Concorrência e Tratamento de Exceções

Paradigmas de Programação – BCC/UFRPE
Lucas Albertins – lucas.albertins@deinfo.ufrpe.br

Agenda

- Introdução – Concorrência
- Conceitos Básicos – Concorrência
- Métodos de Sincronização (Semáforos, Monitores e Troca de Mensagem)
- Tratamento de Exceções
- Conceitos Básicos – Concorrência
- Exemplos C++ e Java

Introdução

- Níveis de Concorrência
  - Instrução de Máquina
    - Duas ou mais instruções simultaneamente – Multicore processing
  - Comando em linguagem de alto nível
    - Dois ou mais comandos simultaneamente
  - Unidade
    - Dois ou mais subprogramas
  - Programa

Introdução

- Níveis de Concorrência
  - Instrução de Máquina
    - Duas ou mais instruções simultaneamente – Multicore processing
  - **Comando em linguagem de alto nível**
    - **Dois ou mais comandos simultaneamente**
  - **Unidade**
    - **Dois ou mais subprogramas**
  - Programa

Categorias de Concorrência

- Categorias
  - Concorrência física – Múltiplos processadores independentes (múltiplas threads de controle)
  - Concorrência lógica – A aparência de concorrência física é apresentada por compartilhamento de tempo de um único processador (software projetado como se houvesse múltiplas threads de controle)
- Uma thread de controle em um programa é uma sequencia de execução como um fluxo de controle

Motivações para Concorrência

- Computadores com multiprocessadores permitindo concorrência física
- Até mesmo um único processador com concorrência pode ser mais eficiente do que sem concorrência
- Projetar software de uma forma diferente, mas que representa várias situações do mundo real
- Várias aplicações podem se espalhar em várias máquinas locais ou espalhadas em uma rede

Conceitos básicos

- Tarefa, Processo ou Thread é uma unidade de programa que pode estar em execução concorrente com outras
  - Quando uma unidade inicia uma thread ela não precisa necessariamente ser suspensa
  - Quando a thread termina o controle pode não retornar para o invocador
  - Pode se comunicar com outras através de variáveis não locais, troca de mensagem ou parâmetros
- Multi-thread

Conceitos básicos

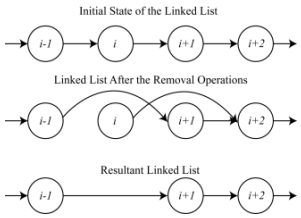
- Execução Paralela de dois ou mais processos
- Acesso sincronizado a dado compartilhado através de exclusão mútua entre processos
- Abstrações de controle concorrente
- Sincronização entre processos
- Transferência de dados sincronizada através de comunicação entre processos

Execução Paralela

- Principal diferença entre programação sequencial e concorrente
- Consequências:
  - A possibilidade de operações de atualização em variáveis pode falhar em produzir resultados válidos
  - Perda de determinismo
  - Exemplos
    - l := true | l := false | h := l
    - n = 7 | n = n + 1;
  - Tipos de erros mais difíceis de se detectar

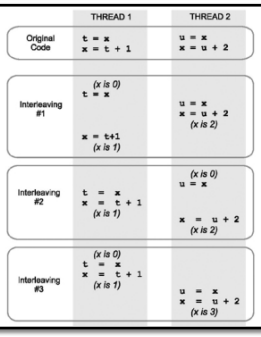
Exclusão Mútua

- Remoção Simultânea de uma lista ligada



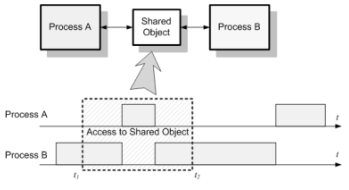
Exclusão Mútua

- Condição de Corrida (*Race Condition)*
- Qual das possibilidades é o desejado?



Exclusão Mútua

- Restaura a semântica de acesso e atualização a variáveis
- Realizada através de sincronizações que são custosas, mas que devem ser consistentes



Abstrações de controle concorrente

- Promovem programação concorrente confiável colocando o fardo de sincronizações dentro das linguagens de programação
- Exemplos:
  - Região Crítica condicional

```
region v when B do
    begin
          ...
    end;
```

  - Monitores

Tipos de Sincronização

- Sincronização controla a ordem de execução das tarefas
  - Cooperação: quando A deve esperar B completar alguma tarefa específica
    - Ex: Produtor-Consumidor
  - Competição: Quando A e B necessitam do uso de um recurso específico
    - Suponha que A deve adicionar 1 a *recurso* que começa com 3 e B multiplica o valor de *recurso* por 2
      - *total = 3*
      - A: *total+= 1*
      - B: *total*=2*
      - Quais os possíveis valores de total?

