Java

Mecanismos de Sincronização Adicionais

- semáforos, locks, barreiras, ...

- **Autores**
  - Versão inicial: alunos da pós, disciplina POD
    - Cristiano André da Costa
    - Alan Carvalho Assis
    - Gustavo Frainer
    - Rômulo Rosinha
  - Revisões, adições: C. Geyer

- **Local**
  - Instituto de Informática
  - UFRGS
  - disciplinas:
    - Programação Distribuída e Paralela
    - Programação com Objetos Distribuídos
  - versão atual:
    - V05.9
    - Abr de 2017

- **Súmula**
  - sincronização em Java
    - Outros mecanismos: visão geral:
      - Variáveis atômicas
      - Locks
      - Barreiras
    - Semáforos
  - Obs.: adicional a monitores (em outro arquivo)

Bibliografia

- **Bibliografia e links**
  - No final desses slides
  - A mesma lista usada em Java Threads (e sincronização básica)

Outros Mecanismos de Sincronização a partir do J2SE 5

Motivação

- **JSR (Java Specification Requests) 166**
  - conjunto de utilitários de nível médio que fornecem funcionalidade adicional e necessária em programas concorrentes
  - proposto por Doug Lea
  - incorporado no JDK 5.0
  - principais contribuições
    - construtores de threads de alto nível, incluindo executores (thread task framework)
    - filas seguras de threads
    - Timers
    - locks (incluindo atômicos)
    - e outras primitivas de sincronização (como semáforos)

Concorrência no JDK 5.0

- **JSR (Java Specification Requests) 166**
  - Motivação (conforme tutorial Oracle – Sun)
    - http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html
    - Reduz esforço de programação
      - Mais fácil usar bibliotecas padrões do que desenvolver uma específica
    - Aumento de desempenho
      - Bibliotecas projetadas e desenvolvidas por especialistas
      - É difícil implementar uma alternativa com melhor desempenho

Concorrência no JDK 5.0

- **JSR (Java Specification Requests) 166**
  - Motivação (conforme tutorial Oracle – Sun)
    - Maior confiabilidade
      - Desenvolver bibliotecas para programação concorrente é algo complexo
      - É fácil introduzir bugs
      - A depuração é custosa
      - Usar bibliotecas padrões reduzem a possibilidade de bugs na camada de aplicação
    - Manutenção simplificada
      - É mais fácil manter programas concorrentes que usam bibliotecas padrões (e que evoluem) do que bibliotecas “confusas” (?) feitas “em casa”

- **JSR (Java Specification Requests) 166**
  - Motivação (conforme tutorial Oracle – Sun)
    - Aumento de produtividade
      - Classes (bibliotecas) desenvolvidas em acordo com conceitos bem estabelecidos
      - => são mais fáceis de entender e usar que
      - Classes desenvolvidas para problemas específicos, de forma ad-hoc

Concorrência no JDK 5.0

- **Pacote**
  - java.util.concurrent.*
    - Alguns pacotes “filhos”

Novos Mecanismos de Sincronização

- **Sincronizadores do pacote**
  - Outros mecanismos de propósito geral para sincronização
  - Lista “completa”
    - locks – classe Lock
    - semáforos – classe Semaphores (incluindo exclusão mútua);
    - barreiras – classe CyclicBarrier
    - travas – classe CountDownLatch
    - trocadores – classe Exchanger<V>.

Variáveis Atômicas

Novos Mecanismos de Sincronização

- **Variáveis Atômicas**
  - manipulação atômica de variáveis (tipos primitivos ou objetos)
  - fornecendo aritmética atômica de alto desempenho e métodos compare-and-set
  - pacote
    - java.util.concurrent.atomic

- **Variáveis Atômicas**
  - permite a utilização de uma única variável sem a necessidade de *locks* por múltiplas threads
  - Métodos básicos de acesso
    - leitura: *get()*
    - Escrita: *set(value)*
  - Garantia básica
    - Todo *get()* executado após um set retorna garantidamente o novo valor do *set(value)*
    - Obedece relação “happens-before relationship” (terminologia Oracle)

+info

Variáveis Atômicas

- Método usualmente fornecido
  - boolean compareAndSet(expectedValue, updateValue);
  - altera atomicamente uma variável para *updateValue* se ela atualmente tem *expectedValue*, retornando *true* se sucesso
  - base para implementação de *lock* com *busy-waiting*
- principais Classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicLong
  - AtomicReference

+info

Variáveis Atômicas

- **Variáveis Atômicas**
  - Vários outros métodos conforme classe
    - Na classe Atomic

