Controladores do Controle de Concorrência

- **Monitores;**
- **Lock´s;**
- **Semáforos;**
- **Condições de Acesso;**

Introdução ao Controle de Concorrência

- “Se bem utilizado”, o paralelismo resulta em um melhor desempenho dos programas;
- Podem ocorrer problemas (disputa de recursos) no acesso concorrente de dados e recursos;
- Dados podem se tornar inconsistentes ao serem acessados concorrentemente;

Ex.: duas pessoas editando o mesmo arquivo

- Alguns recursos não podem ser compartilhados

Ex.: dois programas usando a impressora

Controle de Concorrência ...

```
//Exemplo: Conta Bancária
public class Conta {
	public double saldo = 0;

	public Conta(double saldo) {
	    this.saldo = saldo;
	    System.out.println("Saldo inicial: R$" + saldo);
	}

	public double getSaldo() {return saldo;}

	public void setSaldo(double saldo) {this.saldo = saldo;}
}
```

Controle de Concorrência ...

```
//Exemplo: Classe Banco
public class Banco {
      public boolean saque(Conta conta, double valor) {
          double saldo = conta.getSaldo();
          if (saldo < valor) {
              System.out.println("Saldo insuficiente para o
  saque.");
              return false;
          }
          double novoSaldo = saldo - valor;
          System.out.println(Thread.currentThread().getName());
          System.out.print(" sacou R$" + valor );
          System.out.print(". Saldo após saque: R$"+novoSaldo);
          conta.setSaldo(novoSaldo);
          return true;
      }
}
```

Controle de Concorrência ...

```
//Exemplo: Clientes do Banco
public class Cliente extends Thread {
        private static Banco banco = new Banco();
        private Conta conta = null;
        private double valor = 100;
        public Cliente(String nome, Conta conta) {
            super(nome);
            this.conta = conta;
        }
        public void run() {
            double total = 0;
            while (banco.saque(conta,valor))total += valor;
            System.out.println(getName());
            System.out.print(" sacou total de R$" + total);
        }
}
```

Controle de Concorrência ...

```
//Exemplo: Família com conta conjunta
public class Familia {
	public static void main (String args[]) {
	    // Cria conta conjunta da família
	    final Conta conta = new Conta(100000);
	    // Cria familiares e lhes informa a conta conjunta
	    Cliente pai = new Cliente("Pai ", conta);
	    Cliente mae = new Cliente("Mãe ", conta);
	    Cliente filho = new Cliente("Filho", conta);
	    // Inicia as threads
	    pai.start();
	    mae.start();
	    filho.start();
	}
}
```

Execução do Programa

Ex.: Movimentação da Conta Conjunta
Conta criada. Saldo inicial: R$100000.0
Pai sacou R$100.0. Saldo após o saque: R$99900.0
Pai sacou R$100.0. Saldo após o saque: R$99800.0
Pai sacou R$100.0. Saldo após o saque: R$99700.0
Pai sacou R$100.0. Saldo após o saque: R$99600.0
Pai sacou R$100.0. Saldo após o saque: R$99500.0
Pai sacou R$100.0. Saldo após o saque: R$99400.0
Pai sacou R$100.0. Saldo após o saque: R$99300.0
Pai sacou R$100.0. Saldo após o saque: R$99200.0
Pai sacou R$100.0. Saldo após o saque: R$99100.0
Mãe sacou R$100.0. Saldo após o saque: R$99100.0
Filho sacou R$100.0. Saldo após o saque: R$99100.0
Pai sacou R$100.0. Saldo após o saque: R$99000.0
Pai sacou R$100.0. Saldo após o saque: R$98900.0
Mãe sacou R$100.0. Saldo após o saque: R$98900.0

Problemas da Concorrência

- O acesso simultâneo/concorrente à conta pode causar inconsistências nos dados;
- Caso ocorra uma troca de contexto durante a execução do método Banco.saque():
- Entre a leitura do saldo e a sua atualização;
- Antes da finalização do saque
- É preciso evitar que outras threads alterem o saldo desta conta enquanto um saque estiver em andamento;

Problemas da Concorrência

- Devido a inconsistências que podem ocorrer com código paralelo, é preciso fazer o controle de concorrência entre processos e threads;
- Mecanismos de Controle de Concorrência;
- Limitam o acesso concorrente a dados e recursos compartilhados;
- Garantem o isolamento entre processos e threads concorrentes;
- Buscam evitar inconsistências nos dados causadas pelo acesso concorrente;

