Arthur Zanella Lovato, Caroline Lewandowski, Felipe Roque

ANÁLISE DE SOLUÇÕES PARA O PROBLEMA DOS FILÓSOFOS

O presente trabalho descreve as soluções para o problema dos Filósofos Jantando encontradas pelo grupo da disciplina Fundamentos de Processamento Paralelo e Distribuído do semestre 2022/02, inicialmente formulado por Dijkstra, que exemplifica o problema de concorrência ao propor o cenário de uma mesa de jantar que dispõe o total de 5 garfos, com cinco filósofos que alternam entre pensar e comer, sendo que os filósofos deverão utilizar dois garfos para comer, faz-se necessário distribuir o recurso garfo entre os mesmos, considerando que eles possuem acesso ao garfo disposto na direita e esquerda. Desta forma, os dois cenários a serem evitados são *deadlock*, onde os cinco filósofos ficarão trancados por pegar apenas um garfo, não tendo acesso ao outro garfo ou como soltar o primeiro e um cenário onde um dos filósofos nunca tem a possibilidade de comer, pois devido à concorrência e limitações de recurso os outros filósofos sempre conseguem realizar a ação na sua frente, sendo este chamado *starvation*. A análise e comparação das soluções estará disposta na presente página, seguido dos *dumps* de tela resultantes de suas implementações, e, por fim, da terceira página em diante estarão dispostos os códigos utilizados.

Para evitar *deadlock*, a primeira solução consiste na implementação da Solução de Arbitragem, adicionando mais um elemento ao cenário que servirá como *mutex*: Um árbitro ou, no contexto, um garçom que dará permissões e ordens aos filósofos e dessa forma os filósofos sempre irão solicitar permissão para o garçom antes de uma ação. O garçom irá garantir que um filósofo só possa pegar ambos garfos de uma vez ou então não pegará nenhum. Em relação à implementação, o controle que este garçom estabelece está declarado no método *onReceive* da classe *Waiter*, e, baseando-se na mensagem que recebe como parâmetro o garçom pode controlar as próximas ações: Se um filósofo passar a mensagem *Hungry*, ele apenas permitirá que o *status* dos dois garfos seja atualizado para *Used* após verificar previamente se ambos possuem status *Free*, caso contrário ele passará a mensagem *Think* para o filósofo. Ao receber a mensagem *FinishEat* do filósofo, os garfos voltam ao status *Free* e o filósofo recebe a mensagem para pensar novamente. Este comportamento irá prevenir a ocorrência de *deadlock* no cenário ao permitir que o garçom, que serve como *mutex*, responda apenas uma solicitação por vez, desta forma, dois filósofos não podem receber permissão para comer ao mesmo tempo pois as verificações quanto ao status dos garfos são executadas de forma atômica.

A segunda solução selecionada foi a *Chandy/Misra* que propõe uma comunicação entre os filósofos e define dois estados para o garfo (ex.: sujo e limpo). Os cenários sempre iniciam com todos os garfos sujos, e, assim que um filósofo desejar comer, para cada garfo que este não possuir, o recurso deve ser solicitado para seu vizinho. Ao receber a requisição do recurso, o vizinho pode manter o garfo para ele se estiver limpo e passar adiante se estiver sujo, devendo limpá-lo antes, além disso, para o caso de um par de filósofos estar disputando o garfo, este será concedido ao filósofo que possuir o menor *id*. Estes dois últimos protocolos listados irão evitar que ocorra *starvation* devido a forma como eles implementam priorizações para os filósofos, sendo uma delas a prioridade fixa de um filósofo sobre outro e outra que é definida no decorrer da execução: fazer com que o filósofo recuse passar o garfo limpo garantirá sua prioridade para jantar em dado momento.