+info

Barreiras

CyclicBarrier (Barreiras)

- **Barreiras**
  - permite que um conjunto de threads esperem até um ponto comum
  - a barreira é denominada cíclica porque pode ser reusada após a liberação das *threads* em espera
  - suporta um comando *Runnable* opcional
    - que é executado uma vez por barreira (uso)
    - após a última thread chegar na barreira
    - mas antes de qualquer thread ser liberada
    - normalmente qq thread executa o comando
    - mas é possível escolher uma entre as da barreira

+info

CyclicBarrier (Barreiras)

- **Barreiras**
  - API
    - construtor `CyclicBarrier(N, Runnable)`
      - N: # de threads a bloquearem-se na barreira
    - bloqueio: `await()`
      - thread bloqueia-se esperando que todas as N … (semântica de barreira)

+info

CyclicBarrier (Barreiras)

- **Barreiras**
  - Exemplo: programa paralelo com decomposição
    - Cada thread processa uma linha de uma matriz
    - Depois cada thread espera em uma barreira única
    - Na barreira, uma thread qualquer executa o método *mergeRows* que faz um merge das linhas

+info

CyclicBarrier (Barreiras)

- **Barreiras**
  - Exemplo: programa paralelo com decomposição
    - Código tem 2 classes, uma interna (a thread)
    - ```
      // classe principal
      class Solver {
          final int N;
          final float[][] data;
          final CyclicBarrier barrier; // barreira
      ```

+info
      ```

CyclicBarrier (Barreiras)

- **Barreiras**
  - Exemplo: programa paralelo com decomposição
    - Classe interna thread: recebe linha; processa; espera outras
    - ```
      class Worker implements Runnable {
          int myRow;
          // construtor recebe número da linha
          Worker(int row) { myRow = row; }
          ...
      ```

+info
      ```

CyclicBarrier (Barreiras)

- **Barreiras**
  - Exemplo: programa paralelo com decomposição
    - Classe interna thread: recebe linha; processa; espera outras
    - …

```
        public void run() {
          while (!done()) {
            processRow(myRow);
          try {
            barrier.await();   // espera outras
        } catch (InterruptedException ex) {
            return;
        } catch (BrokenBarrierException ex) {
            return;
        }
      }  // fim loop (while)
    }}  // fim run e classe
```

+info

CyclicBarrier (Barreiras)

- **Barreiras**
  - Exemplo: programa paralelo com decomposição

```
// construtor da classe principal
// cria a barreira

public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    // cria barreira: passando método merge
  barrier = new CyclicBarrier(N,
                    new Runnable() {
                        public void run() {
                          mergeRows(...);
                                    }
                    });
```

+info

CyclicBarrier (Barreiras)

- **Barreiras**
  - Exemplo: programa paralelo com decomposição

```
// construtor da classe principal
// cria e dispara as threads passando linha
      for (int i = 0; i < N; ++i)
        new Thread(new Worker(i)).start();
      waitUntilDone();
    }
  }
```

+info

Trancas

- **CountDownLatch (Trancas)**
  - similar a barreiras, a diferença é a condição para liberação:
    - não é o número de threads que estão esperando, mas sim quando um contador específico chega a zero
  - threads que executarem o *wait* após o contador já ter atingido o zero são liberadas automaticamente
  - o contador não é resetado automaticamente
  - oferece mais flexibilidade que a CyclicBarrier
  - mas mais trabalhosa (?)

+info

CountDownLatch (Trancas)

- **CountDownLatch (Trancas)**
  - Métodos *await*
    - Bloqueia thread até que contador seja zero
  - Método *countDown()*
    - Decrementa o contador
    - Se contador = zero
      - Threads bloqueadas em *await* são acordadas
    - Chamadas seguintes de *await* não bloqueiam a thread