Controle da Concorrência

- **Monitor:** protege trechos de código/métodos que manipulam dados ou recursos compartilhados, impedindo o acesso concorrente;
- **Lock (ou Mutex):** cria uma fila de acesso a um dado/recurso compartilhado, impedindo o acesso concorrente;
- **Semáforo**: limita o número de usuários que acessam simultaneamente um recurso, criando filas de acesso se este número for excedido;

Monitores

- Proposto por Hoare em 1974;
- 1a implementação Pascal Concorrente;
- Usado em várias linguagens, inclusive Java;
- “Monitora” o acesso a dados e recursos compartilhados por processos e threads;
- Encapsula código que faz o acesso a dados e recursos monitorados;
- Executado com exclusão mútua;
- Um acesso por vez;
- Cria uma fila de espera;

Monitores em Java

- Classes e objetos podem ser bloqueados;
- Monitores são definidos usando a palavra chave **synchronized** em blocos ou métodos;
- Métodos e blocos **synchronized** são executados pela JVM com exclusão mútua;
- Threads que tentarem acessar um monitor que esteja em uso entrarão em espera;
- É criada uma fila de espera pelo monitor;
- Threads saem da fila em ordem de prioridade;

Monitores em Java
Exclusão Mútua em Bloco

```
synchronized( Objeto ){

    //Código do Bloco

}
```

- O objeto especificado na abertura do bloco é bloqueado para uso da thread corrente;
- Código fica protegido de acesso concorrente;
- Qualquer outra thread que tentar bloquear o mesmo objeto entrará em uma fila de espera;

Exemplo do Saque Bancário usando Monitores de bloco

```
public class Banco {
    public boolean saque(Conta conta, double valor) {
        synchronized(conta) {
            double saldo = conta.getSaldo();
            if (saldo < valor) {
                System.out.println("Saldo insuficiente para o
  saque.");
                return false;
            }
            double novoSaldo = saldo - valor;
            System.out.println(Thread.currentThread().getName());
            System.out.print(" sacou R$"+valor)
            System.out.print(". Saldo: após saque: R$"+novoSaldo);
            conta.setSaldo(novoSaldo);
            return true;
        }
    }
}
```

Monitores em Java
Exclusão Mútua de Métodos

```
public synchronized void método (int p){
    // código protegido
}
```

- O objeto usado na invocação é bloqueado para uso da thread que invocou o método;
- Se o método for static, a classe é bloqueada;
- Métodos não sincronizados e atributos ainda podem ser acessados;

Monitores em Java
Exclusão Mútua de Método

```
public class Conta {
    ...
    public synchronized double getSaldo() { return this.saldo; }

    public synchronized void setSaldo(double s) { this.saldo = s; }

    public synchronized double debitarValor(double valor) {
        if (this.saldo < valor) {
            System.out.println("Saldo insuficiente para saque.");
            return -1;
        } else {
            this.saldo -= valor;
            return this.saldo;
        }
    }
}
```

Lock´s

- Mecanismo de exclusão mútua;
- Permite somente um acesso por vez;
- Caso o dado/recurso esteja em uso, a thread que tentar bloqueá-lo entra numa fila;
- Principais Métodos:
- **lock()**: primitiva de bloqueio, deve ser chamada antes do acesso ao dado/recurso;
- **unlock()** : primitiva de desbloqueio, usada para liberar o acesso o dado/recurso;

Lock em Java

- tryLock(): bloqueia e retorna true se o lock estiver disponível, caso contrário retorna false;
- getHoldCount(): retorna número de threads que tentaram obter o lock e não o liberaram;
- isHeldByCurrentThread(): retorna true se a thread que fez a chamada obteve o bloqueio;
- isLocked(): indica se o lock está bloqueado;
- getQueueLength(): retorna o número de threads que aguardam pela liberação do lock;

Lock em Java

- Classe ReentrantLock:
  - Implementa mecanismo de bloqueio exclusivo;
  - Por default a retirada de threads da fila não é ordenada, ou seja, não há garantias de quem irá adquirir o lock quando este for liberado;
  - O construtor ReentrantLock(true) cria um lock com ordenação FIFO da fila, o que torna o acesso significativamente mais lento;