Em relação à implementação, as regras citadas previamente são definidas no método *philosopher*, e é utilizado o mecanismo de *mutex* para controlar a posse dos garfos: Inicialmente o filósofo segura o primeiro garfo, o *mutex* é utilizado para trancá-lo, então o id do filósofo é adicionado na variável *heldBy*, utilizada para controlar quem tem a posse do garfo. Para o caso do filósofo não ter pego o segundo garfo, a requisição do *mutex* para dar *lock* neste garfo estará contida em um laço de repetição e será feita uma condição para verificar se o primeiro garfo já foi utilizado por ele, utilizando a variável *prevusedBy*, que servirá como uma forma de condicionar qual filósofo terá prioridade sobre ele, simulando o estado de garfos sujos e limpos. Para o caso do garfo "estar sujo", ele deverá ser retirado da atribuição ao filósofo em *heldBy* e do *lock* pelo *mutex*. Ao obter o segundo garfo, é verificado se o filósofo possui o primeiro, se não, ele é requisitado novamente, no momento em que o filósofo consegue os dois garfos seu estado muda para eating e após algum tempo ele os solta, o *id* do filósofo é adicionado em *prevusedBy* para indicar que os dois garfos estão sujos.

Por fim, para estas soluções foi comparado o tempo gasto para cada filósofo jantar em, aproximadamente, um total de dez vezes, tendo 2 segundos ao total entre pensar e comer, a execução em uma solução sequencial concluiu com 5 filósofos jantando no total, 10 vezes cada em 1 minuto e 40 segundos. Já a solução de arbitragem obteve uma execução de 33.26 segundos para chegar ao mesmo resultado. Por sua vez, a solução Chandy/Misra fez com que os filósofos jantassem um total de 73 vezes em 38 segundos, todos estes resultados podem ser conferidos na próxima página.

Desta forma, pode-se considerar que as duas soluções possuem um tempo de execução similar para obter um resultado próximo. Em questão de performance, embora permita que vários filósofos pensem ao mesmo tempo, foi observado que a solução de arbitragem é uma abordagem que reduz significativamente o paralelismo por permitir apenas um filósofo iniciar sua janta por vez devido ao *mutex* definido pelo garçom, diferente da solução *Chandy/Misra* que torna possível com que dois filósofos comecem a jantar ao mesmo tempo, portanto, foi concluído que esta versão permite mais concorrência durante a execução.

