MAC0438 - Programação Concorrente

Daniel Macêdo Batista

IME - USP, 12 de Março de 2013

Roteiro

Travas e barreiras de sincronização

Ideias para implementar o await

Travas e barreiras de sincronização

Ideias para implementar o await

Travas e barreiras de sincronização
Ideias para implementar o await

Travas e barreiras de sincronização

Solução trivial

- Usar `await`'s incondicionais na seção crítica inteira
  - Garante exclusão mútua por definição
  - Garante ausência de deadlock, ausência de atrasos desnecessários e entrada garantida se o escalonamento for incondicionalmente justo?
  - Mas, como implementar $<$ e $>$?

Segunda solução

- Vamos começar usando $<$ e $>$ mas não na seção crítica toda
- Focar em tentar garantir primeiro a exclusão mútua. Depois as outras propriedades
- Como especificar exclusão mútua?
  - Como indicar que um processo está na seção crítica?
  - Primeiro com dois processos `CS1` e `CS2`

Segunda solução

- `in1` e `in2` são variáveis booleanas inicialmente `falso`
- Se o processo `CS1`(`CS2`) está na seção crítica, `in1`(`in2`) vale `verdade`
- Queremos evitar situações onde `in1` e `in2` sejam `verdade`
- Em outras palavras: antes de `CS1` entrar na seção crítica e fazer `in1 verdade`, ele precisa ter certeza que `in2` é falso
  - Como representar usando uma condição atômica condicional?

Segunda solução

Travas e barreiras de sincronização

Ideias para implementar o await

```
<await (!in2) in1 = true;>
```

Segunda solução


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

process CS1 {
   while (true) {
      <await (!in2) in1 = true;> /* prot. entrada */
      secao critica;
      in1=false; /* prot. saida */
      secao nao critica;
   }
}

process CS2 {
   while (true) {
      <await (!in1) in2 = true;> /* prot. entrada */
      secao critica;
      in2=false; /* prot. saida */
      secao nao critica;
   }
}
```

Segunda solução

- Exclusão mútua: sim, por construção
- Ausência de deadlock: sim, porque `in1` e `in2` nunca serão `verdade` ao mesmo tempo
- Ausência de atrasos desnecessários: sim, porque o processo não é impedido de entrar na seção crítica se o outro está fora dela
- Entrada garantida: não! Qual o motivo?

Terceira solução

- Como generalizar o caso anterior para n processos?

Terceira solução

- Spin locks
- O caso anterior precisaria de n variáveis para n processos
- Mas só existem duas situações de interesse:
  - algum processo está na seção crítica; ou
  - nenhum processo está na seção crítica
- Basta 1 variável

```
lock = (in1 || in2)
```

Terceira solução


```
bool lock=false;

process CS1 {
   while (true) {
      <await (!lock) lock = true;> /* prot. entrada */
      secao critica;
      lock=false; /* prot. saida */
      secao nao critica;
   }
}

process CS2 {
   while (true) {
      <await (!lock) lock = true;> /* prot. entrada */
      secao critica;
      lock=false; /* prot. saida */
      secao nao critica;
   }
}
```

Quarta solução

- Sugestões para tornar o último algoritmo possível?
- Quando a gente usa $<$ e $>$ a gente está roubando!

Quarta solução

- Queremos implementar os algoritmos

  Substituir os $< >$ por instrução que compare e atribua de forma atômica

- A maioria dos conjuntos de instrução de processadores possui instrução com esse objetivo

- Test-and-Set: na verdade lê e salva o conteúdo de uma variável e modifica o valor da variável (atômico)

  Não é comparação e atribuição mas é suficiente?

Quarta solução

- Ação atômica TS (Test-and-Set)

  Lê e salva o valor de uma variável compartilhada

  "Marca" a variável compartilhada

  Retorna o valor inicial da variável compartilhada

```
bool TS(bool trava) {
      <bool inicial=trava; /* Salva o valor inicial */
      trava=true;          /* ‘‘Liga’’ a trava */
      return inicial;>     /* Devolve o valor original */
}
```

Quarta solução