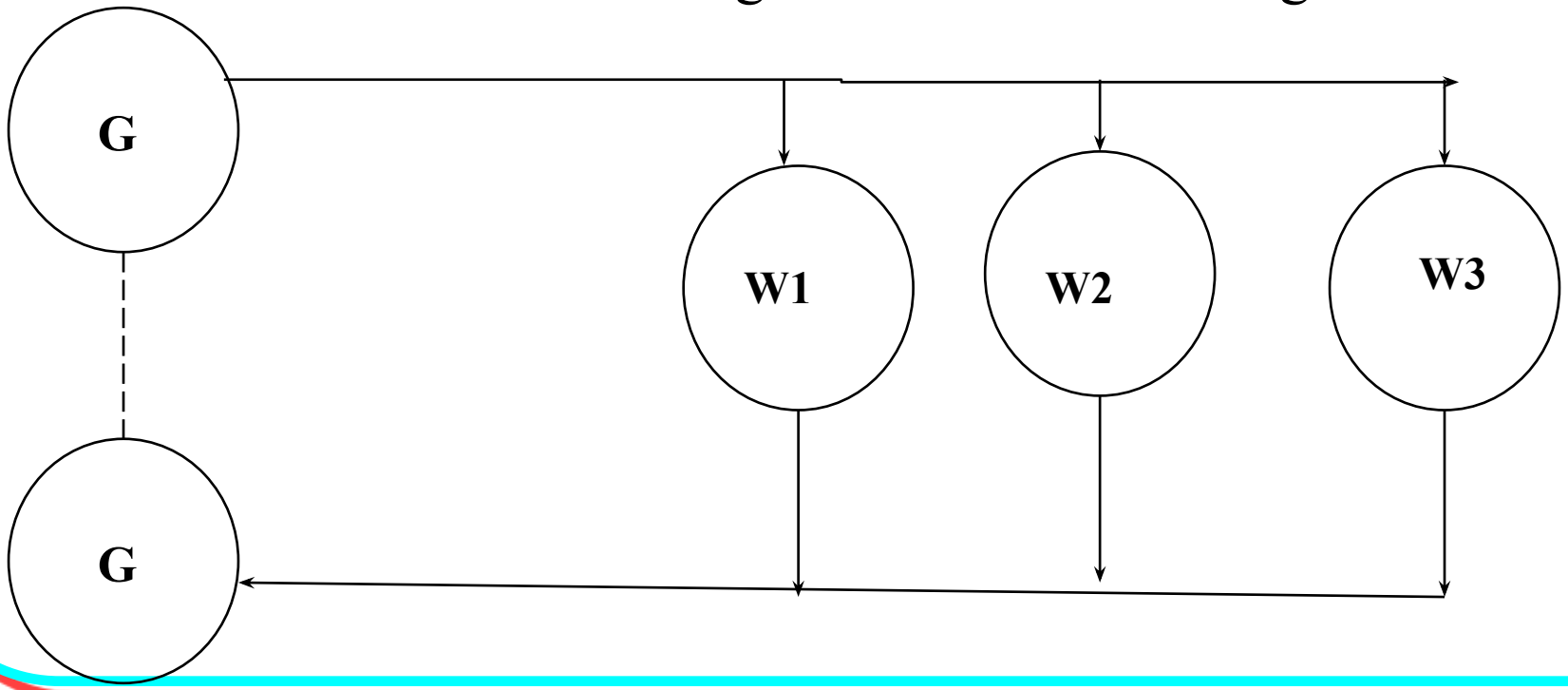
- **CountDownLatch (Trancas)**
  - Aplicações básicas
    - Contador inicializado com 1
    - Todas as threads bloqueiam no *await* até que alguma execute *countDown*
    - Exercícios
      - Compare com eventual uso de *wait/notifyAll*
      - Compare com eventual uso de *lock/unlock*
      - Idem com eventual uso de *P(sem) / V(sem)*

CountDownLatch (Trancas)

- **CountDownLatch (Trancas)**
  - Aplicações básicas
    - Contador inicializado com N
    - Uma thread bloqueia no *await* até que várias threads executem N *countDown*
    - Exercícios
      - Compare com eventual uso de *P(sem) / V(sem)*

CountDownLatch (Trancas)

- Exemplo N workers com gerente
  - Fonte: documentação da Oracle
  - Workers (Wi) esperam sinal inicial do gerente (G)
  - N workers avisam gerente sobre término geral



- **CountDownLatch (Trancas)**
  - Exemplo N workers com gerente
    - Trancas (2):
      - startSignal
        - = 1
        - Workers esperam pelo aviso
        - Gerente avisa início
      - doneSignal
        - = N
        - Gerente espera aviso geral
        - N workers avisam final

CountDownLatch (Trancas)

```
// gerente
class Driver { // …
   void main() throws InterruptedException {
     // 2 trancas são criadas
     CountDownLatch startSignal = new
                        CountDownLatch(1);
     CountDownLatch doneSignal = new
                        CountDownLatch(N);

     // create and start threads
     // passando as trancas
     for (int i = 0; i < N; ++i)
       new Thread(new Worker(startSignal,
           doneSignal)).start();
```

CountDownLatch (Trancas)

- **CountDownLatch (Trancas)**
  -
```
    doSomethingElse();  // don't let run yet
    startSignal.countDown(); // let all threads proceed
    doSomethingElse();
    doneSignal.await();   // wait for all to finish
     }
  }
```

CountDownLatch (Trancas)

- **CountDownLatch (Trancas)**
  - ```
    // código Worker
    class Worker implements Runnable {
       private final CountDownLatch startSignal;
       private final CountDownLatch doneSignal;

       // construtor: recebe trancas
       Worker(CountDownLatch startSignal, CountDownLatch
                     doneSignal) {
         this.startSignal = startSignal;
         this.doneSignal = doneSignal;
       }
    ```
    ```

CountDownLatch (Trancas)

- **CountDownLatch (Trancas)**
  - // código Worker (cont.)