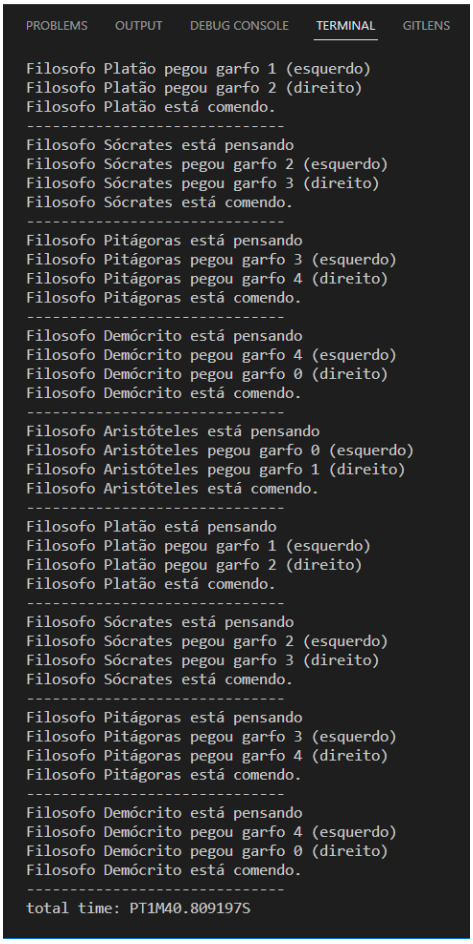

Figura 1 - Resultados de execução da solução sequencial

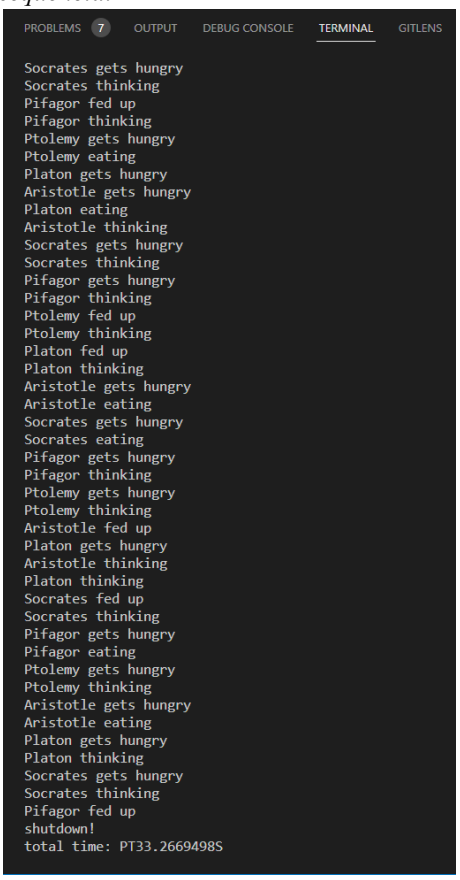

Figura 2 - Resultados de execução da solução de arbitragem

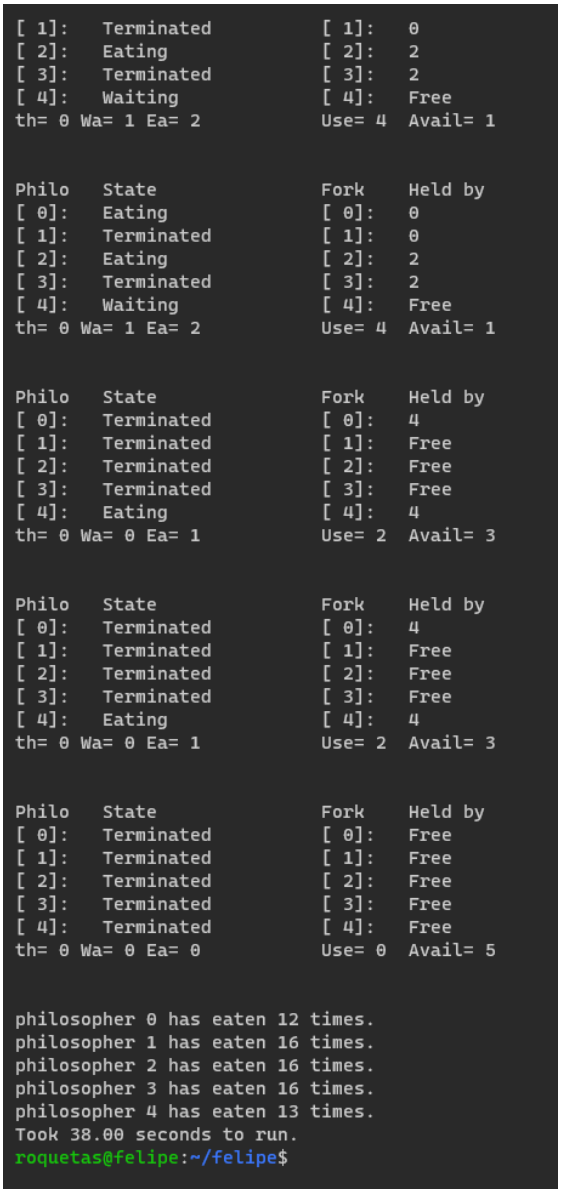

Figura 3 - Resultados de execução da solução Chandy/Misra

CÓDIGO SOLUÇÃO SEQUENCIAL:

Classe App

```
import java.time.Duration;
import java.time.Instant;

public class App {

    public Filosofos filosofo;
    public int[] garfos;
    public int garfoEsquerdo;
    public int garfoDireito;

    public static void main(String[] args) throws Exception {
        String[] nomes = { "Aristóteles", "Platão", "Sócrates",
"Pitágoras", "Demócrito" };

        Instant start = Instant.now();

        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 5; j++) {
                Filosofos filosofo = new Filosofos(nomes[j]);
                filosofo.pensar(j);
                filosofo.comer(j);
            }
        }

        Instant end = Instant.now();
        System.out.println("total time: " + Duration.between(start,
end));
    }
}
```

