Java

Threads: sincronização com monitores

- **Autor**
  - inicial: tradução da versão inglesa por aluno do CIC
  - revisões, adições: C. Geyer e diversos alunos
- **Local**
  - Instituto de Informática
  - UFRGS
  - disciplinas :
    - Programação Distribuída e Paralela (CIC e ECP)
    - Programação com Objetos Distribuídos (PPGC)
  - versão atual:
    - V30.5, março de 2017

- **Fontes principais**
  - Essas transparências foram inicialmente baseadas nas do curso em ingles
    - Java, by Dr. Stefan Schimpf e Heribert Schlebbe
  - as quais são baseadas nas do curso em alemão
    - Java, by Ralf Reissing and Hartrunt Keller, Institut für Informatik, Unioversität Stuttgart
- **Inúmeras revisões e adições foram feitas posteriormente**

Súmula

- **Súmula**
  - sincronização em Java
    - conceitos de monitores
    - exclusão mútua em Java threads
    - sinalização em Java threads
    - Exemplo
  - API Java threads ou relacionada
    - alguns recursos adicionais

- **Súmula**
  - Observações
    - Há outros arquivos de slides sobre Java threads
      - Outros mecanismos de sincronização como locks e semáforos
      - Interrupções

Bibliografia

Bibliografia

- Goetz, B. et al. Java Concurrency in Practice. Addison-Wesley, 2006.
  - Et al. => também Doug Lea
- Lea, D. Concurrent Programming in Java - Design Principles and Patterns. Addison-Wesley, 1999.
  - Ótimo livro com conceitos de concorrência e sincronização
  - Exemplos em Java
- Oaks, S. and Wong, H. Java Threads. O'Reilly, 2004.
  - Bom livro, técnico
- Farley, Jim. Java - Distributed Computing. Ed. O'Reilly, 1998.

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.
- Campione, Mary e Walrath, K. The Java Tutorial. Addison-Wesley, 2a. ed., 1998.
- Cornell, Gary e Horstmann, Cay. Core Java. Sunsoft Press., 1996.
- Flanagan, David. Java in a Nutshell. O'Reilly Assoc., 2a. ed., 1997.

- **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.

Links

- **Endereços**
  - Site Oracle:
    - www.java.com
      - inicial
    - http://www.oracle.com/technetwork/java/index.html
      - inicial
    - http://www.oracle.com/technetwork/java/javase/documentation/index.html
      - Documentação
    - (acessados em 02/03/2015)

- **Endereços**
  - Site Oracle:
    - http://www.oracle.com/technetwork/java/javase/documentation/tutorials-jsp-138802.html
      - tutoriais
    - http://docs.oracle.com/javase/tutorial/index.html
      - tutoriais
    - (acessados em 25/02/2014)

- **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
  - Acessados em mar 2016
    - Desviados para Oracle

- **Endereços (cont.)**
  - outro tutorial (?) Java
    - http://www.phrantic.com/scoop/onjava.html

Sincronização de Java Threads

Monitores

Conceitos de Monitores

Revisão

- **Sincronização**
  - conceito geral de monitores
    - um monitor consiste de:
      - estruturas de dados
      - coleção de procedures que operam sobre as estruturas do monitor
      - conceito de dados protegidos
        - as estruturas só podem ser acessadas pelas procedures do monitor

- **Sincronização**
  - conceito geral de monitores
    - exclusão mútua:
      - somente uma thread cliente pode executar uma procedure do monitor em um momento
      - gera fila de entrada no monitor
        - threads bloqueadas esperando sua vez
      - semântica implícita (sem uso de mutex, ...)