```
public void run() {
      try {
        startSignal.await();   // espera sinal gerente
        doWork();
        doneSignal.countDown();  // avisa gerente
      } catch (InterruptedException ex) {} // return;
   }

   void doWork() { ... }

 }
```

CountDownLatch (Trancas)

- **CountDownLatch (Trancas)**
  - Exercício
    - Reescreva a solução para o problema do Solver paralelo trocando a barreira por uma tranca
      - Compare as soluções

Trocadores

- ***Exchanger* (Trocadores)**
  - servem para trocar dados entre *threads* de forma segura
  - o método *exchange* é chamado com o objeto de dado a ser trocado com outra *thread*
  - se uma *thread* já estiver esperando, o método *exchange* retorna o objeto de dado da outra *thread*
  - se nenhuma *thread* estiver esperando, o método *exchange* ficará esperando por uma
  - obs.: o método *exchange* faz 2 escritas de forma atômica

+info

Locks

Novos Mecanismos de Sincronização (Cont.)

- **Locks**
  - resolve os inconvenientes da palavra-reservada *synchronized*
  - implementa *lock* de alto desempenho com a mesma semântica de memória do que sincronização
  - suportando
    - timeout
    - múltiplas variáveis de condição por lock
      - como em monitores e em Posix threads
    - e interrupção de threads que esperam por lock
  - Pacote
    - java.util.concurrent.locks

- **Lock (mais detalhes)**
  - implementação de exclusão mútua para múltiplas threads
  - substitui o uso de métodos e blocos Syncronized
    - Permite chain lock: adquire A, depois B, libera A, adquire C, ...
  - exemplo de Uso de Lock
    - Lock l = new Lock();

