

Unidade 3

Controle de Concorrência

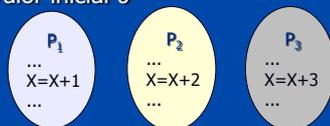
- Monitores
- Locks
- Semáforos
- Concorrência na API Java

Controle de Concorrência

- Se bem utilizado, o paralelismo resulta em um melhor desempenho dos programas
 - Mais threads → processador melhor utilizado
- No entanto, podem ocorrer problemas no acesso concorrente a 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:
 - Suponha que X é um dado compartilhado, com valor inicial 0



- Qual o valor final de X?
- Assembly:

```
Load X
Add 1
Store X
```

```
Load X
Add 2
Store X
```

```
Load X
Add 3
Store X
```

Controle de Concorrência

- Exemplo: Conta Bancária



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();
    }
}
```

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() + " sacou total de R$" + total);
    }
}
```

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() +
            " sacou R$" + valor + ". Saldo após saque: R$" + novoSaldo);
        conta.setSaldo(novoSaldo);
        return true;
    }
}
```

Controle de Concorrência

Exemplo: Conta Bancária

```
public class Conta {
    public double saldo = 0;

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

    public double getSaldo() {
        return saldo;
    }

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

Controle de Concorrência

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
...
```

Controle de 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, e, antes da finalização do saque, uma outra thread altere o valor do saldo, esta alteração será ignorada, pois o saldo após o saque será calculado com base no valor lido inicialmente
- É preciso evitar que outras threads alterem o saldo desta conta enquanto um saque estiver em andamento

Controle de Concorrência

- Problemas causados pela 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 de Concorrência

- Mecanismos de Controle de Concorrência disponíveis em Java:
 - **Monitor**: protege trechos de código/métodos que manipulam dados/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

- **Histórico**
 - Proposto por Hoare em 1974;
 - 1ª implementação: Pascal Concorrente [1975];
 - Usado em várias linguagens, inclusive Java;
- **Funcionamento**
 - “Monitora” o acesso a dados e recursos compartilhados por processos e threads;
 - Encapsula código que faz o acesso a dados e recursos monitorados;
 - Chamadas executadas com **exclusão mútua** - um acesso por vez - criando uma fila de espera.

Monitores

- **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

- **Sincronização de Bloco**

```
synchronized (objeto) {
    // código protegido
}
```

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

Monitores

- **Ex.: Saque com sincronização 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() +
                " sacou R$" + valor + ". Saldo: após saque: R$" + novoSaldo);
            conta.setSaldo(novoSaldo);
            return true;
        }
    }
}
```

Monitores

- **Sincronização de Método**

```
public synchronized void metodo (int param) {
    // 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

- **Ex.: Conta com sincronização 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;
        }
    }
}

public class Banco {
    public boolean saque(Conta c, double v) {
        ...
    }
}
```

Monitores

- Java permite que sejam definidas **condições de acesso** dentro de monitores
 - Uma thread em um monitor deve chamar o método `wait()` se não puder prosseguir devido a alguma condição necessária não-satisfeita; o acesso ao monitor é então liberado
 - Sempre que uma condição de espera for modificada, podemos notificar uma thread em espera na fila, escolhida aleatoriamente, com o método `notify()`, ou notificar todas as threads na fila chamando o método `notifyAll()`
 - Chamá-los fora do monitor resulta em exceção**

Monitores

- Exemplo: Produtor, Consumidor e Buffer
 - Produtor produz itens e os coloca no buffer
 - Consumidor retira e consome os itens



Monitores

```

Exemplo 1
public class ProdCons { // Problema dos Produtores e Consumidores
    public static java.util.Stack buffer = new java.util.Stack(); // buffer
    public static final int BUFSIZE = 5; // tamanho máximo do buffer

    public static void main (String args[]) {
        Runnable prod = new Runnable() { // Código do Produtor
            public void run() {
                for (int i=0; i<100; i++)
                    synchronized(buffer) { // bloqueia acesso ao buffer
                        if (buffer.size() >= BUFSIZE) // se o buffer estiver cheio
                            try { buffer.wait(); } // aguarda lugar no buffer
                            catch (InterruptedException e) {}
                        buffer.push(new Integer(i)); // põe item no buffer
                        System.out.println("Produz "+i+". Total: "+buffer.size());
                        buffer.notify(); // avisa que item foi produzido
                    }
            }
        };
    }
};

