Programação Concorrente

Problemas Clássicos de Concorrência

Prof. Rodrigo Campiolo

UTFPR - Universidade Tecnológica Federal do Paraná
DACOM - Departamento de Computação
BCC - Bacharelado Ciência da Computação

6 de maio de 2019

Introdução

- Apresentar problemas clássicos de concorrência.
- Problemas:
  - O problema do Produtor-Consumidor
  - O problema do Leitor-Escritor
  - O problema do Jantar dos Filósofos
  - O problema dos Fumantes de Cigarro

Problema do Produtor-Consumidor

- Há um **buffer compartilhado** entre dois processos chamados **produtor** e **consumidor**.
- O produtor produz e armazena itens no buffer.
- O consumidor retira e consome itens do buffer.
- Os processos devem acessar o buffer de maneira exclusiva.
- Há duas restrições de sincronização:
  - O consumidor não retira itens de um buffer vazio.
  - O produtor não armazena itens em um buffer cheio.

Implementação: Produtor-Consumidor

- Vamos considerar um buffer circular finito.
- Os itens compartilhados serão do tipo **double**.
- Duas threads: uma produtora e outra consumidora.
- Sincronização usando semáforos.
- Um semáforo binário para a região crítica.
- Dois semáforos por contagem:
  - Manter o produtor esperando quando o buffer está cheio.
  - Manter o consumidor esperando quando o buffer está vazio.

Problema Produtor-Consumidor

```
import java.util.concurrent.Semaphore;

class BoundedBuffer {
    private final int SIZE = 8;
    private final double[] buffer = new double[SIZE];
    private int inBuf = 0, outBuf = 0;

    Semaphore mutex = new Semaphore(1);
    Semaphore canRead = new Semaphore(0);
    Semaphore canWrite = new Semaphore(SIZE);

    public void put(double value) {
        try {
            canWrite.acquire();// wait if buffer is full
            mutex.acquire(); // ensures mutual exclusion
            buffer[inBuf] = value; // update the buffer
            inBuf = (inBuf + 1) % SIZE;
        } catch (InterruptedException ie) {
            System.out.println("IE: " + ie.getMessage());
        } finally {
            mutex.release();
            canRead.release();  // notify any waiting consumer
        }
    }
```

```
    public double get() {
        double value = 0;
        try {
            canRead.acquire(); // wait if buffer is empty
            mutex.acquire();  // ensures mutual exclusion
            value = buffer[outBuf]; //read from buffer
            outBuf = (outBuf + 1) % SIZE;
        } catch (InterruptedException ie) {
            System.out.println("IE: " + ie.getMessage());
        } finally {
            mutex.release();
            canWrite.release(); // notify any waiting producer
        }

        return value;
    }
}
```

Problema do Leitor-Escritor

- Um recurso compartilhado acessado por múltiplos processos.
- Há dois tipos de processos denominados **leitor** e **escritor**.
- O leitor faz leituras no recurso.
- O escritor faz escritas no recurso.
- Há duas restrições de sincronização:
  - Ausência de conflitos de leitura-escrita: leitor e escritor não acessam recurso concorrentemente.
  - Ausência de conflitos de escrita-escrita: escritores não acessam o recurso concorrentemente.

Implementação: Leitor-Escritor

- Sincronização usando semáforos.
- Leitores executam **startRead** para início de leitura e **endRead** para fim de leitura.
- Escritores executam **startWrite** para início de escrita e **endWrite** para fim de escrita.
- Dois semáforos binários:
  - Acesso a região crítica.
  - Controlar o acesso de somente um escritor ou somente leitores.