Classe Filosofos

```
public class Filosofos {

    public String nome;
    public int garfoEsquerdo;
    public int garfoDireito;

    public Filosofos(String nome) {
        this.nome = nome;
    }

    public void comer(int i) throws InterruptedException {
        garfoEsquerdo = i;
        System.out.println("Filosofo " + getNome() + " pegou garfo " +
garfoEsquerdo + " (esquerdo)");
        garfoDireito = (i + 1) % 5;
        System.out.println("Filosofo " + getNome() + " pegou garfo " +
garfoDireito + " (direito)");
        System.out.println("Filosofo " + getNome() + " está comendo.");
        System.out.println("------------------------------");
        Thread.sleep(1000);
    }

    public void pensar(int i) throws InterruptedException {
        System.out.println("Filosofo " + getNome() + " está pensando");
        Thread.sleep(1000);
    }
```

```
    public String getNome() {
        return this.nome;
    }
}

    }

    public String getNome() {
        return this.nome;
    }
}
```

CÓDIGO SOLUÇÃO ARBITRAGEM:

Classe Philosopher

```
package DinningPhilosophers;

import java.time.Duration;
import java.time.Instant;
import java.util.HashMap;
import java.util.Map;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
import akka.actor.UntypedActor;

public class Philosopher extends UntypedActor {
    public static Props mkProps(String aName, ActorRef aWeiter) {
        return Props.create(Philosopher.class, aName, aWeiter);
    }

    private String name;
    private ActorRef weiter;
    private Map<String, Integer> eatingSeq = new HashMap<String,
Integer>();
    private Instant start = Instant.now();
    private Instant end = Instant.now();
    private static final int EATING_MAX = 10;
    private static final int THINK_TIME = 1000;
    private static final int EAT_TIME = 1000;

    private Philosopher(String aName, ActorRef aWeiter) {
        name = aName;
        weiter = aWeiter;
        this.eatingSeq.put(name, 0);

        // Let`s introduce ourselves to Waiter
        aWeiter.tell(new Messages.Introduce(aName), getSelf());
    }

    @Override
    public void onReceive(Object message) throws Exception {
```

```
        int value = 0;

        if (!this.eatingSeq.containsValue(EATING_MAX)) {
            if (message instanceof Messages.Think) {
                System.out.println(name + " thinking");
                Thread.sleep(THINK_TIME);
                System.out.println(name + " gets hungry");
                weiter.tell(new Messages.Hungry(), getSelf());
            } else if (message instanceof Messages.Eat) {
                System.out.println(name + " eating");
                Thread.sleep(EAT_TIME);
                System.out.println(name + " fed up");
                weiter.tell(new Messages.FinishEat(), getSelf());
                value = this.eatingSeq.get(name);
                this.eatingSeq.put(name, value + 1);
            }
        } else {
            this.end = Instant.now();
            // termina
            System.out.println("shutdown!");
            // imprime tempo
            System.out.println("total time: " + Duration.between(start,
end));
            System.exit(0);
        }

    }

    public static void main(String[] args) throws InterruptedException {
        final ActorSystem system = ActorSystem.create();
        final int FORKS = 5;

        ActorRef waiter = system.actorOf(Waiter.mkProps(FORKS));

        ActorRef Aristotle  =
system.actorOf(Philosopher.mkProps("Aristotle", waiter));
        ActorRef Platon     =
system.actorOf(Philosopher.mkProps("Platon", waiter));
        ActorRef Socrates   =
system.actorOf(Philosopher.mkProps("Socrates", waiter));
        ActorRef Pifagor    =
system.actorOf(Philosopher.mkProps("Pifagor", waiter));
        ActorRef Ptolemy    =
system.actorOf(Philosopher.mkProps("Ptolemy", waiter));
    }
}
```