Conceitos básicos

- *Liveness* – propriedade que indica que uma unidade completa sua execução em algum momento
  - Em um ambiente concorrente, tarefas podem facilmente não possuir a propriedade de *liveness*
- Se todas as tarefas em um ambiente concorrente perdem a propriedade de *liveness*, acontece um *deadlock*
- *Jantar dos filósofos*

Métodos de Sincronização

- Semáforos (Dijskstra 1965)
- Monitores
- Troca de Mensagem

Semáforos

- Estrutura de dados com um contador e uma fila que armazena descritores de tarefas
- Implementam guardas no código que acessa estruturas de dados compartilhadas
- Duas operações: *wait* e *release*
- Sincronização tanto em competição quanto cooperação

Sincronização em cooperação com Semáforos

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
      loop
      -- produce VALUE --
      wait (emptyspots); {wait for space}
      DEPOSIT(VALUE);
      release(fullspots); {increase filled}
      end loop;
end producer;
task consumer;
      loop
      wait (fullspots);{wait till not empty}}
      FETCH(VALUE);
      release(emptyspots); {increase empty}
      -- consume VALUE --
      end loop;
end consumer
```

Sincronização em competição com semáforos

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
    loop
    -- produce VALUE --
    wait(emptyspots); {wait for space}
    wait(access);     {wait for access)
    DEPOSIT(VALUE);
    release(access); {relinquish access}
    release(fullspots); {increase filled}
    end loop;
end producer;
```

Sincronização em competição com semáforos

```
task consumer;
    loop
    wait(fullspots);{wait till not empty}
    wait(access);   {wait for access}
    FETCH(VALUE);
    release(access); {relinquish access}
    release(emptyspots); {increase empty}
    -- consume VALUE --
    end loop;
end consumer;
```

Monitores

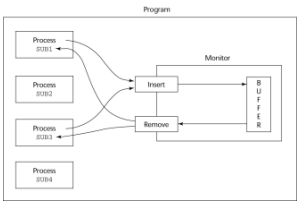
- Ada, Java, C#
- Encapsula o dado compartilhado e suas operações para restringir acesso
- Um monitor é um tipo abstrato de dados para um dado compartilhado

Competição em Monitores

- Dado compartilhado é residente dentro do monitor
- Todos os acessos residentes no monitor
  - Implementação do monitor garante acesso sincronizado permitindo um único acesso por vez
  - Chamadas ao monitor são implicitamente enfileiradas se o monitor estiver ocupado no momento da chamada

Cooperação em Monitores

- Cooperação entre processo é uma tarefa de programação
  - Programador deve garantir que um buffer compartilhado não tenha underflow ou overflow



Troca de Messagem

- Modelo geral para concorrência (pode modelar tanto semáforos quanto monitores)
- Ideia Central: visita a um consultório médico – paciente espera pelo médico ou o médico espera pelo paciente, quando ambos estão prontos, há um encontro (*rendezvous*)
- Quando uma mensagem do transmissor é aceita por um receptor, a transmissão da mensagem é chamada de *Rendezvous*
  - Precisa de um mecanismo que permita uma tarefa indicar quando ela deseja aceitar mensagens
  - Precisa de uma forma de lembrar quem está esperando que suas mensagens sejam aceitas de forma justa

Exemplo CSP

Threads em Java

- Unidades de concorrência em Java são métodos chamados *run*
  - Um método *run* pode estar em execução concorrente com outros métodos
  - O processo no qual os métodos *run* executam é chamado de *thread*

```
class myThread extends Thread
   public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Threads em java

- A classe Thread em Java tem várias formas de controle de execução
  - *yield* –coloca a thread na pilha de execução do processador
  - *sleep* –bloqueia a thread
  - *join* – força um atraso na execução do método até o método run de outra thread termine sua execução
- Threads também podem ter prioridades que podem ser alteradas com o método *setPriority*

Competição em Java

- Um método que inclua o modificador `synchronized` não permite qualquer outro método executar enquanto o objeto estiver em execução

```
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
```

- Os dois métodos acima são sincronizados o que os previne de interferir um com outro
- Se somente parte de um método deve executar sem interferência, ele pode ser sincronizado com o comando `synchronized`