- **Sincronização**
  - conceito geral de monitores
    - varíaveis de condição
      - variáveis especiais sobre as quais podem ser chamadas as operações wait e signal
      - dentro do monitor

- **Sincronização**
  - conceito geral de monitores
    - operação wait:
      - coloca a própria thread em estado bloqueado (dormindo) numa lista de espera do monitor
      - isto permite a execução de outra thread
      - lista de espera: associada à variável de condição
      - vc1.wait()

- **Sincronização**
  - conceito geral de monitores
    - operação signal:
      - acorda uma outra thread que está na lista de espera associada à variável de condição
      - vc1.signal()
      - questão: qual thread continua sua execução imediatamente?
        - a que acorda?: mais eficiente
        - a acordada?: mais seguro

- **Sincronização**
  - Alguns termos usados pela Oracle (tutorial)
    - Interferência
      - Ocorre quando 2 operações, sendo executadas em threads distintas, atuando sobre o mesmo dado, têm seus passos entrelaçados
      - Exclusão mútua, seção crítica, ...
    - Erros de Consistência de Memória
      - Ocorre quando threads distintas têm visões distintas sobre o que deve ser o dado (valor)
      - Problemas de ordenamento em leitura / escrita
      - Sinalização

- **Sincronização**
  - Alguns termos usados pela Oracle (tutorial)
    - Starvation (“morte por fome”)
      - Uma thread não consegue acesso a um recurso compartilhado (variável) e não progride
    - Livelock
      - Duas threads repetidamente respondem a ações da outra (uma da outra) e não progridem
    - Blocos guardados
      - Uso de primitivas bloqueantes (“wait”) dentro de uma condição (if, while, ...)

Monitores em Java

Exclusão Mútua

- **Sincronização**
  - conceito geral de sincronização de threads (Java): monitores
    - Monitor em Java:
      - um objeto qualquer, compartilhado por várias threads (“servidor”), pode ser visto como um monitor
      - diversas diferenças com relação ao conceito clássico de monitor
        - semântica de wait e signal, ...

- **Sincronização: modificadores e métodos básicos**
  - modificador synchronized em método de instância
    - modifica método declarando que o mesmo faz parte de monitor
    - o monitor é o (um) objeto
    - impõe exclusão mútua (lock/unlock) na execução de todos os métodos com synchronized do objeto
      - métodos sem synchronized não verificam o lock
        - são executados concorrentemente entre si e com relação aos synchronized
    - existe um lock associado a cada objeto da classe

- **Sincronização: modificadores e métodos básicos**
  - synchronized em método de instância
    - importante:
      - o normal é execução concorrente dos métodos
      - para ter exclusão mútua, é preciso usar o synchronized

- **Sincronização: modificadores e métodos básicos**
  - synchronized em métodos estáticos
    - o lock é um objeto associado à classe
    - independente dos locks por objetos
    - métodos estáticos sem synchronized
      - idem métodos de instância

- **Sincronização: modificadores e métodos básicos**
  - deadlock
    - ocorre se (por exemplo)
      - thread A tem lock 1 (associado a objeto x)
      - thread B tem lock 2 (associado a objeto y)
      - thread A pede lock 2
        - chama método de y
      - thread B pede lock 1
        - chama método de x

- **Sincronização: modificadores e métodos básicos**
  - deadlock
    - não ocorre se
      - thread tem lock 1
      - chama outro método associado ao mesmo lock
        - método do mesmo objeto

- **Sincronização: modificadores e métodos básicos**
  - synchronized em bloco de comandos
    - esqueleto de código em método:

```
...
synchronized(objetoQualquer) {
   ...
   "bloco de comandos"
...
}
...
```

- **Sincronização: modificadores e métodos básicos**
  - synchronized em bloco de comandos
    - objetoQualquer funciona como um lock
    - todos os blocos synchronized com o mesmo objetoQualquer são executados de forma atômica
    - vantagem: seção crítica mais fina (menor) -> melhor desempenho