Classe Messages

```
package DinningPhilosophers;


public class Messages {
    private Messages() {}
```

```
    public static class Introduce {
        private String mPhilosopher;

        public Introduce(String philosopherName) {
            mPhilosopher = philosopherName;
        }

        public String getPhilosopherName() {
            return mPhilosopher;
        }
    }

    public static class Think {}
    public static class Eat { }
    public static class FinishEat { }
    public static class Hungry { }

}
```

Classe Waiter

```
package DinningPhilosophers;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;

import java.util.ArrayList;
import java.util.Arrays;

public class Waiter extends UntypedActor {

    public static Props mkProps(int forkCount) {
        return Props.create(Waiter.class, forkCount);
    }

    private enum ForkState { FREE, USED }
    private ForkState[] mForks;
    private ArrayList<ActorRef> mPhilosophers;

    private Waiter(int forkCount) {
        mForks = new ForkState[forkCount];
        Arrays.fill(mForks, ForkState.FREE);
        mPhilosophers = new ArrayList<ActorRef>(forkCount);
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof Messages.Introduce) {
            String name = ((Messages.Introduce)
message).getPhilosopherName();
            System.out.println(name + " joined table. Welcome!");
            mPhilosophers.add(getSender());
```

```
            getSender().tell(new Messages.Think(), getSelf());

        }  else if (message instanceof Messages.Hungry) {
            int seat = mPhilosophers.indexOf(getSender());
            if (seat == -1) {
                System.out.println("I don`t know this philosopher");
            } else {
                int leftFork = seat;
                int rightFork = (seat + 1) % mForks.length;
                if (mForks[leftFork].equals(ForkState.FREE) &&
mForks[rightFork].equals(ForkState.FREE)) {
                    mForks[leftFork] = ForkState.USED;
                    mForks[rightFork] = ForkState.USED;
                    getSender().tell(new Messages.Eat(), getSelf());
                } else {
                    getSender().tell(new Messages.Think(), getSelf());
                }
            }
        } else if (message instanceof Messages.FinishEat) {
            int seat = mPhilosophers.indexOf(getSender());
            int leftFork = seat;
            int rightFork = (seat + 1) % mForks.length;
            mForks[leftFork] = ForkState.FREE;
            mForks[rightFork] = ForkState.FREE;
            getSender().tell(new Messages.Think(), getSelf());
        }

    }
}
```

CÓDIGO SOLUÇÃO CHANDY-MISRA

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <string.h>

#define SLEEP_MAX 10000001
unsigned int N, S, T;
time_t start, end;
double diff;

//mutex array associated with the forks
pthread_mutex_t *forkMutex;

//char array to store the philosophers' status
//(Thinking/Eating/Waiting/Terminated)
char **state;
char **lastState;
int *eatingTime;
```

```
int *isTerminated;

//int array to store IDs of the philosophers
//who are currently holding a fork
int *heldBy;

//int array to store IDs of the philosophers
//who were the last one to use a fork
int *prevUsedBy;

//int variable to check if watcher should continue running
int watcherRun = 1;

void *philosopher(void *tid) {
    int thisID = *(int*)tid;
    int firstFork = thisID;
    int secondFork = thisID+1;

    //philosopher N-1 should pick up fork 0 before fork N-1
    if (thisID == N-1) {
        firstFork = 0;
        secondFork = N-1;
    }

    //quit while loop if the ID of this philosopher has been set to -1
    while (*(int*)tid != -1) {
        state[thisID] = "Thinking";
        usleep(1000000);

        state[thisID] = "Waiting";

        //acquire the first fork
        pthread_mutex_lock(&forkMutex[firstFork]);
        heldBy[firstFork] = thisID;

        //if the second fork is not yet available
        //and the fist fork was last used by this philosopher
        //release the first fork while waiting for the second fork
        while (pthread_mutex_trylock(&forkMutex[secondFork]))
            if (prevUsedBy[firstFork] == thisID) {
                heldBy[firstFork] = -1;
                prevUsedBy[firstFork] = -1;
                pthread_mutex_unlock(&forkMutex[firstFork]);
            }
        heldBy[secondFork] = thisID;