```
l.lock();
try {
   //acesso protegido pelo lock
} finally {
   l.unlock();
}
```

+info

Locks (Cont.)

- **Locks (mais detalhes)**
  - ReentrantLock
    - pode ser justo: usar fila FIFO
    - define os métodos isLocked e getLockQueueLength

+info

  - ReadWriteLock
    - pode permitir várias threads acessarem o mesmo objeto para leitura ou somente uma para escrita

+info

  - Condition
    - adiciona variáveis condicionais
    - uma thread suspende a execução até ser notificada por outra de que uma condição ocorreu

+info

Bibliografia

- Advanced Synchronization in Java Threads, Part 1. Disponível por WWW em http://www.onjava.com/pub/a/onjava/excerpt/jthreads3_ch6/index1.html, 2005. (abr. 2005)
- Concurrency JSR-166 Interest Site. Disponível por WWW em http://gee.cs.oswego.edu/dl/concurrency-interest/index.html, 2005. (abr. 2005)
- Concurrent Programming with J2SE 5.0. Disponível por WWW em http://java.sun.com/developer/technicalArticles/J2SE/concurrency/, 2005. (abr. 2005)

- **Bibliografia**
  - J2SE 5.0 in a Nutshell. Disponível por WWW em http://java.sun.com/developer/technicalArticles/releases/j2se15/, 2005. (abr. 2005).
  - JDK 5.0 Documentation. Disponível por WWW em http://java.sun.com/j2se/1.5.0/docs/index.html, 2005. (abr. 2005).
  - JSR 166: Concurrency Utilities. Disponível por WWW em http://jcp.org/en/jsr/detail?id=166, 2005. (abr. 2005).

Semáforos J2SE 5.0

Estudo da API e comparação com definição clássica

Construindo um semáforo

- **Construindo um java.util.concurrent.Semaphore:**
  - Construtores
    - new Semaphore(int permits) ou
    - new Semaphore(int permits, boolean fair)
  - permits define o valor inicial de semáforo
    - esse valor pode ser negativo
      - nesse caso releases (V) devem ser feitos antes que uma thread possa receber uma permissão através de um acquire (P)

Construindo um semáforo

- **Construindo um java.util.concurrent.Semaphore:**
  - fair define se o semáforo é justo ou não:
    - true
      - garante uma ordem de atendimento FIFO para as threads que invocaram acquire(P) e ficaram em espera
    - false
      - nenhuma garantia é feita sobre a ordem de recebimento de permissões

Construindo um semáforo

- **Construindo um java.util.concurrent.Semaphore:**
  - fair define se o semáforo é justo ou não:
    - um valor true
      - deve ser usado no caso de controle a recursos compartilhados, para impedir que ocorra *starvation* de uma thread
    - em outros casos um valor de false
      - pode ser mais desejável, devido ao ganho de desempenho que ocorre neste modo.

Método acquire

- **Método acquire()**
  - o equivalente do P conforme conceitos de semáforos
  - se o número de "permissões" de um semáforo for maior que 0
    - o contador é diminuído e o método retorna imediatamente
  - se não existir nenhuma permissão disponível no semáforo a thread fica bloqueada até que:
    - outra thread chame release() neste semáforo e a thread atual seja a próxima na lista para receber a permissão.
    - outra thread interrompa a thread atual

Método acquire

- **Versões do método acquire:**
  - void acquire()
  - void acquire(int permits)
    - ao invés de adquirir apenas uma permissão adquire o número de permissões do parâmetro permits
    - a thread é bloqueada até conseguir todas as permissões solicitadas
  - void acquireUninterruptibly()
    - similar ao método básico, mas neste método a thread não pode ser interrompida
  - void acquireUninterruptibly(int permits)
    - similar ao método com argumento, mas neste método a thread não pode ser interrompida

Método tryAcquire

- **Métodos tryAcquire**
  - os métodos *tryAcquire*
    - são variações do Acquire para usos especiais
  - … permitindo:
    - que uma thread tente adquirir uma permissão e continue a execução se não conseguir depois de um certo tempo
    - que todos eles retornem um valor booleano indicando se foi possível ou não adquirir a permissão
    - Os métodos *tryAcquire* não respeitam o uso de fila justa de espera

Métodos tryAcquire

- **Métodos tryAcquire**
  - versões:
    - boolean tryAcquire()
      - adquire uma (1) permissão do semáforo, se uma estiver disponível
    - boolean tryAcquire(int permits)
      - adquire o número de permissões passado, se todas estiverem disponíveis

- **Métodos tryAcquire**
  - versões:
    - boolean tryAcquire(int permits, long timeout, TimeUnit unit)
      - adquire o número de permissões dado, se todas ficarem disponíveis até o *timeout*
    - boolean tryAcquire(long timeout, TimeUnit unit)
      - adquire uma permissão, se uma ficar disponível até o *timeout*

Método release

- **Método release()**
  - o equivalente de V conforme conceitos de semáforos
  - o método release libera uma permissão do semáforo (aumenta o contador em um)
    - se alguma thread estiver tentando realizar um acquire, então uma destas threads é selecionada e recebe a permissão
  - uma thread não precisa ter realizado um acquire() antes de realizar um release()

Método release

- **Método release()**
  - versões de release() :
    - void release()
      - libera uma (1) permissão
    - void release(int permits)
      - libera o número de permissões passado como parâmetro

Métodos de verificação

- **Métodos de verificação de estado do semáforo**
  - int availablePermits()
    - retorna o número de permissões disponíveis no semáforo
  - protected Collection<Thread> getQueuedThreads()
    - retorna uma coleção com as Threads que podem estar tentando adquirir uma permissão neste semáforo
  - int getQueueLength()
    - retorna uma estimativa do número de threads tentando adquirir uma permissão neste semáforo

Métodos de verificação

- **Métodos de verificação de estado do semáforo**
  - boolean hasQueuedThreads()
    - verifica se alguma thread está tentando adquirir uma permissão neste semáforo.
  - boolean isFair()
    - retorna true se o semáforo for justo.

- **Métodos auxiliares**
  - int drainPermits()
    - adquire todas as permissões disponíveis imediatamente e retorna quantas permissões foram adquiridas
  - protected void reducePermits(int reduction)
    - reduz o número de permissões disponíveis pelo número indicado pelo parâmetro reduction, que deve ser um número positivo

Exemplo Seção Crítica

- **Exemplo 1: exclusão mútua clássica**
  - Um objeto que controla o acesso a uma seção crítica
  - class Exclusao {

