TPC1: Análise comparativa de soluções para o problema do Jantar dos Filósofos

Fundamentos de Programação Paralela e Distribuída – Prof. Cesar A. F. De Rose

Lorenzo Manica <*lorenzo.manica@edu.pucrs.br*>
Giuvan Reginatto <*giuvan.reginatto@edu.pucrs.br*>

O Jantar dos Filósofos é um problema conhecido de concorrência e sincronização de processos. Foi proposto em 1965 por Dijkstra [1] , tendo sido apresentadas diversas variações e soluções distintas da sua versão original. Este estudo apresenta soluções para o problema do Jantar dos Filósofos, descrevendo os mecanismos que cada uma utiliza para resolver a concorrência entre os processos e as formas de lidar com o *deadlock* e *starvation*, além de analisar os níveis de concorrência obtidos nas soluções quando comparadas a uma solução estritamente sequencial. Ao final, são mostrados os códigos-fontes e os traços de execução de cada solução.

A solução 1 foi proposta por Tanenbaum [2] como uma variação da solução original de Dijkstra. Esta solução baseia-se em semáforos numéricos, onde este adquire um valor inteiro indicando "livre" com valores positivos e "bloqueado" com o valor zero. As chamadas de liberação e aquisição incrementam e decrementam o valor, respectivamente. O semáforo binário opera apenas com os valores 1 e 0 (aberto e fechado, respectivamente). O ponto central desta solução está em realizar o teste da seguinte propriedade do problema: um filósofo pode alimentar-se apenas quando nenhum dos seus vizinhos estiver se alimentando. Isto se deve ao fato da necessidade dos filósofos em compartilhar os garfos com os seus vizinhos. Desta maneira, o filósofo adquire um comportamento de certa forma reservado, uma vez que ele só acessa a mesa quando não há vizinhos, porém colaborativo, dado que após alimentar-se, ele indica aos seus vizinhos que está se retirando. Para manter o controle dos estados dos filósofos, o algoritmo faz uso de uma estrutura de dados que armazena o estado do filósofo, "comendo", "pensando" ou "faminto" além de um conjunto de semáforos, um para cada filósofo, que indicam quando a mesa está livre para um dado filósofo alimentar-se. Para manter uma ordem parcial entre os filósofos, o algoritmo implementa uma seção crítica no momento que o filósofo verifica a condição da mesa e dos seus vizinhos, utilizando para isso um bloqueio (binário) compartilhado entre todos os processos. Este conjunto de medidas faz com que sejam evitados tanto o *deadlock* quanto o *starvation*.

O algoritmo 2 foi proposto pelo site Rosetta Code [4] e implementa uma versão mais simples e direta do problema. Ela se utiliza de um par de canais por filósofo para estabelecer a ligação entre vizinhos e permitir a passagem dos garfos entre dois filósofos. Os canais em Go são canais bufferizados com 1 byte. Se há um byte no canal, o garfo está liberado para uso, se não, o garfo está sendo usado pelo vizinho. O filósofo é modelado por processos concorrentes em threads. O processo do filósofo aguarda até estar no estado "faminto" e então fica aguardando, em ordem, a entrada do garfo no canal do seu lado preferencial e depois no lado oposto. Após a sua utilização, o filósofo retorna os garfos em ordem, primeiramente no canal do seu lado preferencial e depois no lado oposto. A fim de evitar a ciclicidade do grafo, e com isso o *deadlock*, um dos filósofos é tornado "canhoto", assumindo a preferência pelo lado oposto aos demais. Esta solução evita o *starvation* pois o processo do filósofo deliberadamente envia os garfos para os filósofos adjacentes em tempo finito, não sendo possível para um processo tomar posse de nenhum garfo indefinidamente.

