UNIVERSIDADE DO ESTADO DO AMAZONAS

SISTEMAS DE INFORMAÇÃO

BENJAMIM BORGES
JACKSON KELVIN
RAÍ SOLEDADE
RICHARDSON SOUZA

PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO

JANTAR DOS FILÓSOFOS, LEITORES X ESCRITORES E PRODUTORES X CONSUMIDORES

Manaus
2016

BENJAMIM BORGES
JACKSON KELVIN
RAÍ SOLEDADE
RICHARDSON SOUZA

PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO

JANTAR DOS FILÓSOFOS, LEITORES X ESCRITORES, PRODUTORES X CONSUMIDORES

Trabalho apresentado ao Curso de Sistema de informação da UEA - Universidade do Estado do Amazonas, para a disciplina Sistemas Operacionais.

Prof. Carlos Mauricio Serodio Figueiredo

Manaus
2016

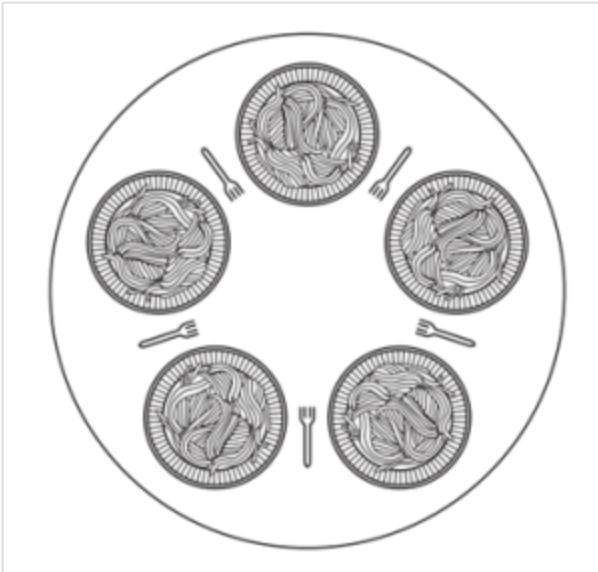
Problemas Clássicos de Sincronização

1. Jantar dos Filósofos

a. ausência de deadlock;
b. ausência de starvation;
c. alto grau de paralelismo - cinco filósofos com fome, dois podem comer;

O problema do jantar dos filósofos é um exemplo ilustrativo de um problema comum de programação concorrente. É mais um problema clássico de sincronização.

O problema dos filósofos jantando foi introduzido e solucionado por **Edsger Dijkstra** em 1965. Neste problema, 5 filósofos estão sentados ao redor de uma mesa redonda e cada filósofo tem um prato de espaguete. Entre cada prato há um garfo e o espaguete está tão escorregadio que o filósofo precisa de 2 garfos para comê-lo.



A vida do filósofo consiste em períodos alternados de comer e de pensar. Quando o filósofo fica com fome, ele tenta pegar o garfo da direita e o da esquerda, um de cada vez, em qualquer ordem.

Uma boa solução para o problema dos filósofos jantando é não permitir que ocorra **deadlock**, ou seja, quando todos os filósofos param esperando que alguém libere o garfo. Outra situação que devemos evitar é **starvation** (inanição)

onde todos os filósofos tentam continuar a executar a tentativa de pegar o garfo, mas não há nenhum progresso, ou seja, todos passam fome. Além disso, se temos 5 filósofos com fome e temos 5 garfos seria interessante deixar 2 filósofos comer obtendo assim um **alto grau de paralelismo**.