```
 private final Semaphore livre =
        new Semaphore(1, true);

  public Object acessaSecaoCritica()
          throws InterruptedException {
    livre.acquire();       // pede acesso a seção crítica
    //faz acesso a seção critíca;
 “código-acesso-secao-critica"
    livre.release();       // libera acesso a ...
      }
}
```

Exemplo Produtor/Consumidor

- **Exemplo 2: Produtores Consumidores (Sun)**
  - Especificação diferente da usual para esse problema
  - Não há o problema de buffer cheio
    - Produtor não bloqueia
    - Produtor marca item como livre
  - Há um conjunto de permissões para consumir
    - Consumidor bloqueia se sem permissão
    - Consumidor marca item como em uso

Exemplo Produtor/Consumidor

- **Exemplo 2: Produtores Consumidores (Sun)**
  - Somente 1 semáforo
    - Oferece k permissões de consumo
    - Consumidor decrementa
    - Produtor incrementa
      - Em certa condição

Exemplo Produtor/Consumidor

- **Exemplo 2: Produtores Consumidores (Sun)**
  - Put:
    - Não inclui novo item
    - (somente) Marca item como não usado (livre)
      - Exclusão mútua (synchronized) no acesso ao buffer
      - Procura em todo o vetor
      - Se item já existe e usado marca como não-usado e retorna T
      - Se item já existe e não usado, retorna F
      - Se item não existe, retorna F
    - Se retorno T incrementa permissões (semáforo)

Exemplo Produtor/Consumidor

- **Exemplo 2: Produtores Consumidores (Sun)**
  - Get:
    - Pede permissão (semáforo)
      - bloqueante
    - Retorna primeiro item não usado
      - Exclusão mútua (synchronized) no acesso ao buffer
      - Procura item não usado
      - Marca como usado
      - Retorna item ou null se não encontrado

Exemplo Produtor/Consumidor

- **Exemplo 2: Produtores Consumidores (Sun)**
  - class Pool {

```
  // quantidade disponível no início
 private static final MAX_AVAILABLE = 100;
  // buffer
 protected Object[] items = new Object[100];
  // vetor de booleanos: indica em uso se true
 protected boolean[] used = new
                  boolean[MAX_AVAILABLE];
  // semáforo disponíveis
 private final Semaphore available =
       new Semaphore(MAX_AVAILABLE, true);
```

Exemplo Produtor/Consumidor

- **Exemplo 2: Produtores Consumidores (Sun)**
  - ```
    // métod get para consumidores
    public Object getItem() throws InterruptedException {
      available.acquire(); // reduz disponíveis (bloqueante)
      return getNextAvailableItem();
    }

    // método put para produtores
    public void putItem(Object x) {
      if (markAsUnused(x))        // marca e testa
    available.release();          // aumenta disponíveis
    }
    ```
    ```

Exemplo 2: Produtores Consumidores (Sun)

- **Exemplo 2: Produtores Consumidores (Sun)**
  - // método que retorna próximo item disponível
    // usado pelos consumidores
    // com exclusão mútua

```
protected synchronized Object getNextAvailableItem() {
    // percorre buffer procurando item não usado
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (!used[i]) {
        used[i] = true;
        return items[i];
      }
    }
    return null; // not reached
  }
```

Exemplo 2: Produtores Consumidores (Sun)

- **Exemplo 2: Produtores Consumidores (Sun)**
  - // método que marca item como não usado
    // usado pelos produtores
    protected synchronized boolean markAsUnused(Object item)
    {
        …
    }

Exemplo 2: Produtores Consumidores (Sun)

- **Exemplo 2: Produtores Consumidores (Sun)**
  - protected synchronized boolean markAsUnused(Object item)

```
{
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
      if (item == items[i]) {
        if (used[i]) {
          used[i] = false;
          return true;
        } else
          return false;
      }
    }
    return false;
}}
```

Exemplo 2: Produtores Consumidores (Sun)

- **Exemplo 2: Produtores Consumidores (Sun)**
  - o mais interessante de se notar neste exemplo
    - é que a thread não está em um método sincronizado quando ela chama acquire e release
    - isto é importante para permitir que várias threads façam acquire
    - e que
      - (get) enquanto uma thread está trancada no acquire
      - (put) outra ainda assim possa colocar um item e realizar um release

Exemplo 2: Produtores Consumidores (Sun)

- **Exemplo 2: Produtores Consumidores (Sun)**
  - o semáforo controla a sincronização necessária para restringir o acesso à fila
    - independentemente da sincronização necessária para manter a consistência da fila

Comparação com conceito de semáforos

- **Comparação entre Semaphore e a definição clássica de semáforos**
  - Semaphore
    - é basicamente uma versão orientada a objetos da definição clássica de semáforos
    - onde P e V foram substituídos pelos métodos do próprio objeto semáforo acquire() e release()
  - Semaphore
    - entretanto, possui alguns métodos que não têm equivalente nos semáforos clássicos
      - o principal sendo tryAcquire()