O algoritmo 3 foi proposto por Chandy e Misra [3] como uma solução alternativa do problema, fazendo uso de formalismos para modelar processos concorrentes orientados a eventos, com recursão e *interleaving*, além de usar canais para o envio de sinais de sincronização entre os processos, utilizando um canal síncrono (FIFO) por filósofo. Esta solução está centrada na troca de mensagens entre os filósofos e no grafo de dependências que é descrito pelo estado dos garfos na mesa. Ao possuir garfos, o filósofo estabelece uma relação de precedência sobre os demais, fazendo com que tenha preferência na alimentação. Para isso os filósofos usam *tokens* para solicitarem garfos aos seus vizinhos, que decidem quando irão conceder o garfo. Os garfos possuem a propriedade de estarem "limpos" ou "sujos", que indicam a maneira pela qual o filósofo aceita transferir o garfo para o vizinho solicitante. Ao adquirir ambos os garfos, o filósofo está apto a alimentar-se, e essa condição, desde que respeitada desde a inicialização do cenário, garante a exclusão mútua entre os processos (e a quebra da ciclicidade do grafo), evitando *deadlock*. Além da "higiene", a solução descreve ainda o comportamento justo do filósofo em relação aos seus pares através do protocolo de alternância do *token* de solicitação, que objetiva evitar o *starvation.* Ao conceder o garfo ao vizinho, o filósofo retém o *token* de solicitação, para que ele seja o próximo a obter o garfo novamente.

A fim de analisar a performance dos algoritmos descritos acima, foi implementada uma versão sequencial, usando uma solução "ingênua" do problema, onde os processos dos filósofos são sincronizados de maneira estática e pré-determinada, a fim de estabelecer uma base de comparação entre as soluções. Os algoritmos foram configurados para executarem um número determinado de rodadas de refeições (r=10), com os tempos de permanência de 100ms no estado "comendo" e 500ms no estado "pensando", e foi introduzido um processo observador informando em intervalos de 50ms a evolução de cada filósofo na sua alimentação.

	Base (seq)	Tanenbaum	Canhoto	Chandy/ Misra
Tempo (s)	30,6	6,2	6,4	7,4
Speedup (x)	-	4,9	4,8	4,1

Tabela 1. Tempo de execução.

O tempo de execução da versão sequencial pode ser calculado a priori, considerando o tempo total do processo, multiplicado pelo número de filósofos e de rodadas: 10x5(100+500). O algoritmo de Tanenbaum, implementado em C, foi o mais rápido (6,2s), embora o algoritmo do Canhoto, implementado em Go, não tenha ficado tão distante

(6,4s). O algoritmo mais lento foi o Chandy/Misra (7,4s), implementado em Orc, um dialeto do CSP baseado em Scala/Java que executa dentro da VM Java, criada pelo próprio Misra e disponibilizado pela Universidade de Austin, Texas.

Após a execução das diferentes soluções, verificou-se que tanto a solução de Tanenbaum como a solução do Canhoto atingem o nível máximo de concorrência possível para um problema de n-filósofos. (No caso testado, de 5 filósofos, a concorrência máxima é 2). Embora o algoritmo de Chandy/Misra possa atingir o grau de paralelismo máximo teoricamente, não foi possível observar essa propriedade, conforme evidenciado na Figura 3.

Algoritmo Sequencial

```
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

Fig.1 Trace do algoritmo sequencial

Algoritmo Tanenbaum

```
1 0 1 0 0
1 0 1 0 0
1 1 1 0 1
1 1 1 0 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
2 1 2 1 1
2 1 2 1 1
2 2 2 1 2
2 2 2 1 2
```

Fig.2 Trace do algoritmo Tanenbaum

Algoritmo Canhoto

```
1 0 1 0 0
1 0 1 0 0
1 1 1 0 1
1 1 1 0 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
2 1 2 1 1
2 1 2 1 1
2 2 2 1 2
2 2 2 1 2
```

Fig.3 Trace do algoritmo Canhoto

Algoritmo Chandy-Misra

```
0 0 0 0 1
0 0 0 0 1
0 0 0 1 1
0 0 0 1 1
0 0 0 1 1
0 0 1 1 1
0 0 1 1 1
0 0 1 1 1
0 1 1 1 1
0 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 2
1 1 1 1 2
1 1 1 2 2
1 1 1 2 2
1 1 1 2 2
1 1 2 2 2
1 1 2 2 2
1 2 2 2 2
1 2 2 2 2
1 2 2 2 2
1 2 2 2 2
```

Fig.4 Trace do algoritmo Chandy-Misra

Referências

[1] DIJKSTRA, E.W. *Hierarchical ordering of sequential processes*. Dijkstra Archive. Center for American History, University of Texas at Austin. Disponível em *<http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>*

[2] TANENBAUM, A; BOS, H. *Modern Operating Systems*. 4 ed. Pearson, 2015

[3] CHANDY, K; MISRA, J. *The Drinking Philosophers Problem*. ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, Pages 632-646. Disponível em <https://www.cs.utexas.edu/users/misra/scannedPdf.dir/DrinkingPhil.pdf>

[4] *Dining Philosophers*. RosettaCode. Disponível em: <https://rosettacode.org/wiki/Dining_philosophers#Go> Acessado em 17 Abr. 2022

Anexo 1: Algoritmo sequencial - “ingênuo” (Java)

