam pegar os garfos e, se conseguirem, comem e soltam os garfos. O estado de cada filósofo determina sua ação:

- THINKING, pensando;
- HUNGRY, esperando para comer;
- EATING, esta comendo.

A ação de pegar os garfos é controlada por um mutex de acordo com o array de semáforos, que determina a exclusão mútua dessas regiões críticas. Para evitar impasses, é preciso decidir, cuidadosamente, se um filósofo é capaz de comer ou não. Isso só acontece quando os vizinhos acabarem de comer. Após comer a refeição, o filósofo verifica o estado de seus vizinhos, testando se eles podem comer; se sim, eles pegam os garfos.

Não há ocorrência de *deadlock*, pois o filósofo só pode comer quando sabe que ambos os garfos estão disponíveis. A verificação e a atualização do estado do semáforo ocorrem de maneira atômica por conta do *mutex*. Dessa forma, o acesso exclusivo aos garfos é garantido.

Já o *starvation* pode ocorrer pois não há uma ordem de quem vai ser o próximo a comer. Digamos que um filósofo tenha o azar de um vizinho comer, ele espera até ele acabar, e quando ele tenta comer, o outro vizinho, que estava na mesma situação, começa a comer antes dele. Isso o leva a passar fome.

A solução envolvendo Hierarquia de Recursos propõe uma ordem parcial para os recursos

(garfos), e cada filósofo tenta pegar sempre os mesmos garfos, mas tenta primeiro o garfo com o menor número na ordem. Para exemplificar, o filósofo 1 tenta sempre pegar os garfos 1 e 2, nesta ordem.

Os mecanismos utilizados na sincronização são 6 semáforos binários: 1 para cada garfo e 1 para imprimir mensagem formatada na tela (por exemplo, "filósofo 3 está pensando"). Na linguagem utilizada (C++11), tais mecanismos são implementados através de *mutex*. Dessa forma, apenas um processo entrará em sua seção crítica por vez.

A hierarquia de recursos evita o *deadlock* porque garante que, pelo menos, um filósofo consiga comer independente se os outros filósofos pegarem um garfo ao mesmo tempo. Pense, por exemplo, em uma ordem de 1 a 5 (tanto para garfos quanto para filósofos): começando do filósofo 1, os filósofos pegam seus respectivos garfos com número de ordem menor, o filósofo 5 não poderá pegar o garfo 1, e o garfo 5 estará disponível para o filósofo 4 comer.

Porém, essa solução não é justa e não consegue prevenir *starvation*, tendo em vista que não tem nenhuma garantia de acesso a recursos para processos mais lentos. Ou seja, é possível que um filósofo espere indefinidamente para acessar sua seção crítica e, literalmente, morrer de fome.

É importante destacar que as soluções concorrentes executaram de forma mais rápida, quando comparadas com a solução sequencial disponibilizada. O fato de 2 processos poderem acontecer ao mesmo tempo torna a execução do programa mais dinâmica, pois não existe tanto tempo de espera. As soluções apresentadas terminaram em menos de 1 minuto, enquanto a sequencial demorou mais que o dobro, conforme as Figuras da página 2.

## Dijkstra

```
dp_14
0 is thinking 2000ms
1 is thinking 2000ms
2 is thinking 2000ms
3 is thinking 2000ms
4 is thinking 2000ms
    0 is State::HUNGRY
        0 is eating 3000ms
            1 is State::HUNGRY
                2 is State::HUNGRY
                    2 is eating 3000ms
                        3 is State::HUNGRY
                            4 is State::HUNGRY
0 is thinking 2000ms
    4 is eating 3000ms
2 is thinking 2000ms
    1 is eating 3000ms
        0 is State::HUNGRY
            2 is State::HUNGRY
4 is thinking 2000ms
1 is thinking 2000ms
    0 is eating 3000ms
        3 is eating 3000ms
            4 is State::HUNGRY
                1 is State::HUNGRY
0 is thinking 2000ms
    1 is eating 3000ms
        4 is eating 3000ms
3 is thinking 2000ms
    0 is State::HUNGRY
        3 is State::HUNGRY
1 is thinking 2000ms
    2 is eating 3000ms
4 is thinking 2000ms
    0 is eating 3000ms
        1 is State::HUNGRY
            4 is State::HUNGRY
2 is thinking 2000ms
    3 is eating 3000ms
0 is thinking 2000ms
    1 is eating 3000ms
        2 is State::HUNGRY
            0 is State::HUNGRY
3 is thinking 2000ms
    2 is eating 3000ms
1 is thinking 2000ms
    4 is eating 3000ms
2 is thinking 2000ms
    3 is State::HUNGRY
        1 is State::HUNGRY
4 is thinking 2000ms
    2 is State::HUNGRY
        4 is State::HUNGRY
1 is thinking 2000ms
3 is thinking 2000ms
    0 is eating 3000ms
        2 is eating 3000ms
            1 is State::HUNGRY
                3 is State::HUNGRY
0 is thinking 2000ms
2 is thinking 2000ms
    4 is eating 3000ms
        1 is eating 3000ms
            0 is State::HUNGRY
                2 is State::HUNGRY
4 is thinking 2000ms
    0 is eating 3000ms
        3 is eating 3000ms
            4 is State::HUNGRY
                3 is State::HUNGRY
3 is thinking 2000ms
    2 is eating 3000ms
        4 is eating 3000ms
            3 is State::HUNGRY
                3 is eating 3000ms
Tempo de execução: 41002 ms
```

