**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

## 1. Análise Geral

Com objetivo de entender mais sobre programação paralela, este trabalho apresenta uma solução para o problema dos filósofos jantando em C++ usando concorrência, desenvolvida por Gustavo Pantuza. Além disso, para fins de comparação, o grupo também testou como ficaria esse algoritmo sem a concorrência.

O problema "Filósofos Jantando" consiste em um problema de sincronização. Cada filósofo altera entre comer e pensar, sendo que para comer, o filósofo necessita de dois garfos. Caso o filósofo consiga pegar os dois garfos necessários, ele come durante determinado tempo e depois realoca os garfos na mesa, voltando a pensar.

## 2. Mecanismos Utilizados

A solução do problema foi baseada no uso de semáforos. É através deles que há exclusão mútua por completo, evitando *starvation* e *deadlock*, mantendo a atenção na seção crítica, que no problema apresentado é o ato de comer.

## 3. Implementação

Na solução proposta em C++, cada garfo é representado como um *mutex* (teremos um vetor de 5 mutexes, visto que temos 5 filósofos), controlados pela função *lock_guard*, realizando a função de semáforo. Logo após o filósofo comer, os garfos que estavam com ele são liberados, e o sistema operacional passa a garantir que outras *threads* interessadas no recurso liberado sejam escalonadas para executar e tentar pegar os garfos livres.

Além disso, o método *think*(), que representa o ato de pensar dos filósofos, coloca a *thread* para dormir durante um tempo randômico de 0 a 5 segundos. Já o método *eat*() tenta pegar os garfos para comer (*lock_acquire*). Ao fazer o *lock* de cada garfo, a biblioteca padrão de C++ garantirá que não haverá deadlock. Feito isso, é utilizado o método *lock_guard*(), que garante que ao final da execução os *mutexes* sejam liberados sem precisar chamar o método *unlock.* No método *eat,* a thread é posta para dormir em um tempo randômico entre 0 e 10 segundos. A ideia é que os filósofos executem aproximadamente o mesmo número de vezes as ações pensar e comer.

## 4. Deadlock e Starvation

O *deadlock* poderia ocorrer caso todos os filósofos pegassem apenas um garfo e ficassem esperando infinitamente pelo segundo garfo para comer. Como mencionado anteriormente, em C++, ao darmos um *lock,* a biblioteca garante que não haja *deadlock*.

O *starvation* pode ocorrer em casos como: um filósofo tenta pegar dois garfos e consegue apenas pegar um, então ele devolve o garfo e aguarda um tempo fixo para tentar novamente. Caso ele não consiga pegar os dois garfos em momento algum, ele morre de fome, consequência gerada por nunca conseguir comer. Desta forma, foi necessário evitar o *starvation* através da edição do código, para que o mesmo rode até que cada filósofo tenha comido ao menos 10 vezes antes do término da execução.

Em contrapartida, no código sequencial os filósofos pensam e comem um após o outro, sendo que o processo de pensar e comer de um filósofo não pode ser sobreposto por outro filósofo que deseja comer ou pensar, visto que eles esperam durante a execução para que os outros filósofos que não terminaram de comer possam comer até 10x. Desta forma, não há *starvation,* porém, aumenta o tempo de execução.

## 5. Análise de Concorrência

No código de Gustavo Pantuza, são inicializadas cinco *threads*, uma para cada filósofo. Cada filósofo pensa aproximadamente 30 vezes e come aproximadamente 10 vezes. Estas ações ocorrem em 0.745849 milisegundos.

Já no código não concorrente, desenvolvido pelo grupo para fins de comparação, os filósofos pensam 20 vezes e comem 10 vezes, em um tempo de 0.420794 milisegundos .

O código sequencial implementado produziu sua saída com menor tempo de execução, isso se deu pela maior simplicidade do código desenvolvido pelo grupo (não concorrente) em relação ao código concorrente. É possível concluir isto a partir da análise do código concorrente, que espera que todos os filósofos comam ao menos 10 vezes antes de terminar sua execução. Esta edição foi necessária dado que a estratégia adotada nesta implementação faz com que os filósofos disputem entre si os recursos, não se preocupando com a velocidade de cada filósofo e sua sincronização. Os filósofos mais lentos podem, portanto, atrasar o término do código, pois fazem com que os outros esperem que os filósofos que estão comendo terminem de comer no contexto da aplicação.