Comparação com conceito de semáforos

- **Comparação entre Semaphore e a definição clássica de semáforos**
  - nos Semaphores
    - o contador também não pode ser simplesmente colocado em algum valor arbitrário depois da criação
    - mas é possível manipular este contador através de reducePermits() e de releases()

Executores

Novas formas para expressão da concorrência

- **Executores**
  - Conjunto de novas interfaces para criação e gerência de threads
  - Desde mais concisão (simplicidade) até modelo roubo de tarefas passando por melhor desempenho na criação
  - Principais interfaces
    - Executor:
      - disparo simples de nova thread
    - ExecutorService:
      - Inclui novos recursos para otimizar o ciclo de vida
    - ScheduleExecutorService
      - Suporta execução periódica ou no futuro

- **Executores**
  - Interface *Executor*
    - Oferece um novo método *execute()*
    - Substitui instanciação (*new*) + disparo (*start*)
    - Recebe objeto *runnable*
    - Uso padrão
      - e: objeto *Executor*
      - r: objeto *runnable*
      - `e.execute(r)`

- **Executores**
  - Interface *Executor* (...)
    - Dependendo da implementação:
      - Cria uma nova *thread*
      - Ou reusa *thread* existente
      - Ou coloca em uma fila à espera de *thread* disponível

- **Executores**
  - Interface *ExecutorService*
    - Oferece um novo método *submit()*
    - *submit()* pode receber objetos *Callable*
    - Objetos *Callable*
      - Tarefa pode retornar um valor
    - *submit()* retorna um objeto *Future*
      - Usado para buscar o valor de retorno de *Callable*

- **Executores**
  - Interface *ExecutorService*
    - É possível submeter um grande número de *Callable*
    - Oferece métodos para gerenciamento do ciclo de vida (*shutdown*) executor
    - Nesse caso (*shutdown*), as tarefas devem tratar interrupções de forma correta

- **Executores**
  - Interface *ScheduleExecutorService*
    - Oferece um novo método *schedule()*
    - *schedule()* executa *Runnable* ou *Callable* após um dado tempo
    - Também oferece os métodos *scheduleAtFixedRate e scheduleWithFixedDelay*
      - Tarefas executadas repetidamente com intervalos fixos

- **Executores**
  - Thread Pools
    - Muitas implementações dos Executores usam thread pools
    - Thread pools permitem reusar uma thread para diferentes tarefas (*Runnable* e *Callable*)
    - Principal vantagem
      - Minimizar o custo de criação de thread
      - Objetos thread consomem muita memória
      - Se aplicação usa muitos objetos, alocar e desalocar memória pode ter alto custo

- **Executores**
  - Fixed thread pool
    - Usam sempre uma quantidade fixa de threads
    - Se um thread é terminada (por algum motivo) o sistema cria uma nova thread automaticamente
    - Há uma fila interna
      - Mantém tarefas sem threads quando há mais tarefas que threads

- **Executores**
  - Fixed thread pool
    - Exemplo de uso em servidor web
      - Caso A):
        - Servidor cria uma thread a cada conexão
        - Servidor pode parar de responder se criar mais threads do que sua capacidade
      - Caso B):
        - Servidor cria antecipadamente o número de threads que sua capacidade permite

- **Executores**
  - Fixed thread pool
    - Criação básica
      - Método factory *newFixedThreadPool()*
    - Criação flexível
      - Permite expandir o pool
      - Método factory *newCachedThreadPool()*
    - Criação com uma única ativa
      - Método factory *newSingleThreadExecutor()*

Resumo

Geral de Sincronização +

- **Novos recursos na versão Java SE 8.0**
  - Classes and interfaces have been added to the java.util.concurrent package.
  - Methods have been added to the java.util.concurrent.ConcurrentHashMap class to support aggregate operations based on the newly added streams facility and lambda expressions.
  - Classes have been added to the java.util.concurrent.atomic package to support scalable updatable variables.

- **Novos recursos na versão Java SE 8.0**
  - Methods have been added to the java.util.concurrent.ForkJoinPool class to support a common pool.
  - The java.util.concurrent.locks.StampedLock class has been added to provide a capability-based lock with three modes for controlling read/write access.