```

Monitores

```

Runnable cons = new Runnable() { // Código do Consumidor
    public void run() {
        for (int i=0; i<100; i++)
            synchronized(buffer) { // bloqueia acesso ao buffer
                if (buffer.size() == 0) // se o buffer estiver vazio
                    try { buffer.wait(); } // aguarda item para consumir
                    catch (InterruptedException e) {}
                int j = ((Integer)buffer.pop()).intValue(); // consome item
                System.out.println("Consome "+j+". Total: "+buffer.size());
                buffer.notify(); // avisa que lugar foi liberado
            }
        }
};

new Thread (prod).start(); // Inicia o Produtor
new Thread (cons).start(); // Inicia o Consumidor
};

```

Monitores

```

Exemplo 2
public class WaitNotify { //Exemplo do uso do wait/notify
    public static void main(String[] args) {
        for (int i = 1; i < 10; i++) {
            Aluno a = new Aluno();
            a.start();
        }
    }
}

class Aluno extends Thread {
    static Banheiro b = new Banheiro();
    public void run() {
        try {
            b.vaso();
            Thread.sleep(50); //Indo para a pia
            w.pia();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Monitores

```

class Banheiro { //Banheiro tem apenas um vaso sanitário (região crítica)
    int i = 1;
    synchronized void vaso() throws InterruptedException {
        while (i <= 0) { //Banheiro ocupado
            System.out.println("Sou a " + Thread.currentThread().getName() +
                " e entrei na fila do banheiro (WAIT).");
            wait(); //Espera na fila do banheiro
        }
        i--;
        System.out.println("---Sou a " + Thread.currentThread().getName()
            + " e estou no vaso sanitario => i = " + i);
        Thread.sleep(4000); //Fazendo as necessidades
        System.out.println("---Sou a " + Thread.currentThread().getName()
            + " e estou saindo do vaso sanitario => i = " + i);
    }
}

```

Monitores

```
synchronized void pia() throws InterruptedException {
    i++;
    System.out.println("---Sou a " + Thread.currentThread().getName()
        + " e estou na pia lavando as mãos => i = " + i);
    Thread.sleep(2000); //Lavando as mãos
    System.out.println("---Sou a " + Thread.currentThread().getName()
        + " e sai do pia => i = " + i);
    System.out.println();
    notifyAll(); //Avisa todos que o banheiro está livre
}
}
```

Monitores: Barreira de sincronização

```
import java.util.*;
class Barreira {
    private int ThreadsParticipante, EsperandoNaBarreira;
    public Barreira(int num) {
        ThreadsParticipante = num;
        EsperandoNaBarreira = 0;
    }
    public synchronized void Alcançado() {
        EsperandoNaBarreira++;
        if(ThreadsParticipante != EsperandoNaBarreira) {
            try {
                wait();
            } catch (InterruptedException e) {}
        } else {
            notifyAll();
            EsperandoNaBarreira=0;
        }
    }
}
```

Locks

- Interface **Lock**
 - 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

Locks

- Outros métodos da interface Lock:
 - **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

Locks

- 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

Locks

- Ex.: Classe Conta com ReentrantLock

```
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
}
```

Locks

■ Ex.: uso do tryLock

```
class SleepLock {
    Lock l;
    public SleepLock() {
        l = new ReentrantLock(true);
    }
    void put() {
        l.lock(); // analise o comportamento substituindo por l.tryLock
        try {
            System.out.println("Entrou !");
            Thread.sleep(1000);
            System.out.println("Saiu !");
        } catch (InterruptedException e) {e.printStackTrace();}
        l.unlock();
    }
}
```

Locks

■ Ex.: uso do tryLock