Exemplo de código em **Python**:

```
#!/usr/bin/env python
import thread
import time, random
import threading

garfo = list()
for i in range(5):
   garfo.append(threading.Semaphore(1))

def filosofo(f):
   f = int(f)
   while True:
      # garfo da esquerda
      garfo[f].acquire()
      # garfo da direita
      garfo[(f + 1) % 5].acquire()
      print "Filosofo %i comendo..." %f
      time.sleep(random.randint(1, 5))
      garfo[f].release()
      garfo[(f + 1) % 5].release()
      print "Filosofo %i pensando..." %f
      time.sleep(random.randint(1, 10))

for i in range(5):
   print "Filosofo", i
   thread.start_new_thread(filosofo, tuple([i]))

while 1: pass
```

2. Problema dos Leitores e Escritores

Esse problema, proposto por **Courtois** e outros em 1971, é utilizado para modelar o acesso concorrente de processos leitores e escritores a uma base de dados. O acesso pode ser feito por vários leitores simultaneamente. Se um escritor estiver executando uma alteração, nenhum outro leitor ou escritor poderá acessar a base de dados.

a. **Leitores**: processos, os quais não são requeridos excluir uns aos outros (entre eles).

b. **Escritores**: Processos, os quais são requeridos excluir todos os outros processos, leitores e escritores.

Esse problema é uma abstração do acesso à uma base de dados, onde não existe perigo em termos de diversos processos lendo concorrentemente, mas escrevendo ou ou mudando os dados deve ser feito sobre exclusão mútua para garantir consistência.

Exemplo de código em **Python**:

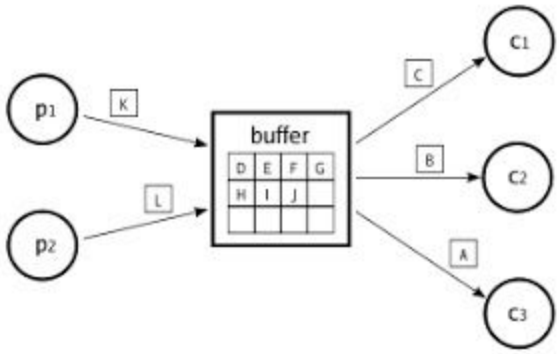
```
#!/usr/bin/env python

import thread
import time, random
import threading

class BancoDados:
   contLeitor = 0
   mutex      = threading.Semaphore(1)
   bd         = threading.Semaphore(1)

   def acquireReadLock(self):
      global contLeitor
      self.mutex.acquire()
      self.contLeitor += 1

      # E o primeiro leitor?
      if self.contLeitor == 1:
         self.bd.acquire()

      self.mutex.release()

   def releaseReadLock(self):
      global contLeitor
      self.mutex.acquire()
      self.contLeitor -= 1

      # E o ultimo leitor?
      if self.contLeitor == 0:
         self.bd.release()

      self.mutex.release()

   def acquireWriteLock(self):
      self.bd.acquire()

   def releaseWriteLock(self):
      self.bd.release()

bd = BancoDados()

def escritor(e):
   while True:
```

```
        time.sleep(random.randint(1, 5))
        bd.acquireWriteLock()
        print "Escritor %i - escrevendo..." %e
        time.sleep(random.randint(1, 5))
        bd.releaseWriteLock()
        print "Escritor %i - parou de escrever." %e

def leitor(l):
    while True:
        time.sleep(random.randint(1, 10))
        bd.acquireReadLock()
        print "Leitor %i - lendo..." %l
        time.sleep(random.randint(1, 5))
        bd.releaseReadLock()
        print "Leitor %i - parou de ler." %l

for i in range(2):
    print "Escritor", i
    thread.start_new_thread(escritor, tuple([i]))
for i in range(3):
    print "Leitor", i
    thread.start_new_thread(leitor, tuple([i]))

while 1: pass
```

3. Problema Produtores e Consumidores

a. Produtor insere em posição: Ainda não consumida;
b. O consumidor remove de posição: Já foi consumida;;
c. Espera ociosa x escalonamento de processo x uso cpu;

O problema do Produtor e Consumidor (também conhecido como o problema do buffer limitado), consiste em um conjunto de processos que compartilham um mesmo buffer. Os processos chamados produtores põem informação no buffer. Os processos chamados consumidores retiram informação deste buffer.