```
synchronized (expression)
   statement
```

Cooperação em Java

- Métodos `wait`, `notify`, e `notifyAll`
- `wait` coloca o objeto em uma fila de espera
- `notify` avisa quem estiver esperando (wait) que o evento aconteceu
- `notifyAll` acorda todas as threads na lista de espera
- Também pode ser usada a classe Semaphore para utilizar semáforos

Tratamento de Exceções

- Linguagem sem tratamento de exceção
  - Quando a exceção ocorre, controle vai para o sistema operacional, uma mensagem é mostrada e o programa termina
- Linguagem com tratamento de exceção
  - Programas podem capturar algumas exceções, fornecendo assim a possibilidade de resolver o problema e continuar

Conceitos Básicos

- **Exceção** (*Exception*) é qualquer evento estranho, seja um erro ou não, detectável pelo hardware ou software, que requer algum processamento especial
- O tratamento especial requerido após a detecção da exceção é chamado tratamento de exceção
- O código que trata uma exceção é chamado de tratador da exceção (*Exception Handler)*

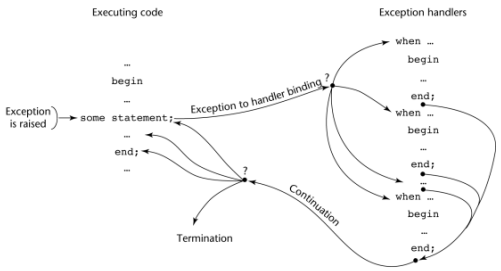
Conceitos Básicos

- Uma linguagem sem suporte a tratamento de exceção ainda pode tratar exceções por mecanismos de software construídos pelo usuário
- Dentre as vantagens de já possuir tratamento de exceções em linguagens podemos citar
  - Escrever código de detecção de erro é tedioso e bagunça o programa
  - Tratamento de erro encoraja programadores a considerar várias possibilidades de erro
  - Reuso de código para tratamento de exceção

Decisões de projeto

- Como e onde os tratadores de exceção são especificados e qual é o seu escopo?
- Como uma exceção é ligada ao seu tratamento?
- Informações sobre exceção podem ser passadas para o tratador da exceção?
- Onde a execução continua? (local da exceção ou após o tratamento)
- Como exceções definidas pelo usuário podem ser especificadas?
- Exceções podem ser desabilitadas?

Fluxo de controle de tratamento de exceção



Tratamento de Exceção em C++

- Forma do tratamento de exceção

```
try {
   -- código esperado para lançar a exceção
   }
   catch (parametro formal) {
   -- código do tratador da exceção (exception handler)
   }
   ...
   catch (parametro formal) {
   -- código do tratador da exceção (exception handler)
   }
```

Lançando Exceções

- Exceções podem ser lançadas explicitamente com o comando:
  `throw [expression];`
- Um **`throw`** sem operando pode aparece apenas no tratador da exceção, o que faz a exceção ser lançada novamente
- Exceções não capturadas são propagadas para o invocador da função na qual ela foi chamada até a função main
- Após o *handler* terminar sua execução o fluxo de controle vai para o primeiro comando após o último *handler*

Avaliação

- Exceções não tem nome
- Exceções detectadas por hardware não podem ser tratadas pelo sistema
- Ligar exceções a *handlers* através de tipos não ajuda na legibilidade do programa
- O parâmetro da exceção provê uma forma de passar informação para o tratamento da exceção

Tratamento de Exceção em Java

- Baseado em C++, no entanto mais alinhado com a filosofia OO
- Todas as exceções são objetos de classes descendentes da classe **Throwable**, a qual tem duas subclasses
  - **Error**:
    - lançada pelo interpretador Java para eventos internos (e.g., heap overflow) nunca tratados por programas usuários
  - **Exception**:
    - Exceções definidas pelo usuário
    - Exemplos de sublcasses: **IOException, RuntimeException, ArrayIndexOutOfBoundsException** e **NullPointerException**

Tratamento de Exceção em Java

- O parâmetro do catch deve ser um descendente de **Throwable**
- Exceções são lançadas com **throw**, incluindo um objeto **Throwable** (`throw new MyException();`)
- Vinculação da exceção com primeiro tratador cuja classe é a mesma da exceção
- Se nenhum tratador for encontrado a exceção é propagada para o main como em C++

Exceções checadas e não checadas

- Exceções das classes **Error** e **RunTimeException** e todos os seus descendentes são chamadas exceções não checadas
- Exceções checadas que podem ser lançadas por um método devem:
  - Ser listadas na cláusula **throws** do método, ou
  - Ser tratada no método
- Quais são as alternativas de um método que invoca um outro método que lista exceções checáveis na sua cláusula **throws**?

A cláusula **finally**

- Pode aparecer no final de um construtor **try**
- Forma

```
finally {
...
}
```

- Propósito: Especificar código que deve ser executado independente do que acontece dentro de um construtor **try**

Avaliação

- Os tipos das exceções fazem mais sentido do que em C++
- A cláusula **throws** é melhor do que em C++ pois diz mais informação
- A cláusula **finally** é bastante útil
- O interpretador Java lança diversas exceções que podem ser tratadas pelo programas definidos pelo usuário

Leitura Adicional

- Capítulo 13 e 14 – Concorrência e Tratamento de Exceções. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011
- Próxima aula
  - Revisão para a prova da 1a VA