Implementação (*template*): Leitor-Escritor

```
import java.util.concurrent.Semaphore;

class ReaderWriter {
    int numReaders = 0;
    Semaphore mutex = new Semaphore(1);
    Semaphore wlock = new Semaphore(1);

    public void startRead() throws InterruptedException {
    }

    public void endRead() throws InterruptedException {
    }

    public void startWrite() throws InterruptedException {
    }

    public void endWrite() {
    }
}
```

Problema do Jantar dos Filósofos

- Problema proposto por Dijkstra.
- Aborda questões de concorrência e simetria.
- O problema consiste de vários filósofos, sentados ao redor de uma mesa circular, pensando e comendo spaghetti. No entanto, há somente 1 garfo entre cada par de filósofo e, para comer, um filósofo precisa usar dois garfos, um em cada mão.
- Problema: Como coordenar o acesso dos processos (filósofos) aos recursos compartilhados (garfos) para realizar uma operação (comer macarrão)?

Problema do Jantar dos Filósofos

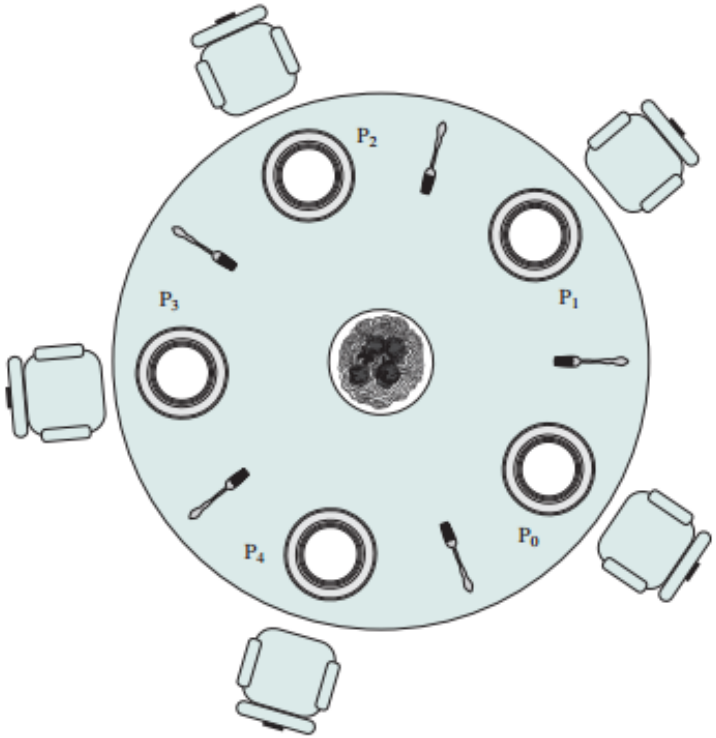

Figura: Jantar dos Filósofos.

(Fonte: https://www.thecrazyprogrammer.com)

Implementação 1: Jantar dos Filósofos

- Filósofo executa ciclos: *pensando*, *faminto*, *comendo*.
- Para comer, um filósofo precisa pegar o garfo da esquerda e direita (recursos i e (i+1)%n).
- Um semáforo binário para cada garfo (recurso).
- Cada garfo é alocado ao obter uma permissão no semáforo.

Implementação (ideia): Jantar dos Filósofos

```
import java.util.concurrent.Semaphore;

class DiningResource {

    int numberResources = 0;   // = numberPhilosophers
    Semaphore[] fork = null;

    public DiningResource(int initResources) {
        numberResources = initResources;
        fork = new Semaphore[numberResources];
        for (int i = 0; i < numberResources; i++) {
            fork[i] = new Semaphore(1);
        }
    }

    public void take(int idPhilosopher) {
        try {
            fork[idPhilosopher].acquire();
            fork[(idPhilosopher + 1) % numberResources].acquire();
        } catch (InterruptedException ex) {}
    }

    public void release(int idPhilosopher) {
        fork[idPhilosopher].release();
        fork[(idPhilosopher + 1) % numberResources].release();
    }
}
```

Implementação 1: Problemas

- Problema de simetria pode gerar deadlock: todos filósofos pegam o garfo esquerdo!
- Soluções:
  1. Introduzir assimetria: obrigar um filósofo a pegar em diferente ordem.
  2. Pegar ambos garfos: atomicamente garantir que um filósofo pega os dois garfos simultaneamente.
  3. Temporização para pegar qualquer garfo: no máximo número de filósofos - 1 devem esperar um tempo.
- Discussões:
  - Em 2, a solução é livre de deadlock, mas não livre de starvation.
  - Como fazer uma solução livre de deadlock e starvation?