Lock em Java

```
import java.util.concurrent.lock.*;
public class Conta {
    private double saldo = 0;
    private Lock lock = new ReentrantLock();

    public double getSaldo() {
        lock.lock();
        try {
            return saldo;
        } finally {
            lock.unlock();
        }
    }
    // ... idem para os demais métodos
}
```

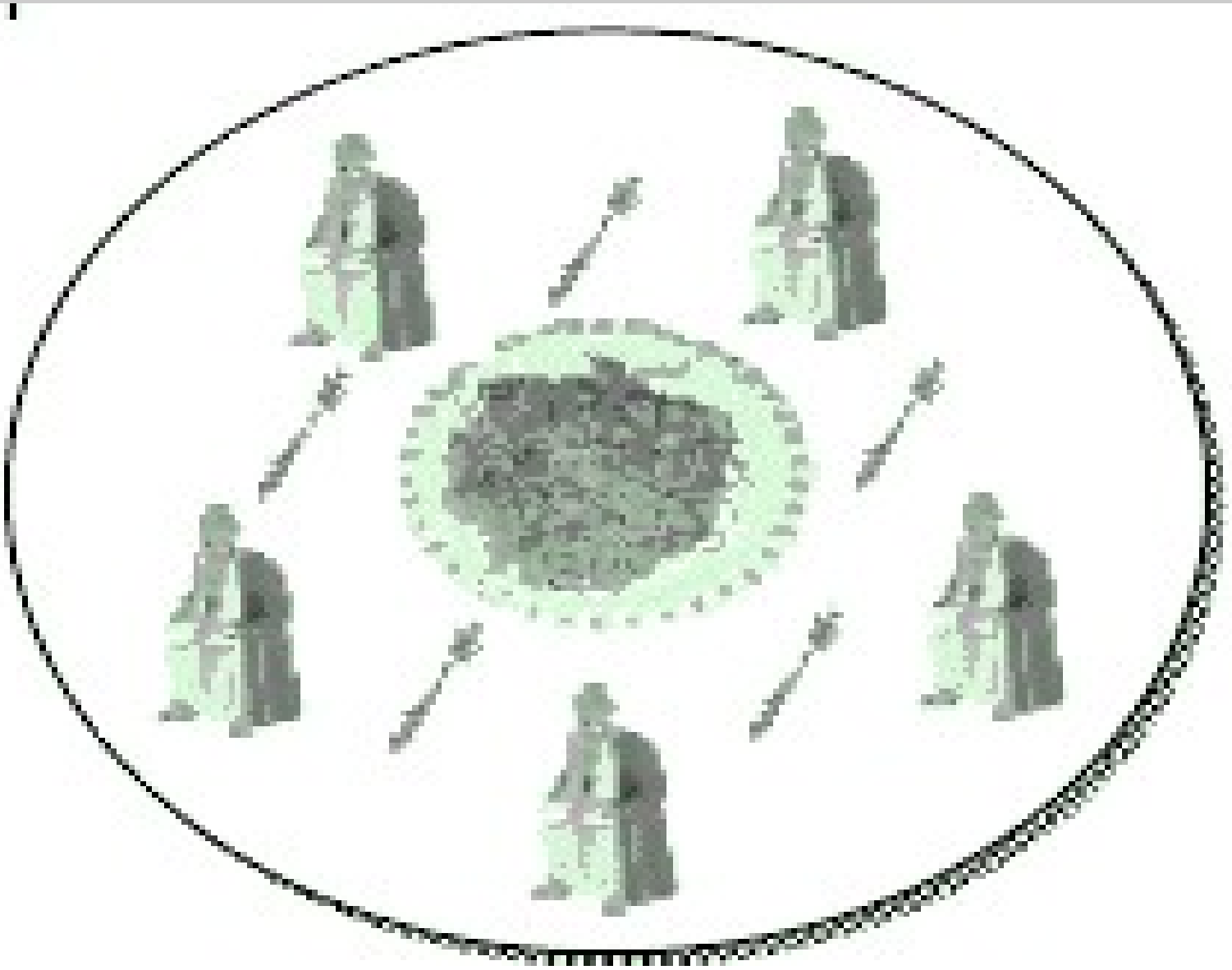
Lock em Java

- Interface ReadWriteLock
  - Possui dois Locks:
    - readLock(): para acesso compartilhado de threads com direito de leitura (acesso);
    - writeLock(): para acesso exclusivo de uma thread com direito de escrita (modificação)
  - Baseada em ReentrantReadWriteLock;
  - Por default não garante a ordem de liberação nem preferência entre leitores e escritores;
  - Ordenação FIFO é garantida passando o valor ‘true’ para o construtor da classe;