- **Sincronização: modificadores e métodos básicos**
  - synchronized em bloco de comandos
    - o objeto lock (objetoQualquer) pode ser um objeto externo ao objeto onde está inserido o código sincronizado
    - nesse caso, pode-se sincronizar blocos "independentes", em diferentes objetos e classes, mas compartilhando o mesmo lock
      - condição: passar o objeto lock a todos os objetos com o mesmo bloco
        - na construção de uma thread, por exemplo

- **Sincronização: modificadores e métodos básicos**
  - synchronized em bloco de comandos
    - tipo de variável do lock?
    - em métodos estáticos
      - com variável de classe
        - obrigatoriamente: método estático não tem acesso direto a variáveis de instâncias
      - ou objeto (variável de instância) passado como argumento
        - cuidar que deve ser um único objeto passado em todas as chamadas a sincronizar entre si

- **Sincronização: modificadores e métodos básicos**
  - synchronized em bloco de comandos
    - tipo de variável do lock?
    - em métodos de instância
      - com variável de classe
        - opcional: controla acesso a variáveis estáticas
      - com variável de instância
        - para controlar variáveis de classe: usar objeto lock único em todos os objetos

- **Sincronização: modificadores e métodos básicos**
  - synchronized em bloco de comandos
    - lock x objeto x variável
      - o lock está sempre associado a um objeto e não a uma variável (um nome simbólico em um único contexto - objeto)
      - um objeto pode estar sendo referenciado em *n* variáveis (usual em OO)

Sinalização em Java

- **Sincronização: modificadores e métodos básicos**
  - Variáveis de condição
    - Não existiam formalmente no início de Java
    - Criadas na década de 2000
  - Métodos de sinalização
    - wait, notify, ...
    - São métodos da classe Object
    - Classe Object: raiz da hierarquia de classes
    - Outros métodos: equals, finalize, ... (poucos)

- **Sincronização: modificadores e métodos básicos**
  - wait()
    - Chamada por thread que está com o lock
    - Semântica similar à do conceito de monitores
    - Bloqueio
      - a thread é bloqueada incondicionalmente
      - a thread é incluída na lista de espera do monitor (lock)
      - libera lock permitindo a execução de outra thread que espera pelo lock

- **Sincronização: modificadores e métodos básicos**
  - wait()
    - Retorno à execução
      - Outra thread chama método *notify* ou *notifyall* no mesmo lock (ver semântica dos *notify*)
      - Outra thread interrompe (método *interrupt()*) essa thread
      - Time-out ocorre (caso especial do *wait*)
      - Por motivo não previsto (erro? “spurious wake-up”)

- **Sincronização: modificadores e métodos básicos**
  - notify()
    - caso exista, escolhe-se uma thread de forma arbitrária, remove-se a thread da fila associada (lock) e coloca-se a thread em fila de ready
    - a thread corrente não perde o lock (estado running)
    - a thread deve obter o lock para continuar sua execução
      - em algum momento posterior
      - nesse momento o lock (certamente) está com a thread que executou o notify,
      - entretanto uma outra thread pode obter antes o lock

- **Sincronização: modificadores e métodos básicos**
  - notify()
    - Documentação Sun da API:
      - “The choice is arbitrary and occurs at the discretion of the implementation.”
      - “The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.”
      - “The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object”
      - “For example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.”

- **Sincronização: modificadores e métodos básicos**
  - notifyall
    - Diferença para notify:
      - acorda todas as threads bloqueadas na fila do lock associado ao notifyall
  - wait
    - duas versões com time-out
    - passado o tempo, a thread (ou threads) é acordada automaticamente
  - wait e notify
    - runtime verifica se notify e wait estão dentro do lock (bloco ou método inteiro sincronizado)
      - não verifica se o lock é externo

- **Sincronização: modificadores e métodos básicos**
  - wait/notify usados em métodos estáticos
    - wait/notify são métodos não estáticos e finais
      - não podem ser reescritos
      - não podem ser chamados pelo nome da classe
    - alternativa em bloco synchronized em método estático
      - usar um objeto comum a toda a classe
        - Uma variável do tipo Class, inicializado com o objeto da classe específica
        - Class c = AlgumaClasse.class
      - um objeto único passado em todas as chamadas dos métodos estáticos (argumento)