```
import java.util.concurrent.locks.*;
import java.util.concurrent.locks.ReentrantLock;
public class TestTryLock {
    public static void main(String[] args) {
        SleepLock sl = new SleepLock();
        for (int i = 0; i < 10; i++) {
            new ThreadLock(sl).start();
        }
    }
}
class ThreadLock extends Thread {
    SleepLock sl;
    public ThreadLock(SleepLock sl) {this.sl = sl;}
    public void run() {
        sl.put();
    }
}
```

Locks

■ 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)
- Implementada por **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

Locks

■ Ex.: Conta com ReentrantReadWriteLock

```
import java.util.concurrent.locks.*;
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()
}
```

Locks

■ Exemplo: Jantar dos Filósofos



Locks

```
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
        }
    }
}
```

Locks

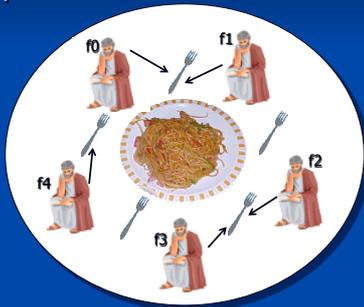
- **Deadlock**
 - Caso os cinco filósofos peguem o garfo da esquerda, nenhum deles conseguirá comer
 - Esta situação é chamada de **deadlock**
 - **Deadlock** ocorre quando, em um grupo de processos/threads em espera, uma aguarda o término da outra para que possa prosseguir
 - Em Java, as threads ficarão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

Locks

- **Jantar dos filósofos**
 - Cada processo deve ter um identificador (0 a 4)
 - Os pares tentam pegar primeiro o garfo a esquerda
 - Os ímpares tentam pegar primeiro o garfo a direita

Locks

- **Exemplo: Jantar dos Filósofos**



Locks

- **Deteção de Deadlock**
 - Verificar o estado do sistema periodicamente para determinar se ocorreu **deadlock** `isLocked()`
 - Precisa saber que bloqueios estão ativos
 - **Deadlocks** são detectados usando gráfico de espera, no qual um ciclo indica um **deadlock**

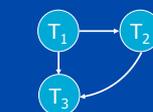


Gráfico sem ciclo

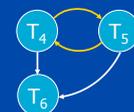


Gráfico com ciclo

Locks

- **Recuperação de Deadlock**
 - 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)
 - etc.

Locks

- Locks podem ser associados a **condições de acesso** usando a classe **Condition**
 - Seu uso é mais flexível que em monitores
 - Condição deve ser associada a um Lock
 - Criada com o método `newCondition()` de Lock
 - Principais métodos:
 - `await()`: aguarda condição ser alterada; tempo limite de espera pode ser estipulado
 - `signal()`: sinaliza que houve alteração da condição, tirando uma thread da fila
 - `signalAll()`: retira todas as threads da fila

Locks

```
public class CircularBuffer implements Buffer { // © Deitel & Assoc.
    private int[] buffer = { -1, -1, -1 }; // buffer com 3 lugares
    private int occupiedBuffers = 0, writeIndex = 0, readIndex = 0;
    private Lock lock = new ReentrantLock(); // lock de acesso ao buffer
    private Condition canWrite = lock.newCondition(); // condição p/ ler
    private Condition canRead = lock.newCondition(); // cond. p/ escrita

    public void set(int value) { // coloca um valor no buffer
        lock.lock(); // bloqueia o acesso ao buffer
        try {
            while ( occupiedBuffers == buffer.length ) // buffer cheio
                canWrite.await(); // espera que haja lugar no buffer
            buffer[ writeIndex ] = value; // coloca valor no buffer
            writeIndex=(writeIndex+1)%buffer.length; // calc. próx. posição
            occupiedBuffers++; // mais um lugar foi ocupado
            canRead.signal(); // avisa threads esperando para ler do buffer
        } catch ( InterruptedException e ) { e.printStackTrace(); }
        finally { lock.unlock(); } // libera acesso ao buffer
    } // fim do método set
}
```

Locks

