MAC0438 - Programação Concorrente

Daniel Macêdo Batista

IME - USP, 8 de Março de 2013

Roteiro

Lembretes

Mais sobre histórias e propriedades

Mais sobre ações atômicas

Comando `await`

Justiça e escalonamento

Travas e barreiras de sincronização

Lembretes

Meus TODOs

- ☐ Monitora definida: Patricia Araújo (mestranda)

Mais sobre histórias e propriedades

Tarefa extra

- Se um programa concorrente possui n processos e cada processo executa m ações atômicas, quantas histórias diferentes podem existir?
- $(n \times m)!/(m!^n)$

Propriedades

- Propriedade: atributo que é verdade para todas as histórias de um programa concorrente
  - Propriedades tipo Safety: O programa nunca entra em um estado "ruim"
  - Propriedades tipo Liveness: O programa eventualmente entra em um estado "bom"

Propriedades safety

- Corretude parcial
  - O programa termina **e** o estado final é correto (bom)
  - E se o programa entra em loop infinito?
- Exclusão mútua

Propriedades Liveness

- Conclusão
  - Todos os loops e chamadas de procedimento terminam
  - i.e. o tamanho de todas as histórias é finito
  - E se o programa termina com estado incorreto? (ruim)
- Entrada eventual (no acesso à seção crítica)

Como verificar as propriedades?

- Ficar rodando num loop infinito é suficiente?
- Outras formas?

Mais sobre ações atômicas

Equivalência a ações atômicas

- expressão e não referencia variáveis alteradas por outro processo
- o mesmo vale para a atribuição `x = e`
- Mas isso não costuma acontecer em programação concorrente

Propriedade No Máximo Uma Vez (PNMUV)

- **Referência Crítica** é quando há referência a uma variável que é modificada por outro processo (variáveis "simples" comentadas anteriormente)
- Uma expressão `x = e` satisfaz **PNMUV** se:
  - **e** contém no máximo 1 referência crítica e `x` não é lido por outro processo
  - **e** não contém referências críticas

- A expressão será equivalente a uma ação atômica
- Resultados intermediários não serão vistos pelos processos

PNMUV


```
int x = 0, y = 0;
co x = x + 1;
// y = y + 1;
oc;
```

- □ Respeita a PNMUV?

PNMUV


```
int x = 0, y = 0;
co x = y + 1;
// y = y + 1;
oc;
```

- Quais os valores finais de x e y?
- Respeita a PNMUV? Ou seja, há necessidade de forçar as atribuições a serem atômicas?

PNMUV


```
int x = 0, y = 0;
co x = y + 1;
// y = x + 1;
oc;
```

- Respeita a PNMUV?
- Quais os valores finais de x e y?
- Como garantir a execução "correta" dos processos sem a visão de resultados intermediários

Comando `await`

O que fazer com expressões que não respeitam a PNMUV?

- Precisamos de uma forma de definí-las como atômicas
- Exemplo do algoritmo para encontrar o máximo
  - Usamos < e >

Exemplos

- Uma base de dados com duas variáveis `x` e `y` que **sempre** precisam ter os valores iguais
- Uma lista encadeada que na remoção e inserção precisa atualizar dois ponteiros (anterior e o fim da lista)
- Sem garantir que as ações sejam atômicas, execuções concorrentes nesses exemplos podem levar a inconsistências

Formas de utilizar o `await`

- Para sincronização por condição basta: `<await (B);>`
  - `<await (count > 0);>`
- Como implementar com `while` se B respeitar a PNMUV?

Implementando o `await`

- Se `B` respeita a PNMUV
  - `<await (B);>` $\rightarrow$ `while (!B)`
  - Qual o problema?

Produtor/Consumidor com `await`

- Similar ao problema do grep
- A ideia é copiar o conteúdo de um vetor para outro

```
int buf, p = 0, c = 0;
process Produtor {
  int a[n];
  while (p<n) {
     < ... >
     buf = a[p];
     p = p +1;
  }
}
```

Produtor/Consumidor com `await`


```
process Consumidor {
  int b[n];
  while (c<n) {
     < ... >
     b[c] = buf;
     c = c + 1;
  }
}
```

Produtor/Consumidor com `await`


```
int buf, p = 0, c = 0;
process Produtor {
  int a[n];
  while (p < n) {
     <await (p == c);>
     buf = a[p];
     p = p +1;
  }
}

process Consumidor {
  int b[n];
  while (c < n) {
     <await (p > c);>
     b[c] = buf;
     c = c + 1;
  }
}
```

Justiça e escalonamento

Relembrando propriedades liveness

- "O programa alguma hora entrará em um estado bom"
- O programa abaixo termina? (Bom = O programa vai terminar)

```
bool continue = true;

co while (continue);
// continue = false;
oc
```

- Suponha que o escalonador escalona um processador por processo até que ele termine e só há um processador

Justiça incondicional

- Uma política de escalonamento segue a justiça incondicional se cada ação atômica incondicional elegível é executada alguma hora

```
bool continue = true;

co while (continue);
// continue = false;
oc
```

- Round-robin respeitaria para 1 processador
- Execução paralela respeitaria para múltiplos processadores
- E se houver um await?

Justiça fraca

- Uma política de escalonamento segue a justiça fraca se:
  1. Segue a justiça incondicional
  2. Cada ação atômica condicional elegível é executada alguma hora, desde que sua condição (B) torne-se verdade e permaneça verdade até ser vista pelo processo que executa a ação atômica condicional

```
<await (B) S;>
```

- Round-robin respeitaria se cada processo tivesse chance de executar
- E se a condição fica sempre mudando entre verdade e falso?

Justiça forte

- Uma política de escalonamento segua justiça forte se:
  1. Segue a justiça incondicional
  2. Cada ação atômica condicional elegível é executada alguma hora, desde que sua condição (`B`) permaneça verdade uma quantidade infinita de vezes
- Obs.: quantidade infinita de vezes = a condição é verdade um número infinito de vezes em cada história de um programa que não termina
- Em outras palavras: o processo com await tem que ser selecionado quando `B` for verdade

Justiça fraca X Justiça forte


```
bool continue = true, try = false;

co while (continue) {try = true; try = false;}
// <await (try) continue = false;>
oc
```

- O programa termina com justiça forte?
- O programa termina com justiça fraca?
- Como implementar uma política de escalonamento com justiça forte? Ideias?

Travas e barreiras de sincronização

O problema da seção crítica

- n processos repetidamente executam uma seção de código crítica e uma seção de código não crítica

```
process SC[i=1 to n] {
  while (true) {
     protocolo de entrada;
     secao critica;
     protocolo de saida;
     secao nao critica
  }
}
```

- Considerando que um processo que entra na seção crítica, sairá alguma hora

O problema da seção crítica

- Os protocolos precisam respeitar essas 4 propriedades:
  1. Exclusão mútua
  2. Ausência de deadlock (livelock)
  3. Ausência de espera desnecessária
  4. Entrada garantida

Solucionando o problema da SC para 2 processos


```
bool in1 = false; in2 = false;

process cs1 {
  while (true) {
     <await (!in2) in1 = true;>
     secao critica;
     in1 = false;
     secao nao critica;
  }
}
process cs2 {
  while (true) {
     <await (!in1) in2 = true;>
     secao critica;
     in2 = false;
     secao nao critica;
  }
}
```

- Respeita as 4 propriedades?