Algumas Técnicas de Sincronização em Java

Considerações sobre o uso de sincronização

- **Considerações sobre o uso de sincronização**
  - desenvolvimento conservativo
    - inicialmente coloque synchronized em todos os métodos das classes com objetos compartilhados
    - teste e verifique a correção do programa
    - teste e analise o desempenho do programa
    - se desempenho “ruim”
      - reexamine o código procurando
        - métodos que não precisem de synchronized
        - seções críticas mais finas que um método, isto é, uso de bloco synchronized

Exemplos de Sincronização

2 versões do problema do Produtor Consumidor

Ou do Buffer

Exemplo Buffer

- **Exemplo Buffer**
  - 2 soluções são apresentadas
  - Somente o código do buffer como um monitor
  - Solução A:
    - Do 1º livro sobre Java (Arnold e Gosling, ...)
    - Mais abstrata
  - Solução B:
    - Do livro do Doug Lea
    - Código completo: pode ser compilado e usado

Exemplo Buffer

- **Exemplo A:**
  - classe que implementa uma fila
  - conceito de fila: ordem FIFO: First In, First Out
    - inclusão (append): no fim (tail) da fila
    - retirada (get): do início (head) da fila
  - fila monothread
    - se a fila está vazia em retirada: retorna um aviso
    - se a fila está cheia em inclusão: idem

Exemplo Buffer

- **Exemplo: classe que implementa uma fila**
  - fila multithread
    - usuário ou cliente (processo, thread): espera em caso de exceção
      - fila vazia em retirada: até haver inclusão
      - fila cheia: considera-se buffer ilimitado
    - concorrência possível nos acessos a fila: head, tail, ...
      - várias threads inserindo e excluindo elementos

Exemplo Buffer

- **Exemplo: classe que implementa uma fila (cont.)**
  - fonte: Arnold, K. and Gosling, J. The Java Language. Addison-Wesley, 1996.
  - classe Queue {
    // first and last element of the queue
    // structure: value of element and pointer to next elem.
    Element head, tail;

Exemplo Buffer

- **Exemplo: classe que implementa uma fila (cont.)**
  - // inserts element at the end of the queue

```
public synchronized void append(Element p) {

   if (tail == null)  // if queue is empty: simple case
     head = p;    // first element is the new element
  else
     tail.next  = p;  // last elem. points to new element

 p.next = null;   // next of last element is null
 tail = p;            // last element is the new element
 notify();     // notify that 1 element arrived
 }
```

Exemplo Buffer

- **Exemplo: classe que implementa uma fila (cont.)**
  - // gets an element of queue
    public synchronized Element get() {

```
try {
  while (head == null)   // if  queue is empty?
          wait();        // wait for an element
} catch (InterruptedException e) {
  return null;
}
```

Exemplo Buffer

- **Exemplo: classe que implementa uma fila (cont.)**
  - ```
    Element p = head;    // save first element
    head = head.next;    // take out of  queue
    if (head == null)  // queue is empty?
        tail = null;      // last element null
    return p;
  }
}
```
```

Buffer: Exemplo B

Doug Lea

Exemplo Buffer

- **Exemplo Buffer**
  - solução com buffer em array
  - fonte: Lea, D. Concurrent Programming in Java. Addison-Wesley, 1997.
  - solução similar à usada em livros clássicos sobre concorrência
  - class BoundedBufferWithStateTracking {