```
// Continuação da classe CircularBuffer
public int get() { // retira valor do buffer
    int readValue = 0; // valor que será lido do buffer
    lock.lock(); // bloqueia acesso ao buffer
    try {
        while ( occupiedBuffers == 0 ) // se o buffer estiver vazio
            canRead.await(); // aguarda que haja um valor para ser lido
        readValue = buffer[ readIndex ]; // lê um valor do buffer
        readIndex =(readIndex+1)%buffer.length; // calc. próx. posição
        occupiedBuffers--; // um lugar foi liberado
        canWrite.signal(); // avisa threads esperando para escrever
    } catch ( InterruptedException e ) { e.printStackTrace(); }
    finally {
        lock.unlock(); // libera acesso ao buffer
    }
    return readValue;
} // fim do método get
} // fim da classe CircularBuffer
```

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

- Exemplo: Estacionamento



Semáforos

- Métodos da classe **Semaphore**
 - **acquire(int permits)**: solicita um dado número de acesso (permits) a um dado ou recurso, entrando em espera se todos os direitos de acesso solicitados não estiverem disponíveis;
 - **release(int permits)**: libera os direitos de acesso de volta para o semaforo;
 - **int availablePermits()**: retorna o número de corrente de permissões disponíveis neste semáforo;

Semáforos

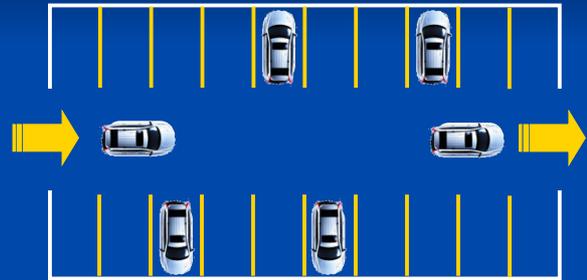
- Métodos da classe **Semaphore**
 - **reducePermits(int reduction)**: reduz o quantidade de permissões disponíveis;
 - **boolean tryAcquire(int permits)**: adquire um número de permissões deste semáforo, só se estiver disponível no momento da invocação;
 - **boolean tryAcquire(int permits, long timeout, TimeUnit unit)**: adquire um número de permissões deste semáforo, só se estiver disponível até o tempo de espera definido e a thread não tiver sido interrompida;

Semáforos

- Métodos da classe **Semaphore**
 - `int availablePermits()`: retorna o número de corrente de permissões disponíveis neste semáforo;
 - `int drainPermits()`: adquire e retorna todas permissões que estão imediatamente disponíveis;

Semáforos

- Exemplo: Estacionamento



Semáforos

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

Semáforos

```
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();
    }
}
```

Semáforos

```
public class LimitadorConexoes{
    private final Semaphore s;
    private LimitadorConexoes (int numMaxRequisicoes) {
        s = new Semaphore(numMaxRequisicoes);
    }
    public URLConnection acquire(URL url) throws InterruptedException,
    IOException {
        s.acquire(); // thread fica bloqueada se alcançar o limite
        return url.openConnection();
    }
    public void release(URLConnection conn) {
        try {
            // ...
        } finally {
            s.release();
        }
    }
}
```

Concorrência na API Java

- Exercício:
 - Implemente um semáforo equivalente à classe `Semaphore` do Java usando o mecanismo de lock (classe `ReentrantLock`).

Concorrência na API Java

- Algumas classes da API Java controlam a concorrência internamente → *thread safe*
 - Ex.: *Vector*, *Hashtable*, ...
- Outras classes não fazem o controle
 - São *thread unsafe*, ou seja, não garantem a sua consistência se usadas por várias threads
 - Estas classes são em geral mais rápidas, pois controle de concorrência reduz o desempenho
 - Classes *thread unsafe* devem ser protegidas se forem usadas por mais de uma thread
 - Ex.: componentes do *Swing*, *LinkedList*, ...

Concorrência na API Java

- Para evitar acesso concorrente a classes *thread unsafe*, podemos criar novas classes protegidas que as encapsulem

```
public class SynchronizedLinkedList {
    LinkedList lista = new LinkedList();
    public synchronized void add(Object o) {
        lista.add(o);
    }
    public synchronized Object get(int index) {
        return lista.get(index);
    }
    // idem para os demais métodos de LinkedList
}
```

Concorrência na API Java

- Mesmo ao usar classes da API *thread safe*, é preciso tomar cuidado ao utilizá-las

```
Vector v = new Vector();
Object o;
...
// Percorre o Vetor
for (int i=0; i<v.size();i++) {
    o = v.get(i); /* Pode causar
                 ArrayIndexOutOfBounds */
}
...
}
```