Problema dos Fumantes de Cigarro

- *Cigarette smokers problem* proposto por Suhas Patil que afirmava que não poderia ser resolvido por semáforos.
- O problema consiste de uma situação com três fumantes (*smokers*) e um fornecedor (*agent*). Cada fumante faz os próprios cigarros e os fuma. Para fazer os cigarros, cada fumante precisa de três itens: tabaco, papel e fósforos. Cada fumante tem estoque exclusivo e infinito de um desses itens. O fornecedor possui estoque infinito dos três itens. Os fumantes encontram-se inicialmente aguardando itens. O fornecedor seleciona aleatoriamente dois itens e um fumante pega os itens, faz o cigarro e fuma. O ciclo se repete.

Problema dos Fumantes de Cigarro

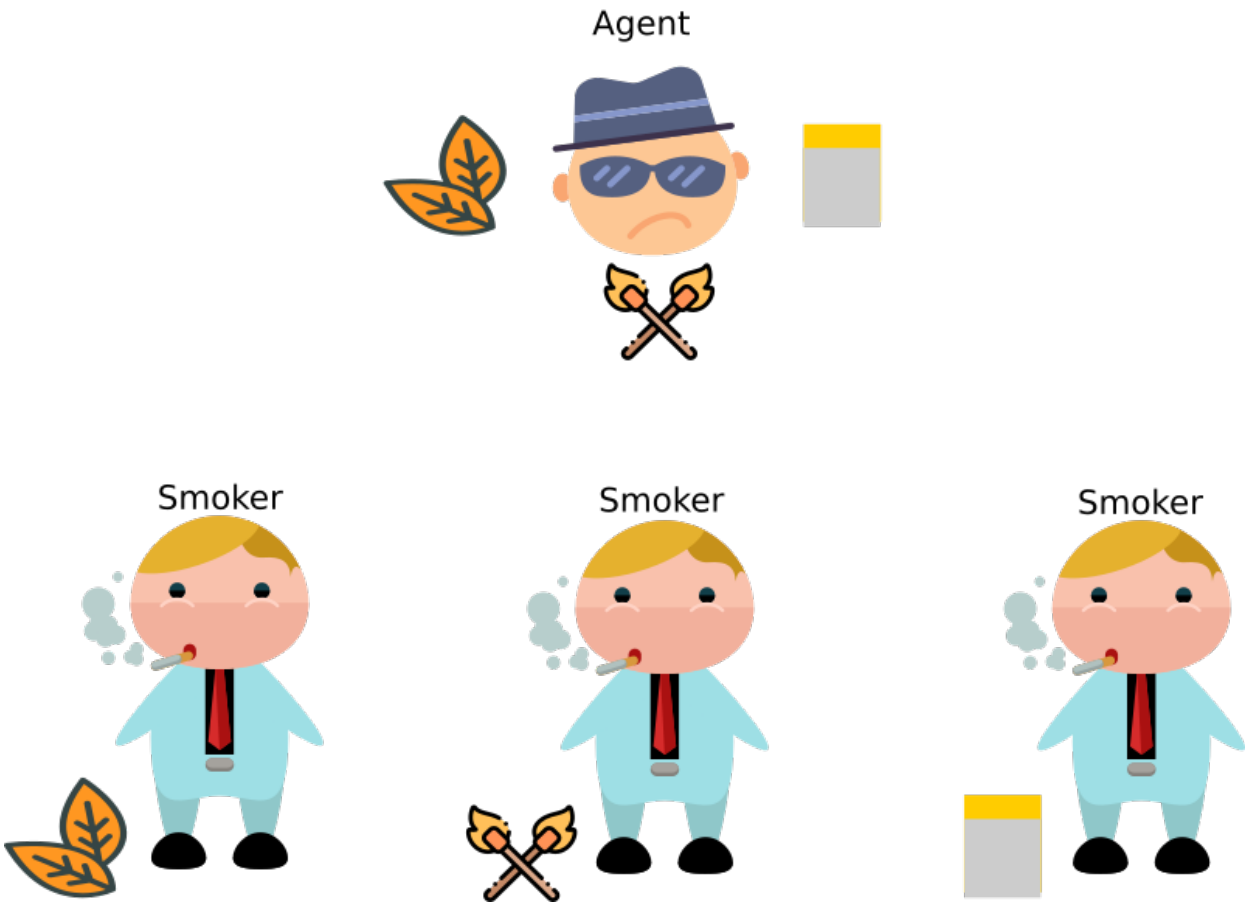

Figura: Fumantes de Cigarro

Restrições

- Não é permitido alterar o código do fornecedor.
- Não é permitido o uso de sentenças condicionais.

Versões

1. O fornecedor disponibiliza apenas um par de itens aos fumantes.
2. O fornecedor pode disponibilizar mais itens aos fumantes.

Implementação da Versão 1 - Solução 1

Semáforos

```
agentSem = Semaphore (1)
tabaco = Semaphore (0)
papel = Semaphore (0)
fosforo = Semaphore (0)
```

Agentes

```
agentSem.acquire ()

// possibilidade 1
tabaco.release ()
papel.release ()

// possibilidade 2
papel.release ()
fosforo.release ()

// possibilidade 3
tabaco.release ()
fosforo.release ()
```

Fumante com fósforo

```
tabaco.acquire ()
papel.acquire ()
agentSem.release ()
```

Fumante com tabaco

```
papel.acquire ()
fosforo.acquire ()
agentSem.release ()
```

Fumante com papel

```
tabaco.acquire ()
fosforo.acquire ()
agentSem.release ()
```

Problema da solução 1: DEADLOCK

- Suponha que o agente produza papel e tabaco.
- O fumante com tabaco pode pegar o papel.
- O fumante com papel pode pegar o tabaco.
- Logo, não há como prosseguir com nenhum fluxo.