```
protected final Object[]  array;     // the elements
protected int putPtr = 0;                  // circular indices
protected int takePtr = 0;
protected int usedSlots = 0;         // the count
```

Exemplo Buffer

- public BoundedBufferWithStateTracking(int capacity)
  throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array = new Object[capacity];
  }

  public synchronized int size() { return usedSlots; }

  public int capacity() { return array.length; }

Exemplo Buffer

- **Exemplo Buffer**
  - public synchronized void put(Object x)
    throws InterruptedException {

    while (usedSlots == array.length) // wait until not full
      wait();

    array[putPtr] = x
    putPtr = (putPtr + 1) % array.length; // cyclically inc

    if (usedSlots++ == 0) // signal if was empty
      notifyAll();
    }

Exemplo Buffer

- **Exemplo Buffer**
  - public synchronized Object take()
    throws InterruptedException{

```
while (usedSlots == 0)                  // wait until not empty
  wait();

Object x = array[takePtr];
array[takePtr] = null;
takePtr = (takePtr + 1)       % array.length;
```

Exemplo Buffer

- **Exemplo Buffer**
  - public synchronized Object take()
    throws InterruptedException{

    …

    if (usedSlots-- == array.length) // signal if was full
      notifyAll();
    return x;
  }
}

Exercícios

- **Exercício A**
  - explique eventuais diferenças de semântica (execução) caso a condição de wait seja feita por um if
- **Exercício B**
  - explique (justifique ou não) se o programa funciona corretamente para dois ou mais produtores ou consumidores
- **Exercício C**
  - faça um pseudo código de um programa com 1 produtor, 1 consumidor e um buffer

Controle de Java Threads

API: outros métodos

- **Controle de threads: outros métodos nativos**
  - observações
    - esses métodos são em geral chamados por outra thread (objeto) fazendo referência a uma segunda thread (objeto)
      - exemplo: em thread a sobre thread b
        b.stop();
  - start()
    - inicia execução da thread principal (run) do objeto
    - JVM chama o método run()

Controle de threads

- **Controle de threads: outros métodos nativos**
  - stop()
    - força a própria thread a terminar sua execução
  - isAlive()
    - testa se a thread foi iniciada e ainda não terminou
  - suspend()
    - suspende a thread até ser reativada (*resume()*)

- **Controle de threads: outros métodos nativos (cont.)**
  - resume()
    - reativa thread suspensa anteriormente
  - sleep(long ml)
    - a thread é colocada em espera (“dormindo”) por ml milisegundos
    - a thread não abandona o monitor (synchronized)

- **Controle de threads: outros métodos nativos**
  - os métodos abaixo foram *deprecated*
    - *stop(), suspend(), resume()*
    - podem provocar erros de programação de difícil depuração
    - maiores detalhes em
      - http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

- **Controle de threads: outros métodos nativos**
  - os métodos abaixo foram *deprecated*
    - por exemplo
      - ao ser morta (*stop()*) uma thread deveria liberar todos os monitores (locks)
      - um desses monitores pode estar em um estado inconsistente
        - variáveis com estado inconsistente
      - o programador poderia desenvolver código para corrigir o estado das variáveis
        - tarefa que pode ser extremamente complexa

Controle de threads

- **Threads e o coletor de lixo (garbage collector)**
  - objeto normal
    - coletado quando não houver nenhuma referência ao mesmo
  - objeto Thread
    - coletado quando
      - não houver nenhuma referência ao mesmo
      - **E** execução da thread tiver terminado

- **Outros tópicos**
  - método currentThread
  - não usar dados public ou protected
  - herança x sync (Arnold pg173)
  - uso dado sincronizado como lock (Arnold pg174)
  - tornar classe sincronizada (Arnold pg175)

- **Outros tópicos**
  - wait x while x if (Arnold pg175)
  - wait, notify x sync (Arnold pg176)