**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

6. **Dumps de Tela para ilustrar o funcionamento do código.**


**Dumps do código sequencial:**

```
Aristotle is thinking
Aristotle started eating
Aristotle finished eating
Aristotle is thinking
Platon is thinking
Platon started eating
Platon finished eating
Platon is thinking
Descartes is thinking
Descartes started eating
Descartes finished eating
Descartes is thinking
Kant is thinking
Kant started eating
Kant finished eating
Kant is thinking
Nietzsche is thinking
Nietzsche started eating
Nietzsche finished eating
Nietzsche is thinking
Aristotle is thinking
...
Platon is thinking
Platon started eating
Platon finished eating
Platon is thinking
Descartes is thinking
Descartes started eating
Descartes finished eating
Descartes is thinking
Kant is thinking
Kant started eating
Kant finished eating
Kant is thinking
Nietzsche is thinking
Nietzsche started eating
Nietzsche finished eating
Nietzsche is thinking
Execution time (ms): 0.420794
```

**Dumps do código concorrente:**


```
Philosopher 0 thinking..
Philosopher 0 taking forks..
Philosopher 1 thinking..
Philosopher 1 taking forks..
Philosopher 0 eating..
Philosopher 0 releasing fork..
Philosopher 2 thinking..
Philosopher 2 taking forks..
Philosopher 2 eating..
Philosopher 2 releasing fork..
...
Philosopher 4 taking forks..
Philosopher 0 eating..
Philosopher 0 releasing fork..
Philosopher 0 thinking..
Philosopher 1 eating..
Philosopher 1 releasing fork..
Philosopher 0 taking forks..
Philosopher 4 eating..
Philosopher 4 releasing fork..
Philosopher 4 thinking..
Philosopher 4 taking forks..
Philosopher 0 eating..
Philosopher 0 releasing fork..
Philosopher 4 eating..
Philosopher 4 releasing fork..
Philosopher 4 thinking..
Philosopher 4 taking forks..
Philosopher 4 eating..
Philosopher 4 releasing fork..
Philosopher 4 thinking..
Philosopher 4 taking forks..
Philosopher 4 eating..
Philosopher 4 releasing fork..
Philosopher 4 thinking..
Philosopher 4 taking forks..
Philosopher 4 eating..
Philosopher 4 releasing fork..
Execution Time: 0.745849
```

**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

## 7. Código fonte

**Código Sequencial:**

```
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <chrono>
#include <thread>

using namespace std;

class chopstiks {
private:
    bool take;
    int idCount = 0;

public:
    int id;

    chopstiks() {
        take = false;
        id = idCount + 1;
        idCount++;
    }

    chopstiks(int id) {
        take = false;
        id = id;
        idCount++;
    }

    void take_chopstiks() { // validar se pode pegar o garfo
        take = true;
    }

    void drop_chopstiks() {
        take = false;
    }
};
```

**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

```
class philosopher {
private:
   int                     id;
   const string            name;
public:
   chopstiks              left_chopstik;
   chopstiks              right_chopstik;

   philosopher(int id, const string name, chopstiks lf, chopstiks rf) :
      id(id), name(name), left_chopstik(lf), right_chopstik(rf)
   {}

   string getName () {
      return name;
   }

   int getId () {
      return id;
   }
};

void think(philosopher ph) {
   cout <<  ph.getName() << " is thinking" << "\n";
}

void eat(philosopher ph) {
   think(ph);
   ph.left_chopstik.take_chopstiks();
   ph.right_chopstik.take_chopstiks();
   cout <<  ph.getName() << " started eating" << "\n";
   cout <<  ph.getName() << " finished eating" << "\n";
   ph.left_chopstik.drop_chopstiks();
   ph.right_chopstik.drop_chopstiks();
   think(ph);
}
void dine() {
   vector<string> names = {"Aristotle","Platon","Descartes","Kant","Nietzsche"};
   vector<chopstiks> chopstiks = {0,1,2,3,4};
   vector<philosopher> philosophers;

   int j = 1;
   for(int i = 0; i < 5; i++, j++) {
      if(i == 4){
         j = 0;
      }
      philosophers.emplace_back(move(philosopher(i, names.at(i), chopstiks.at(i), chopstiks.at(j))));
   }
   for(int j = 0; j < 10; j++) {
      for(int i = 0; i < 5; i++) {
         eat(philosophers.at(i));
      }
   }

};

int main() {
   auto t1 = chrono::high_resolution_clock::now();
   dine();
   auto t2 = chrono::high_resolution_clock::now();
   chrono::duration<double, milli> result = t2 - t1;
   cout << "Execution time (ms): " << result.count() << "\n";
   return 0;
}
```