```
bool trava=false;
process CS[i=1 to n] {
   while (true) {
      while (TS(trava)) skip; /* prot. entrada */
      secao critica;
      trava=false; /* prot. saida */
      secao nao critica;
   }
}
```

- Sem entrada garantida (só com escalonador com justiça forte) :(

Lembrete para todas soluções onde há spin-locks

- Protocolos de saída

  Em uma solução para o problema da seção crítica baseada em spin-lock, o protocolo de saída deve apenas atribuir às variáveis compartilhadas os seus valores iniciais.

Quarta solução

- Ainda não tiramos os $<$ e $>$!
- É necessário usar instruções de baixo nível

  `bt` (bit test)

  `bts` (bit test-and-set)

```
    bts eax, 0
```

- `bts`: copia o bit 0 de eax para o `cf` e atribui 1 ao bit 0 de eax

Quarta solução

- Como implementar o TS com as instruções `bts`, `jc` e `mov`?

- Considere que a variável compartilhada é o bit 0 de `eax`

Quarta solução


```
volta: bts eax, 0 # Prot. entrada
       jc volta
       secao critica
       mov eax, 0 # Prot saida
```

- Solução dependente do hardware
- Código em C com instruções em assembly

Quinta solução

- O algoritmo com Test-and-Set tem problemas quando há múltiplos processos:

  Leitura da variável compartilhada

  Cache em máquinas multiprocessadas com memória compartilhada

- Como melhorar?

Quinta solução

- Test and Test-and-Set

  Ao invés de ficar no loop esperando `TS==true`, vamos tentar aumentar a probabilidade de que o `TS` sempre vai retornar `true`.

- Como resolver usando a ideia do algoritmo para achar o máximo de um vetor?

Quinta solução


```
while (trava) skip; /* Gira enquanto trava eh verdade */
while (TS(trava)) { /* Tenta marcar a trava */
   while (trava) skip; /* Gira de novo se tiver falhado */
     }
```

- Melhorias?

Quinta solução

- Nos dois laços adicionais, `trava` é apenas lida (pode usar o cache)
- Contenção de memória é reduzida

Quinta solução


```
bool trava = false; /* variavel compartilhada */

process CS[i=1 to n] {
      while (true) {
            while (trava) skip; /* prot. entrada */
            while (TS(lock)) {
                  while (trava) skip;
            }
            secao critica;
            trava=false; /* prot. saida */
            secao nao critica;
      }
}
```

Ideias para implementar o await

await × problema da seção crítica: primeira solução

- Qualquer solução para o problema da seção crítica pode ser usado na implementação de uma ação atômica condicional
- <S;> é equivalente a:

```
CSEntrada;
S;
CSSaida;
```

- Obs.: para dar certo todos os processos devem fazer o mesmo

segunda solução

- Como implementar `<await (B) S;>`?
- Lembrando:

  Atrasa a execução até `B` ser verdade

  `B` deve ser verdade quanto `S` começa a executar

segunda solução


```
CSEntrada;
while (!B) { ??? }
S;
CSSaida;
```

- Obs.: para dar certo todos os processos devem fazer o mesmo (Vale para todas soluções seguintes)
- Qual o problema da solução? (Simule dois processos, lembrando que o CSEntrada vai garantir apenas 1 processo por vez na seção crítica)

terceira solução

- Na solução anterior, enquanto `B` não mudar, outros processos não conseguirão executar

```
CSEntrada;
while (!B) {CSSaida; CSEntrada;}
S;
CSSaida;
```

- Qual o problema em termos de eficiência?

quarta solução

- Na solução anterior, contenção de memória. Pode não dar tempo de `B` mudar

```
CSEntrada;
while (!B) {CSSaida; Delay; CSEntrada;}
S;
CSSaida;
```

- O `Delay` pode ser definido como tempos aleatórios (nos exemplos anteriores, o `Delay` pode substituir o `skip`)

quinta solução (se `S` é `skip`)

- Como implementar `<await (B);>` neste caso?

quinta solução (se `S` é `skip`)

- Se `B` satisfaz a PNMUV:

```
while (!B) skip;
```

Utilização na prática

- Similar ao implementado no hardware de placas Ethernet (protocolo de back-off exponencial)
- Uma placa envia um quadro que pode colidir com o quadro de outra placa da rede
- As colisões são detectadas caso ocorram
- Em caso de colisão, os computadores envolvidos na colisão esperam um tempo aleatório