```
public class DiningPhilosophers implements Runnable {

    public static final int N = 5;
    public static final int ROUNDS = 10;
    public static final int PHIL_THINKING = 0;
    public static final int PHIL_HUNGRY = 1;
    public static final int PHIL_EATING = 2;
    public static final int FORK_FREE = -1;
    public static final long THINK_TIME = 500;
    public static final long EAT_TIME = 100;
    public static final int[] iterations = new int[N];

    int[] philosopher;
    Semaphore[] fork;
    int[] forkMap;

    public DiningPhilosophers() {
        philosopher = new int[N];
        forkMap = new int[N];
        fork = new Semaphore[N];
        // iterations = new int[N];

        for (int i = 0; i < N; i++) {
            fork[i] = new Semaphore(1);
            forkMap[i] = FORK_FREE;
        }
    }

    @Override
    public void run() {
        try {
            for (int n = 0; n < ROUNDS; n++) {
                for (int i = 0; i < N; i++) {
                    // printStates();
                    think(i);
                    philosopher[i] = PHIL_HUNGRY;
                    takeForks(i);
                    eat(i);
                    leaveForks(i);
                    philosopher[i] = PHIL_THINKING;
                }
            }
        } catch (Exception e) {
            System.out.println("End of program");
        }
    }

    private void printStates() {
        StringBuilder sb = new StringBuilder();
        for (int k = 0; k < N; k++) {
            sb.append(iterations[k]).append(" ");
        }
        System.out.println(sb);
    }

    private void think(int i) throws InterruptedException {
        philosopher[i] = PHIL_THINKING;
        Thread.sleep(THINK_TIME);
    }


    private void takeForks(int i) throws InterruptedException {
        fork[i].acquire();
        forkMap[i] = i;
        if (fork[rightFork(i)].tryAcquire()) {
            forkMap[rightFork(i)] = i;
        } else {
            fork[i].release();
            forkMap[i] = FORK_FREE;
        }
    }

    private int rightFork(int i) {
        return (i + 1) % N;
    }

    private void eat(int i) throws InterruptedException {
        if (forkMap[i] == i && forkMap[rightFork(i)] == i) {
            philosopher[i] = PHIL_EATING;
            Thread.sleep(EAT_TIME);
            iterations[i]++;
        }
    }

    private void leaveForks(int i) {
        forkMap[rightFork(i)] = FORK_FREE;
        fork[rightFork(i)].release();
        forkMap[i] = FORK_FREE;
        fork[i].release();
    }
}
```

Anexo 2: Algoritmo Tanenbaum (C)

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define ROUNDS 10

int rounds[N];
bool finish = false;
int state[N];

sem_t mutex;
sem_t s[N];

typedef struct {
    int index;
} thread_arg, *ptr_thread_arg;

void think() {
    usleep(500000);
}

void eat() {
    usleep(100000);
}

void test(int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        sem_post(&s[i]);
    }
}

void take_forks(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    test(i);
    sem_post(&mutex);
    sem_wait(&s[i]);
}

void put_forks(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void philosopher(int i) {
    while (rounds[i]<ROUNDS) {
        think();
        take_forks(i);
        eat();
        rounds[i]++;
        put_forks(i);
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    pthread_t thread[N+1];
    thread_arg args[N+1];

    int i;
    for (i=0; i<N+1; i++) {
        state[i] = THINKING;
        args[i].index = i;
        if (i==N) pthread_create(&(thread[N]), NULL, watcher_task, &(args[i]));
        else pthread_create(&(thread[i]), NULL, philosopher_task, &(args[i]));
    }
    for (i=0; i<N+1; i++) {
        pthread_join(thread[i], NULL);
    }
    return 0;
}
```