Esse é um problema clássico em sistemas operacionais, que busca exemplificar de forma clara, situações de impasses que ocorrem no gerenciamento de processos de um sistema operacional. Como sabemos, precisamos nos

preocupar com acessos ilegais a certos recursos que são compartilhados entre os processos, e manter sincronismo entre os mesmos.

Exemplo de código em **Python**:

```
#!/usr/bin/en python
import thread
import time, random
import threading

class BufferLimitado:
   TAM_BUFFER = 5
   mutex  = threading.Semaphore(1)
   empty  = threading.Semaphore(TAM_BUFFER)
   full   = threading.Semaphore(0)
   buffer = range(TAM_BUFFER)
   cheio  = 0
   livre  = 0

   def insert(self, item):
      self.empty.acquire()
      self.mutex.acquire()
      self.buffer[self.livre] = item
      self.livre = (self.livre + 1) % self.TAM_BUFFER
      self.mutex.release()
      self.full.release()

   def remove(self):
      self.full.acquire()
      self.mutex.acquire()
      item = self.buffer[self.cheio]
      self.cheio = (self.cheio + 1) % self.TAM_BUFFER
      self.mutex.release()
      self.empty.release()
      return item

b = BufferLimitado()

def produtor():
   while True:
      time.sleep(random.randint(1, 10) / 100.0)
      item = time.ctime()
      b.insert(item)
      print "Produtor produziu:", item, b.livre, b.cheio

def consumidor():
   while True:
      time.sleep(random.randint(1, 10) / 100.0)
      item = b.remove()
      print "Consumidor consumiu:", item, b.livre, b.cheio

thread.start_new_thread(produtor, ())
thread.start_new_thread(consumidor, ())

while 1: pass
```

4. Outros exemplos de códigos

a. Jantar dos Filósofos em **C**:

```c
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <semaphore.h>
#include <math.h>
#define N 5
#define LEFT (i+4)%N
#define RIGHT (i+1)%N
#define PENSANDO 0
#define FAMINTO 1
#define COMENDO 2

int state[N];
sem_t mutex;
sem_t s[N];
typedef char* string;
string  nomes[5]  =  {"Epicuro  (1)","Wittgeinstein  (2)","Schopenhauer  (3)","Godel
(4)","Kant (5)"};  // Nomes dos filósofos

// Protótipos das funções
void eat(int i);
void think(int i);
void take_forks(int i);
void put_forks(int i);
void test(int i);

void eat(int i)
{
 sleep(i);               // Tempo de comer do filósofo i
}

void think(int i)
{
 sleep(i+1);             // Tempo de pensar do filósofo i
}

// Thread do filósofo
void * philosopher(void *a)
{
   int k = *(int *)a;
   while (1)
    {
    // Comportamento do filósofo
       think(k);
       take_forks(k);
       eat(k);
       put_forks(k);
    }
}
```

```
void take_forks(int i)
{
   sem_wait(&mutex);     // Entra na região critica
   state[i] = FAMINTO;    // Define o estado como com fome
   test(i);              // Verifica se pode pegar os garfos para comer
   sem_post(&mutex);     // Libera a região critica
   sem_wait(&s[i]);      // Se o filosofo não pegou os garfos deve esperar
}

void put_forks(int i)
{
    // Liberar os garfos na mesa
   sem_wait(&mutex);     // Entra na regiao critica
   state[i] =  PENSANDO; // Define o estado como pensando
    test(LEFT);           // Verifica se o filósofo à esquerda pode pegar os garfos
para comer
    test(RIGHT);           // Verifica se o filósofo à direita pode pegar os garfos
para comer
   sem_post(&mutex);     // Sai da regiao critica
}
    // Funções para imprimir os estados
void imprime_estados(int k)
{
   if(state[k] == FAMINTO) printf("FAMINTO\n");
   if(state[k] == COMENDO) printf("COMENDO\n");
   if(state[k] == PENSANDO) printf("PENSANDO\n");
}

void imprime()
{
   printf("Epicuro(1): ");
   imprime_estados(0);
   printf("Wittgeinstein(2): ");
   imprime_estados(1);
   printf("Schopenhauer(3): ");
   imprime_estados(2);
   printf("Godel(4): ");
   imprime_estados(3);
   printf("Kant(5): ");
   imprime_estados(4);
   printf("\n");
}

void test(i)
{
        if(state[i]==FAMINTO && state[LEFT]!=COMENDO && state[RIGHT]!=COMENDO) // Se
está faminto e os vizinhos não estão comendo...
        {
           state[i] = COMENDO;       //... entao come
             imprime();                // imprime estado de todos os filósofos nesse
instante
        sem_post(&s[i]);          // impede-se de ser bloqueado, pois pode pegar os
garfos
        }
}

int main()
{
```

```
    pthread_t filosofos[N];   // Declara os filosofos
    sem_init(&mutex,0,1);     // Inicia o semaforo mutex
    int aux[5];

    int j;
    for(j=0;j<5;j++)
      {
         sem_init(&s[j],0,0);  // Inicia os semaforos de cada filosofo
      }

    for(j=0;j<5;j++)
    {
         aux[j] = j;
             pthread_create(&filosofos[j],NULL,philosopher,(void*)&aux[j]);  // Cria
os filosofos
    }

     pthread_join(filosofos[0],NULL);     // Força a main a esperar até que a thread
filosofos[0] encerre-se (no caso é infinita)

return 0;
}
```