  - notify x notifyAll (Arnold pg177)
  - join (Arnold pg184)

- **Outros tópicos**
  - volatile (Arnold pg188)
  - conceitos de grupos (Arnold pg188 e Evandro)
  - prioridades (Arnold pg188 e alunos)
  - término de thread em loop (Arnold pg183)
  - término de aplicação x setDeamon (Arnold pg185)
  - depuração (Arnold pg192)

Outras Formas de Concorrência

Timer Tasks

- **Timer Tasks**
  - classe Timer
    - objetos definem tempos
  - classe TimerTask
    - objeto define uma tarefa com restrições de tempo
    - algumas variações de restrições
      - após certo tempo
      - periódico
      - em determinado dia e hora

- **Timer Tasks**
  - métodos de fim de uma tarefa
    - por método *cancel()*
    - tornando a tarefa em um deamon
    - remoção de todas as referências à tarefa após o final da mesma (método usual)
    - pelo método *System.exit()*
      - encerra o programa

- **Timer Tasks**
  - fim de programa X tarefas (threads)
    - outras threads podem impedir fim de programa
    - exemplo:
      - função beep do toolkit AWT
    - para terminar o programa, pode ser necessário uso do *System.exit()*

Sincronização de Java Threads

Outros Recursos: locks, semáforos,…

Novos Recursos SDK 5

- **Novos recursos na versão 5 do SDK da Sun**
  - http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html
  - pacote java.util.concurrent
  - diversas novas classes, interfaces, exceções, ..., para programação concorrente
  - alguns recursos
    - classe semáforo
    - locks (mais flexíveis que synchronized)
    - Barreiras
  - Obs.: detalhamento em outros slides

- **Exercícios**
  - A) Exercício 1 - Projeto de Desenvolvimento de Mecanismos de Sincronização
    - Descrição: Desenhe um projeto (especificação) da implementação de um mecanismo adicional de sincronização usando somente "synchronized" (modificador ou bloco) e wait/notify; escolha uma das opções abaixo.
      - A: lock
      - B: semáforo
      - C: variáveis de condição

- **Exercícios**
  - A) Exercício 1 - Projeto de Desenvolvimento de Mecanismos de Sincronização
    - Dicas:
      - não é necessário implementar integralmente, somente um código abstrato
      - nas pastas de slides da disciplina INF01151 - SOII (ftp), há slides sobre esses mecanismos

- **Resumo Java Monitores**
  - Uso de recursos de linguagens para exclusão mútua
    - Modificador de método
    - Bloco synchronized
  - Qualquer classe pode ser (transformada em) monitor
  - Filas não fifo
  - Sem variáveis de condição
  - Notify: mantém lock
  - Wait: indefinido momento em que recupera lock (estado running)

Revisão

- **Revisão de sincronização**
  - qual a semântica (comportamento) de um objeto “normal” acessado por múltiplas threads concorrentemente?
  - quais as questões a serem verificadas e resolvidas pelo programador desse objeto?
  - qual o recurso básico em Java para obter-se exclusão mútua?
  - qual a semântica desse recurso quando usado de forma básica em objetos?
    - em especial, quais os objetos e métodos afetados?
  - o que é deadlock (bloqueio perpétuo) e como pode ocorrer em Java no contexto acima?

- **Revisão de sincronização**
  - Synchronized em métodos estáticos
    - Qual o contexto da exclusão mútua?
    - Qual é o lock implícito?
    - Qual o contexto se em método estático e em método de instância?

- **Revisão de sincronização**
  - como pode obter-se uma seção crítica (exclusão mútua) de grão mais fino (mais eficiente)?
  - o que é o lock nesses casos?
  - quais os recursos básicos para sinalização entre threads em Java?
  - onde ficam as threads bloqueadas?
  - quem é acordada?
  - quem recebe o lock após uma thread ser acordada?

- **Revisão de sincronização**
  - os mecanismos acima tem relação com qual mecanismo de sincronização clássico?
  - qual a diferença básica na exclusão mútua?
  - quais as diferenças principais com relação à sinalização?
  - Cite alguns mecanismos de sincronização em Java?
  - Porque / como decidir quais usar?