Anexo 3: Algoritmo “Canhoto” (Go)

```
#package main

import (
	"log"
	"os"
	"time"
)

var ph = []string{"Aristotle", "Kant", "Spinoza", "Marx", "Russell"}
var it = []int{0, 0, 0, 0, 0}

const hunger = 10
const think = 500 * time.Millisecond
const eat = 100 * time.Millisecond

var fmt = log.New(os.Stdout, "", 0)

var done = make(chan bool)

type fork byte

func philosopher(id int, phName string, dominantHand, otherHand chan fork, done chan bool) {
	for h := 0; h < hunger; h++ {
		time.Sleep(think)
		<-dominantHand
		<-otherHand
		time.Sleep(eat)
		it[id] = it[id] + 1
		dominantHand <- 'f'
		otherHand <- 'f'
	}
	done <- true
}

func main() {
	place0 := make(chan fork, 1)
	place0 <- 'f'
	placeLeft := place0
	for i := 1; i < len(ph); i++ {
		placeRight := make(chan fork, 1)
		placeRight <- 'f'
		go philosopher(i, ph[i], placeLeft, placeRight, done)
		placeLeft = placeRight
	}

	go philosopher(0, ph[0], place0, placeLeft, done)

	for range ph {
		<-done
	}
}
```

Anexo 4: Algoritmo Chandy/Misra (Orc)

```
import class ScalaSet = "scala.collection.mutable.HashSet"
import class System = "java.lang.System"

def Set[A](items :: List[A]) = ScalaSet[A]() >s> joinMap(s.add, items) >> s

type Message = (String, lambda((String, lambda(Top) :: Signal)) :: Signal)
type Xmitter = lambda(Message) :: Signal

val TIME_EATING = 100
val TIME_THINKING = 500

def philosopher(name :: Integer, ctr :: Array, mbox :: Channel[Message], missing :: ScalaSet[Xmitter], iterations :: Integer,
done :: Semaphore) :: Bot =
  val send = mbox.put
  val receive = mbox.get
  val deferred = Channel[Xmitter]()
  val clean = Set[Xmitter]([])

  def sendFork(p :: Xmitter) =
    missing.add(p) >>
    p(("fork", send))

  def requestFork(p :: Xmitter) =
    clean.add(p) >>
    p(("request", send))

  def digesting() :: Bot =
      thinking()
    | Rwait(TIME_THINKING) >>
      send(("rumble", send)) >>
      stop

  def thinking() :: Bot =
    def on(("rumble", _) :: Message) =
      map(requestFork, missing.toList() :!: List[Xmitter]) >>
      hungry()
    def on(("request", p)) =
      sendFork(p :!: Xmitter) >>
      thinking()
    on(receive())

  def hungry() :: Bot =
    def on(("fork", p) :: Message) =
      missing.remove(p :!: Xmitter) >>
      (
        if missing.isEmpty() then
          eating()
        else hungry()
      )
    def on(("request", p)) =
      if clean.contains(p :!: Xmitter) then
        deferred.put(p :!: Xmitter) >>
        hungry()
      else
        sendFork(p :!: Xmitter) >>
        requestFork(p :!: Xmitter) >>
        hungry()
    on(receive())

  def eating() :: Bot =
    clean.clear() >>
    Rwait(TIME_EATING) >>
    map(sendFork, deferred.getAll()) >>
    ctr(name)? >x> ctr(name):=x+1 >>
    (if (ctr(name)? <: iterations + 1)
      then digesting()
      else done.release() >> stop)

  digesting()

def philosophers(n :: Integer) =
  val cs = Table(n, lambda (_) = Channel())
  val c = Channel()
  val done = Semaphore(0)
  val ctr = fillArray(Array(n), lambda(_) = 0)
  val t = System.currentTimeMillis()

  philosopher(0, ctr, cs(0), Set[Xmitter]([]), 10, done)
  | for(1, n-1) >i>
    philosopher(i, ctr, cs(i), Set[Xmitter]([cs(i-1).put]), 10, done)
  | philosopher(n-1, ctr, cs(n-1), Set[Xmitter]([cs(n-2).put, cs(0).put]), 10, done)
  | Println(t) >> watcher()

philosophers(5)
```