b. Leitores x Escritores em **C**:

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

void le_texto(int);                     // Protótipos das funções
void interpreta_texto(int);
void pensa_novo_texto(int);
void escreve_novo_texto(int);

pthread_mutex_t mutex, caderno;       // Declarações dos semáforos
int rc = 0;   // numero de processos lendo ou querendo ler

void le_texto(int a){
  printf("Leitor %d: Lendo caderno (Regiao critica)\n",a);
  srand(time(NULL));
  sleep(rand()%2 + 1);
}

void interpreta_texto(int a){
  printf("Leitor %d: Interpretando (Regiao não-critica)\n",a);
  srand(time(NULL));
  sleep(rand()%2 + 1);
}
```

```c
void pensa_novo_texto(int a){
  printf("Escritor %d: Pensando em novo texto (Regiao não-critica)\n",a);
  srand(time(NULL));
  sleep(rand()%2 + 1);
}

void escreve_novo_texto(int a){
  printf("Escritor %d: Escrevendo no caderno (Região critica)\n",a);
  srand(time(NULL));
  sleep(rand()%2 + 1);
}

void * reader(void *a)
{
 int k = *(int *)a; // Converte para o int

 while(1)
  {
    pthread_mutex_lock(&mutex); // Obtem acesso à região critica rc
    rc++;    // Mais um leitor
     if(rc==1) pthread_mutex_lock(&caderno);  // Obtém acesso exclusivo aos leitores
na região critica caderno
    pthread_mutex_unlock(&mutex);        // Sai da região critica rc
    le_texto(k);      // acessa dados em região critica

    // Leitor quer sair
    pthread_mutex_lock(&mutex);            // Obtem acesso a regiao critica rc
    rc--;                                                  // Um leitor a menos
      if(rc==0) pthread_mutex_unlock(&caderno);  // Abre o acesso à região critica
caderno
    pthread_mutex_unlock(&mutex);                 // Sai da região critica rc
    interpreta_texto(k);      // acesso em região não-critica
  }
}

void * writer(void *a)
{
int k = *(int *)a;
 while(1)
  {
    pensa_novo_texto(k);    // produz texto
    pthread_mutex_lock(&caderno);  // obtém acesso exclusivo ao caderno
    escreve_novo_texto(k);           // escreve no caderno
    pthread_mutex_unlock(&caderno);   // libera o caderno
  }
}

int main(){
   pthread_mutex_init(&mutex,NULL);     // inicia os mutexes
   pthread_mutex_init(&caderno,NULL);
   int leitor[] = {1, 2, 3, 4, 5};
```