Exercícios

- **Exercícios adicionais**
  - A) morte de uma thread
    - os métodos previstos no pacote Java threads para a suspensão (suspend()) ou término de execução (stop()) não devem ser usados segundo recomendação da documentação da Sun;
    - se usados, podem colocar o programa em um estado inconsistente;
    - descreva uma solução alternativa para o problema usando comandos normais de Java (atribuição, variáveis simples, chamada de método da aplicação, ...);
    - dica: busy waiting

Exercícios

- B) explique a API do recurso semáforo introduzido na versão 5 do SDK da Sun
  - http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html
  - descrição da API
  - compare com semáforos
  - entre 1 e 2 páginas
- C) compare locks do SDK 5 com synchronized
  - idem

Revisão

Geral de Sincronização +

- **Revisão de sincronização + em Java threads**
  - Liste 3 novos mecanismos (tipos de classes) para sincronização
  - Qual a motivação para a sua introdução em Java? (2 a 3 motivos)

- **Revisão de mecanismos adicionais de sincronização**
  - Variáveis atômicas
    - Qual a função básica?
    - Qual a vantagem?
    - Um método especial ?
    - Onde não podem ser aplicadas?
  - Barreiras
    - Qual a função básica?
    - Qual o método principal?
    - Quantas threads fazem parte da barreira?
    - Uma barreira pode ser reusada? Como ou quando?

- **Revisão de mecanismos adicionais de sincronização**

- **Revisão de mecanismos adicionais de sincronização**
  - Trancas
    - Qual a principal diferença com relação a barreiras?
    - Quantas threads podem participar de uma barreira?
    - Quais os principais métodos de uma tranca?

- **Revisão de mecanismos adicionais de sincronização**
  - Locks
    - Liste e explique 2 diferenças em relação ao conceito de lock
  - Semáforos
    - Liste 2 métodos adicionais (ou variantes de) ao conceito de semáforos
    - Liste uma diferença (ou variação) no comportamento de semáforos

Revisão Java threads

- **Revisão de gerência + em Java threads**

Revisão Java threads

- **Revisão de gerência + em Java threads**

- **Bibliografia**
  - Lea, D. Concurrent Programming in Java - Design Principles and Patterns. Addison-Wesley, 1997.
  - Oaks, S. and Wong, H. Java Threads. O'Reilly, 1997.
  - Goetz, B. et al. Java Concurrency in Practice. Addison-Wesley, 2006.

- **Bibliografia (cont.)**
  - Schildt, H. Java The Complete Reference. McGraw-Hill, 2011.
  - Schildt, H. Java – a Beginner's Guide. McGraw-Hill, 2011.
  - Zakhour, S. The Java Tutorial: a Short Course on the Basics. Prentice-Hall, 2012.
  - Campione, Mary e Walrath, K. The Java Tutorial. Addison-Wesley, 2a. ed., 1998.
  - Cornell, Gary e Horstmann, Cay. Core Java. Prentice Hall, 2007.
  - Flanagan, David. Java in a Nutshell. O'Reilly Assoc., 2a. ed., 1997.

- **Bibliografia (cont.)**
  - Arnold, K. and Gosling, J. The Java Language. Addison-Wesley, 1996.
  - Orfali, R. and Harkey, D. Client/Server Programming with JAVA and CORBA. John Wiley, 1997.
  - Wutka, M. Java - Expert Solutions. Que, 1997.
  - Walnum, Clayton. Java by Examples. Que, 1996.

- **Bibliografia (cont.)**
  - Grand, Mark. Java Language Reference. O'Reilly Assoc., 2a. ed., 1997.
  - Niemeyer, P. e Peck, Josh. Exploring Java. O'Reilly Assoc., 2a. ed., 1997.

- **Endereços**
  - Site Oracle:
    - www.java.com
    - http://www.oracle.com/technetwork/java/index.html
    - http://www.oracle.com/technetwork/java/javase/documentation/index.html
    - http://www.oracle.com/technetwork/java/javase/documentation/tutorials-jsp-138802.html
    - http://docs.oracle.com/javase/tutorial/index.html
    - (acessados em 07/03/2012)

- **Endereços**
  - site da Sun sobre tecnologia Java
    - http://java.sun.com
  - notas técnicas
    - http://java.sun.com/jdc/tecDocs/newsletter/index.html
  - tutorial Java
    - http://javasoft.com/docs/books/tutorial/index.html
  - Documentação Java:
    - http://java.sun.com/docs/white/index.html

- **Endereços (cont.)**
  - tutor Java
    - http://www.mercury.com/java-tutor/
  - Java ensina Java
    - http://www.neca.com/~vmis/java.html
  - outro tutorial Java
    - http://www.phrantic.com/scoop/onjava.html

Java

Mecanismos de Sincronização Adicionais

- semáforos, locks, ...