**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

**Código concorrente:**

```
19  #include <iostream>
20  #include <thread>
21  #include <vector>
22  #include <chrono>
23  #include <mutex>
24
25  using namespace std;
26  using std::thread;
27  using std::cout;
28  using std::endl;
29  using std::vector;
30  using std::this_thread::sleep_for;
31  using std::rand;
32  using std::chrono::milliseconds;
33  using std::mutex;
34  using std::lock;
35  using std::lock_guard;
36  using std::adopt_lock;
37
38  mutex cout_lock;
39
40  /**
41   * Class that implements a dinner Table with all the forks resources
42   */
43  class Table {
44    public:
45
46      /* The shared resource that each Philosopher will concur */
47      vector<mutex> forks;
48
49      Table (): forks(5)
50      {
51      }
52  };
53
54
55
56  /**
57   * Class that implements the existence of a Philosopher that is
58   * to think and eat
59   */
60  class Philosopher {
61
62    public:
63      Philosopher (const int id, Table& table)
64      :philosopher_id(id),
65       table(table)
66      {
67      }
68
69      /*
70       * Spawns the parallel execution of a Philosopher that shares the same
71       * table with other Philosophers
72       */
73      thread* Spawn ()
74      {
75        return new thread([=] { exist(); });
76      }
77
```

**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

```
78  private:
79    /* The Philosopher identifier */
80    int philosopher_id;
81
82    /* A reference to the shared table instance */
83    Table& table;
84
85    /**
86     * The existence of a Philosopher
87     */
88    void exist ()
89    {
90      for (int i = 0; i < 10; i++) {
91
92        think();
93        eat();
94      }
95    }
96
97    /**
98     * The randomized time that each Philosopher spends thinking
99     */
100   void think ()
101   {
102     cout_lock.lock();
103     cout << "Philosopher " << philosopher_id << " thinking.." << endl;
104     //sleep_for(milliseconds(rand() % 5000));
105     cout_lock.unlock();
106   }
107
108   /**
109    * The eating process of each Philosopher instance. First of all, a
110    * Philosopher tries to lock both forks that he needs to eat. There
111    * are 5 forks. If one fork is acquired by any other Philosopher, the
112    * thread will wait until the resource/fork is released.
113    *
114    * When both forks are acquired by the current thread execution, it time
115    * to the Philosopher to eat. At the end of this process both forks are
116    * released.
117    */
118   void eat ()
119   {
120
121     int left = philosopher_id;
122     int right = (philosopher_id + 1) % table.forks.size();
123
124     cout_lock.lock();
125     cout << "Philosopher " << philosopher_id << " taking forks.." << endl;
126     cout_lock.unlock();
127
128     /* Locks boths mutexes without any deadlock */
129     lock(table.forks[left], table.forks[right]);
130
131     /* At the end of this scope mutexes will be released */
132     lock_guard<mutex> left_fork(table.forks[left], adopt_lock);
133     lock_guard<mutex> right_fork(table.forks[right], adopt_lock);
134
135     cout_lock.lock();
```

**Integrantes:** André Sacilotto Santos, Daniela Rigoli e Morgana Weber
**Disciplina:** Fundamentos de Processamento Paralelo e Distribuído
**Turma:** 031 - 2021/2

```
136         cout << "Philosopher " << philosopher_id << " eating.." << endl;
137         //sleep_for(milliseconds(rand() % 10000));
138 
139         cout << "Philosopher " << philosopher_id << " releasing fork.." << endl;
140         cout_lock.unlock();
141       }
142   };
143 
144 
145 
146   int
147   main (int argc, char* argv[])
148   {
149     cout << "THE PHILOSOPHERS DINNER" << endl << endl;
150     Table table;
151 
152     vector<thread *> threads(5);
153 
154     auto t1 = chrono::high_resolution_clock::now();
155     for(int i=0; i < 5; i++) {
156 
157       Philosopher* philosopher = new Philosopher(i, table);
158       threads[i] = philosopher->Spawn();
159     }
160 
161     for(int i=0; i < 5; i++) {
162 
163       threads[i]->join();
164     }
165     auto t2 = chrono::high_resolution_clock::now();
166     chrono::duration<double, milli> result = t2 - t1;
167 
168 
169     cout << "Execution Time: " << result.count() << "\n";
170 
171     return EXIT_SUCCESS;
172   }
```