Lock em Java

```
import java.util.concurrent.lock.*;
public class Conta {
    private double saldo = 0;
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    public double getSaldo() {
        lock.readLock().lock();
        try { return this.saldo; }
        finally { lock.readLock.unlock(); }
    }
    public double setSaldo(double saldo) {
        lock.writeLock().lock();
        try { this.saldo = saldo; }
        finally { lock.writeLock.unlock(); }
    }
    // ... idem para debitarValor()
}
```

Lock´s
Jantar dos Filósofos



Lock´s
Jantar dos Filósofos

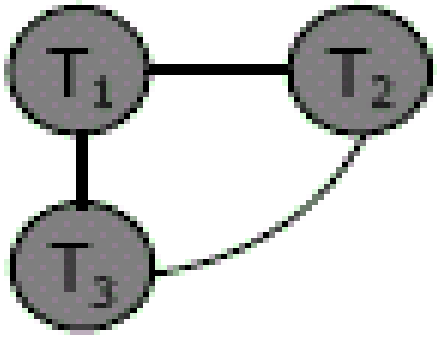
```
public class Filosofo extends Thread{
    private static Lock garfo[] = new Lock[5];
    private int id = 1;
    public Filosofo(int id) { super("Filosofo-"+id); this.id = id; }
    public void run() {
        while(true) try {
            sleep((long)(Math.random()*5000)); // Pensando...
            garfo[id-1].lock(); // Pega garfo à sua esquerda
            garfo[id%5].lock(); // Pega garfo à sua direita
            sleep((long)(Math.random()*5000)); // Comendo...
        } catch(InterruptedException ie) {}
        finally {
            garfo[id-1].unlock(); // Solta garfo à sua esquerda
            garfo[id%5].unlock(); //Solta garfo à sua direita
        }
    }
}
```

Dead Lock
Jantar dos Filósofos

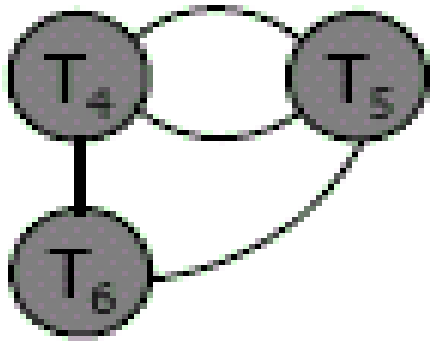
- Caso os cinco filósofos peguem o garfo da esquerda, nenhum deles conseguirá comer;
- Esta situação é chamada de deadlock;
- Deadlock ocorre quando, em um grupo de processos/threads em espera, uma aguarda o término da outra para que possa prosseguir;
- Em Java, as threads ficarão em espera indefinidamente;
- Algumas linguagens/sistemas detectam o deadlock e reportam exceções;

Detecção de Dead Lock

- Verificar o estado do sistema periodicamente para determinar se ocorreu deadlock;
- Precisa saber que bloqueios estão ativos;
- Deadlocks são detectados usando gráfico de espera, no qual um ciclo indica um deadlock;