```
Vector v = new Vector();
Object o;
...
synchronized(v) { // Bloqueia v
    // Percorre o Vetor
    for (int i=0; i<v.size();i++) {
        o = v.get(i);
    }
} // Libera o acesso ao Vetor
```

Concorrência na API Java

- Interface **BlockingQueue<E>**
 - Fornece métodos para acesso a uma fila de elementos genérica que bloqueia automaticamente se alguma condição impedir o acesso
 - Principais métodos:
 - **put()** coloca elemento na fila, aguardando se ela estiver cheia
 - **take()** retira o elemento da fila, aguardando se ela estiver vazia
 - **remainingCapacity()** informa o número de lugares restantes na fila

Concorrência na API Java

- Implementações de **BlockingQueue**
 - **ArrayBlockingQueue**: array bloqueante
 - **DelayQueue**: fila na qual um elemento só pode ser retirado após seu delay expirar
 - **LinkedBlockingQueue**: fila encadeada bloqueante
 - **PriorityBlockingQueue**: fila bloqueante com acesso aos elementos em ordem de prioridade
 - **SynchronousQueue**: cada **put()** é sincronizado com um **take()**

Concorrência na API Java

```
import java.util.concurrent.ArrayBlockingQueue;
public class BlockingBuffer implements Buffer { // © Deitel & Assoc.
    private BlockingQueue<Integer> buffer =
        new ArrayBlockingQueue<Integer>(3); // buffer de 3 lugares

    public void set(int value) { // coloca um valor no buffer
        try {
            buffer.put(value); // coloca valor no buffer
        } catch (InterruptedException e) { e.printStackTrace(); }
    } // fim do método set

    public int get() { // retira valor do buffer
        int readValue = 0; // valor que será lido do buffer
        try {
            readValue = buffer.take(); // lê um valor do buffer
        } catch (InterruptedException e) { e.printStackTrace(); }
        return readValue;
    } // fim do método get
} // fim da classe BlockingBuffer
```

Concorrência na API Java

- O pacote `java.util.concurrent.atomic` fornece classes *thread safe* equivalentes a alguns tipos de dados do Java:
 - `AtomicBoolean`
 - `AtomicInteger` e `AtomicIntegerArray`
 - `AtomicLong` e `AtomicLongArray`
 - `AtomicReference` e `AtomicReferenceArray`
 - etc.

Concorrência na API Java

- Exemplos de métodos dos tipos atômicos
 - `get()`, `set()`: retorna/altera valor atômicamente
 - `compareAndSet()`: compara o valor e, caso seja igual, o modifica
 - `getAndAdd()`: retorna valor atual e adiciona
 - `addAndGet()`: adiciona e retorna novo valor
 - `getAndDecrement()`, `getAndIncrement()`: retorna valor atual e decrementa/incrementa
 - `decrementAndGet()`, `incrementAndGet()`: decrementa/incrementa e retorna novo valor

Concorrência na API Java

- Componentes Swing e Threads
 - Componentes Swing não são *thread safe*
 - Torná-los *thread safe* reduziria o desempenho
 - Todas as alterações em componentes devem ser efetuadas pela thread de despacho de eventos, ou podem ocorrer inconsistências
 - Alterações são agendadas para serem executadas pela thread de despacho usando o método `SwingUtilities.invokeLater(Runnable r)`
 - Alterações em componentes devem estar no código do método `run()` do `Runnable`

Concorrência na API Java

- Ex.: Acesso a componentes Swing em threads

```

...
private static java.util.Random generator = new Random();
private javax.swing.JLabel output = new javax.swing.JLabel();
final String threadName = Thread.currentThread().getName();
...
// Gera um caractere aleatoriamente e depois mostra na tela
javax.swing.SwingUtilities.invokeLater(
    new Runnable() {
        public void run() {
            // gera caractere aleatório
            char displayChar = (char) (generator.nextInt(26) + 'A');
            // mostra o caractere no JLabel output
            output.setText( threadName + ": " + displayChar );
        } // fim do método run
    } // fim da classe interna anônima
); // fim da chamada a SwingUtilities.invokeLater
...

```