```
   int escritor[] = {1, 2, 3};

   pthread_t leitores[5];                // declara as threads
   pthread_t escritores[3];
   int j;
   for(j=0;j<3;j++)
     {
     pthread_create(&escritores[j],NULL,writer,(void*)&escritor[j]);              //
inicia as threads de escritores
     }

   for(j=0;j<5;j++)
     {
     pthread_create(&leitores[j],NULL,reader,(void*)&leitor[j]);                  //
inicia as threads de leitores
     }

    pthread_join(leitores[0],NULL);  // A main espera mulheres[0] terminar antes de
finalizar.

return 0;
}
```

c. Produtores x Consumidores em **Java**:

```
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to solve Producer Consumer problem using wait and notify
 * method in Java. Producer Consumer is also a popular concurrency design pattern.
 *
 * @author Javin Paul
 */
public class ProducerConsumerSolution {

    public static void main(String args[]) {
        Vector sharedQueue = new Vector();
        int size = 4;
        Thread prodThread = new Thread(new Producer(sharedQueue, size), "Produtor");
                Thread  consThread  =  new  Thread(new  Consumer(sharedQueue,  size),
"Consumidor");
        prodThread.start();
        consThread.start();
    }
}

class Producer implements Runnable {

    private final Vector sharedQueue;
```

```
    private final int SIZE;

    public Producer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println("Produzido: " + i);
            try {
                produce(i);
            } catch (InterruptedException ex) {
                  Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null,
ex);
            }

        }
    }

    private void produce(int i) throws InterruptedException {

        //wait if queue is full
        while (sharedQueue.size() == SIZE) {
            synchronized (sharedQueue) {
                                    System.out.println("Lista  esta  cheia  "  +
Thread.currentThread().getName()+"    esta    esperando    ,    tamanho:    "    +
sharedQueue.size());

                sharedQueue.wait();
            }
        }

        //producing element and notify consumers
        synchronized (sharedQueue) {
            sharedQueue.add(i);
            sharedQueue.notifyAll();
        }
    }
}

class Consumer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Consumer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }
```

```
    @Override
    public void run() {
        while (true) {
            try {
                System.out.println("Consumido: " + consume());
                Thread.sleep(50);
            } catch (InterruptedException ex) {
                  Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null,
ex);
            }

        }
    }

    private int consume() throws InterruptedException {
        //wait if queue is empty
        while (sharedQueue.isEmpty()) {
            synchronized (sharedQueue) {
                                    System.out.println("Lista  esta  vazia  "  +
Thread.currentThread().getName()
                                            + " esta esperando , tamanho: " +
sharedQueue.size());

                sharedQueue.wait();
            }
        }

        //Otherwise consume element and notify waiting producer
        synchronized (sharedQueue) {
            sharedQueue.notifyAll();
            return (Integer) sharedQueue.remove(0);
        }
    }
}
```

5. Referências

SAKATA, T. C. Problemas clássicos de sincronização. FACENS - Faculdade de Engenharia Sorocaba. Disponível em: <http://docslide.com.br/documents/problemas-classicos-de-sincronizacao-fecens.html>. Acesso em: 4/2016.

MINICZ, M. Semáforos para cuidar da seção crítica de um programa. Deadlock com semáforos. Problemas clássicos de sincronização. Wiki Python. Disponível em: <http://wiki.python.org.br/SemaforosDeadlock>. Acesso 4/2016.

TONIN, R. C. Problemas Clássicos. Disponível em: <http://escalonamentoprocessos.blogspot.com.br/2010/10/problemas-classicos_127.html>. Acesso em: 4/2016.

ALBIZZATI, A. C. Sistemas operacionais. Relatório CES-33. ITA - Instituto Tecnológico de Aeronáutica. Disponível em: <http://ces33.wikidot.com/relas:alexandre>. Acesso: 04/2016.

PAUL, J. Producer Consumer Problem with Wait and Notify Example. Disponível em: <http://java67.blogspot.com.br/2012/12/producer-consumer-problem-with-wait-and-notify-example.html>. Acesso: 04/2016