        //when the second fork is available and successfully acquired
        //check if the first fork is still held by this philosopher
        //if not, re-acquire the first fork
        if (heldBy[firstFork] != thisID) {
            pthread_mutex_lock(&forkMutex[firstFork]);
            heldBy[firstFork] = thisID;
        }

        state[thisID] = "Eating";
```

```
        eatingTime[thisID]++;

        usleep(1000000);

        //release both forks
        heldBy[firstFork] = -1;
        prevUsedBy[firstFork] = thisID;
        pthread_mutex_unlock(&forkMutex[secondFork]);
        heldBy[secondFork] = -1;
        prevUsedBy[secondFork]= thisID;
        pthread_mutex_unlock(&forkMutex[firstFork]);

    }

    state[thisID] = "Terminated";
    pthread_exit(NULL);
}

void *watcher() {
    int i, th, wa, ea, use, avail;
    while (watcherRun) {
        //wait for 0.5 seconds
        usleep(500000);

        //start printing
        printf("Philo   State               Fork    Held by\n");
        th = 0; wa = 0; ea = 0; use = 0; avail = 0;
        for (i=0; i<N; i++) {
            if (heldBy[i] != -1) {
                printf("[%2d]:   %-20s[%2d]:
%d\n",i,state[i],i,heldBy[i]);
                use++;
            } else {
                printf("[%2d]:   %-20s[%2d]:   Free\n",i,state[i],i);
                avail++;
            }

            if (strcmp(state[i],"Thinking") == 0)
                th++;
            else if (strcmp(state[i],"Waiting") == 0)
                wa++;
            else if (strcmp(state[i],"Eating") == 0)
                ea++;

        }
        printf("th=%2d Wa=%2d Ea=%2d           Use=%2d
Avail=%2d\n\n\n",th,wa,ea,use,avail);
    }

    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    N = atoi(argv[1]);
    S = atoi(argv[2]);
```

```
    T = atoi(argv[3]);

    time (&start);

    int i;
    int philID[N];
    pthread_t threads[N+1];
    forkMutex = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t)*N);
    state = (char**)malloc(sizeof(char*)*N);
    lastState = (char**)malloc(sizeof(char*)*N);
    eatingTime = (int*)malloc(sizeof(int*)*N);
    isTerminated = (int*)malloc(sizeof(int*)*N);
    heldBy = (int*)malloc(sizeof(int)*N);
    prevUsedBy = (int*)malloc(sizeof(int)*N);

    //initialize all arrays
    for (i=0; i<N; i++) {
        philID[i] = i;
        pthread_mutex_init(&forkMutex[i],NULL);
        state[i] = "Waiting";
        lastState[i] = "Waiting";
        heldBy[i] = -1;
        prevUsedBy[i] = i;
        eatingTime[i] = 0;
        isTerminated[i] = 0;
    }

    //create and run threads
    srandom(S);
    for (i=0; i<N; i++) {
        pthread_create(&threads[i],NULL,philosopher,(void*)&philID[i]);
    }
    pthread_create(&threads[N],NULL,watcher,NULL);

    sleep(T);

    //cancel and wait for all philosohper threads to terminate
    for (i=0; i<N; i++)
        philID[i] = -1; //mark that this thread should terminate after
                //completing the current thinking-eating cycle

    for (i=0; i<N; i++)
        pthread_join(threads[i],NULL);

    //cancel and wait for watcher thread to terminate
    watcherRun = 0;
    pthread_join(threads[N],NULL);

    //destroy variables and free up memory
    for (i=0; i<N; i++)
        pthread_mutex_destroy(&forkMutex[i]);

    free(forkMutex);
    free(state);
    free(heldBy);
```

```
    free(prevUsedBy);

    time (&end);
    diff = difftime (end, start);

    for (i = 0; i < N; i++) {
        printf("philosopher %d has eaten %d times.\n", i, eatingTime[i]);
    }

    printf("Took %.2lf seconds to run.\n", diff);

    pthread_exit(NULL);
}
```