## Hierarquia de Recursos

```
dining Philosophers C++11 with Resource hierarchy
5 thinks 2000ms
4 thinks 2000ms
1 thinks 2000ms
2 thinks 2000ms
3 thinks 2000ms
    5 is hungry
        5 eats 3000ms
            1 is hungry
                4 is hungry
                    2 is hungry
                        2 eats 3000ms
                            3 is hungry
0 is thinking 2000ms
    4 eats 3000ms
2 thinks 2000ms
    1 eats 3000ms
        5 is hungry
            2 is hungry
4 thinks 2000ms
    3 eats 3000ms
1 thinks 2000ms
    5 eats 3000ms
        4 is hungry
            1 is hungry
3 thinks 2000ms
    2 eats 3000ms
5 thinks 2000ms
    4 eats 3000ms
        3 is hungry
            5 is hungry
2 thinks 2000ms
    1 eats 3000ms
4 thinks 2000ms
    3 eats 3000ms
1 thinks 2000ms
    2 eats 3000ms
3 thinks 2000ms
    5 eats 3000ms
        1 is hungry
            3 is hungry
5 thinks 2000ms
    4 eats 3000ms
2 thinks 2000ms
    1 eats 3000ms
4 thinks 2000ms
    3 eats 3000ms
1 thinks 2000ms
    5 eats 3000ms
3 thinks 2000ms
    4 eats 3000ms
        3 is hungry
            5 is hungry
2 thinks 2000ms
    1 eats 3000ms
4 thinks 2000ms
    3 eats 3000ms
1 thinks 2000ms
    5 eats 3000ms
3 thinks 2000ms
    2 eats 3000ms
        1 is hungry
            3 is hungry
4 eats 3000ms
1 eats 3000ms
3 eats 3000ms
Tempo de execução: 41002 ms
```

## Código Sequencial

```
T4
E4
    T4
    E4
        T4
        E4
            T4
            E4
                T5
                E5
                    T5
                    E5
                        T5
                        E5
                            T5
                            E5
Sequential Dinner took 2m5.006687922s
```

## Código de Dijkstra (C++20)

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

constexpr const size_t N = 5; // number of philosophers (and forks)
int rodadas = 5; // number of philosophers (and forks)

enum class State
{
    THINKING = 0, // philosopher is THINKING
    HUNGRY = 1,   // philosopher is trying to get forks
    EATING = 2,   // philosopher is EATING
};

size_t inline left(size_t i)
{
    // number of the left neighbor of philosopher i, for whom both forks are available
    return (i - 1 + N) % N; // N is added for the case when i - 1 is negative
}

size_t inline right(size_t i)
{
    // number of the right neighbour of the philosopher i, for whom both forks are available
    return (i + 1) % N;
}

State state[N]; // array to keep track of everyone's both_forks_available state

std::mutex critical_region_mtx; // mutual exclusion for critical regions for
// (picking up and putting down the forks)
std::mutex output_mtx; // for synchronized cout (printing THINKING/HUNGRY/EATING status)

// array of binary semaphores, one semaphore per philosopher.
// Acquired semaphore means philosopher i has acquired (blocked) two forks
std::binary_semaphore both_forks_available[N]
{
    std::binary_semaphore{0}, std::binary_semaphore{0},
    std::binary_semaphore{0}, std::binary_semaphore{0},
    std::binary_semaphore{0}
};

size_t my_rand(size_t min, size_t max)
{
    static std::mt19937 rnd(std::time(nullptr));
    return std::uniform_int_distribution<>(min, max)(rnd);
}

void test(size_t i)
// if philosopher i is hungry and both neighbours are not eating then eat
{
    // i: philosopher number, from 0 to N-1
    if (state[i] == State::HUNGRY &&
        state[left(i)] != State::EATING &&
        state[right(i)] != State::EATING)
    {
        state[i] = State::EATING;
        both_forks_available[i].release(); // forks are no longer needed for this eat session
    }
}

void think(size_t i)
```

```

{
    size_t duration = 2000;
    {
        std::lock_guard<std::mutex> lk(output_mtx); // critical section for uninterrupted print
        std::cout << i << " is thinking " << duration << "ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
}