grafo sem ciclo



grafo com ciclo

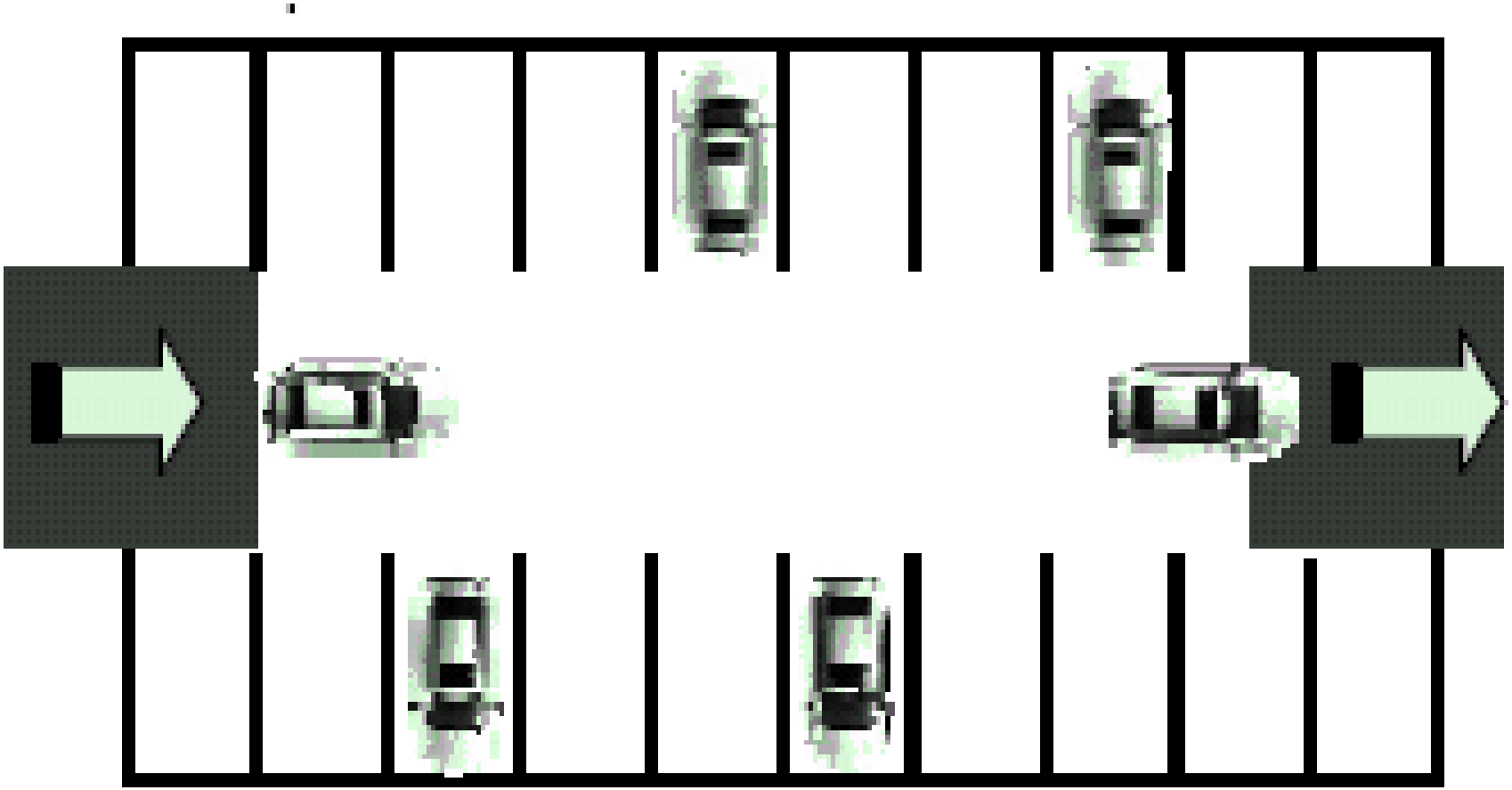
Recuperação de Dead Lock

- Ao detectar um deadlock, deve-se abortar uma thread/processo para quebrar o ciclo de espera;
- Thread/processo abortado pode ser reiniciado;
- Critérios possíveis de escolha da vítima:
- Tempo em que iniciou o processamento;
- Tempo necessário para sua conclusão;
- Operações de I/O já efetuadas ou a efetuar;
- Número de abortos sofridos (para evitar que a vítima seja sempre a mesma);

Semáforos

- Permite controlar o número de acessos simultâneos a um dado ou recurso;
- Métodos da classe Semaphore:
  - Semaphore(int acessos [, boolean ordem]): construtor; parâmetros definem o número de acessos simultâneos possíveis e se a ordem de liberação de threads em espera será FIFO;
  - acquire(): solicita acesso a um dado ou recurso, entrando em espera se todos os direitos de acesso estiverem sendo usados;
  - release(): libera um direito de acesso;

Semáforos



Java com Semáforos problema do Estacionamento

```
import java.util.concurrent.*;
public class Carro extends Thread {
    private static Semaphore estacionamento = new Semaphore(10,true);
    public Carro(String nome) { super(nome); }
    public void run() {
        try {
            estacionamento.acquire();
            System.out.println(getName() + " ocupou vaga.");
            sleep((long)(Math.random() * 10000));
            System.out.println(getName() + " liberou vaga.");
            estacionamento.release();
        } catch(InterruptedException ie){ ie.printStackTrace(); }
    }
    public static void main(String args[]) {
        for (int i = 0; i< 20; i++){
            new Carro("Carro #"+i).start();
        }
    }
}
```