Implementação da Versão 1 - Solução 2 (por Parnas)

Variáveis e semáforos adicionais

```
isTabaco = isPapel = isFosforo = False
tabacoSem = Semaphore (0)       // sinaliza fumante com tabaco
papelSem = Semaphore (0)        // sinaliza fumante com papel
fosforoSem = Semaphore (0)      // sinaliza fumante com fosforo
mutex = Semaphore(1)            // seção crítica dos intermediadores
```

Intermediador tabaco

```
// deve-se ter um intermediador
// para cada item
tabaco.acquire ()
mutex.acquire ()
if (isPapel) {
  isPapel = False
  fosforoSem.release ()
} else if (isFosforo) {
  isFosforo = False
  papelSem.release ()
} else {
  isTabaco = True
}
mutex.release ()
```

Fumante com tabaco

```
tabacoSem.acquire ()
makeCigarette ()
agentSem.release ()
smoke ()
```

Outros problemas de sincronização

- O problema do jantar dos selvagens (*dining savages problem*).
- O problema da barbearia (versões: clássica, FIFO e Hilzer).
- O problema do Papai Noel (*Santa Claus problem*).
- Problema H_2O (*Building* H_2O).
- Problema de atravessar o rio (*river crossing problem*).
- Problema da montanha russa (*roller coaster problem*)

Atividades

1. Implementar soluções para o problema dos leitores-escritores que:
   - 1.1 priorize os leitores.
   - 1.2 sem inanição.
   - 1.3 priorize os escritores.
2. Implementar 3 soluções distintas para o jantar dos filósofos que não causem deadlock.
3. Implementar a versão 2 do problema dos fumantes de cigarro.

Referências

- Downey, Allen B. **The Little Book of Semaphores**, version 2.2.1. Second Edition, 2016. Disponível em: `http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf`. Acessado em 01/10/2018.
- Garg, Vijay K. **Concurrent and Distributed Computing in Java.** IEEE Press; Hoboken, N.J.: Wiley-Interscience, c2004. xx, 309, 2004. Disponível em: `http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=5259924`. Acessado em: 01/10/2018.