void take_forks(size_t i)
{
    {
        std::lock_guard<std::mutex> lk{critical_region_mtx}; // enter critical region
        state[i] = State::HUNGRY; // record fact that philosopher i is State::HUNGRY
        {
            std::lock_guard<std::mutex> lk(output_mtx); // critical section for uninterrupted print
            std::cout << "\t\t" << i << " is State::HUNGRY\n";
        }
        test(i); // try to acquire (a permit for) 2 forks
    } // exit critical region
    both_forks_available[i].acquire(); // block if forks were not acquired
}

void eat(size_t i)
{
    size_t duration = 3000;
    {
        std::lock_guard<std::mutex> lk(output_mtx); // critical section for uninterrupted print
        std::cout << "\t\t\t\t" << i << " is eating " << duration << "ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
}

void put_forks(size_t i)
{
    std::lock_guard<std::mutex> lk{critical_region_mtx}; // enter critical region
    state[i] = State::THINKING; // philosopher has finished State::EATING
    test(left(i)); // see if left neighbor can now eat
    test(right(i)); // see if right neighbor can now eat
    // exit critical region by exiting the function
}

void philosopher(size_t i)
{
    // Registra o tempo inicial
    auto start_time = std::chrono::high_resolution_clock::now();

    int j = 0;
    while (j < 5)
    {
        // repeat forever
        think(i); // philosopher is State::THINKING
        take_forks(i); // acquire two forks or block
        eat(i); // yum-yum, spaghetti
        put_forks(i); // put both forks back on table and check if neighbours can eat
        j++;
    }

    rodadas = rodadas - 1;

    if(rodadas == 0){
        // Registra o tempo final
        auto end_time = std::chrono::high_resolution_clock::now();

        // Calcula a diferença de tempo em milissegundos
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

        // Exibe o tempo de execução em milissegundos
    }
}

```

```

    std::cout << "Tempo de execução: " << duration.count() << " ms" << std::endl;
}

}

int main() {

    std::cout << "dp_14\n";

    std::jthread t0([&] { philosopher(0); }); // [&] means every variable outside the ensuing lambda
    std::jthread t1([&] { philosopher(1); }); // is captured by reference
    std::jthread t2([&] { philosopher(2); });
    std::jthread t3([&] { philosopher(3); });
    std::jthread t4([&] { philosopher(4); });

}

```

## Código de Hierarquia de Recursos (C++11)

```

#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>
#include <random>
#include <ctime>

using namespace std;

int myrand(int min, int max) {
    static mt19937 rnd(time(nullptr));
    return uniform_int_distribution<>(min,max)(rnd);
}

void philosopher(int ph, mutex& ma, mutex& mb, mutex& mo) {
    for (int i=0; i < 5; i++) { // prevent thread from termination
        int duration = 2000;
        {
            // Block { } limits scope of lock
            lock_guard<mutex> gmo(mo);
            cout<<ph<<" thinks "<<duration<<"ms\n";
        }
        this_thread::sleep_for(chrono::milliseconds(duration));
        {
            lock_guard<mutex> gmo(mo);
            cout<<"\t\t"<<ph<<" is hungry\n";
        }
        lock_guard<mutex> gma(ma);
        // sleep_for() Delay before seeking second fork can be added here but should not be required.
        lock_guard<mutex> gmb(mb);
        duration = 3000;
        {
            lock_guard<mutex> gmo(mo);
            cout<<"\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
        }
        this_thread::sleep_for(chrono::milliseconds(duration));
    }
}

int main() {
    // Registra o tempo inicial
    auto start_time = std::chrono::high_resolution_clock::now();

    cout<<"dining Philosophers C++11 with Resource hierarchy\n";
    mutex m1, m2, m3, m4, m5; // 5 forks are 5 mutexes
}

```

```

mutex mo;          // for proper output
// 5 philosophers are 5 threads
thread t1([&] {philosopher(1, m1, m2, mo);});
thread t2([&] {philosopher(2, m2, m3, mo);});
thread t3([&] {philosopher(3, m3, m4, mo);});
thread t4([&] {philosopher(4, m4, m5, mo);});
thread t5([&] {philosopher(5, m1, m5, mo);}); // Force a resource hierarchy
t1.join(); // prevent threads from termination
t2.join();
t3.join();
t4.join();
t5.join();

// Registra o tempo final
auto end_time = std::chrono::high_resolution_clock::now();

// Calcula a diferença de tempo em milissegundos
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

// Exibe o tempo de execução em milissegundos
std::cout << "Tempo de execução: " << duration.count() << " ms" << std::endl;
}

```

## Bibliografia

Códigos: [Dining philosophers problem – Wikipédia, a enciclopédia livre \(wikipedia.org\)](https://pt.wikipedia.org/wiki/Dining_philosophers_problem)