MAC0438 - Programação Concorrente

Daniel Macêdo Batista

IME - USP, 19 de Março de 2013

Roteiro

Soluções justas para o problema da seção crítica

Barreiras de sincronização

Soluções justas para o problema da seção crítica

Ticket – Uma última observação


```
int senhaGeral=1, senhaProxima=1, senha[1:n]=([n] 0);

process CS[i=1 to n] {
  while (true) {
    senha[i]=FA(senhaGeral,1); /* Prot. Entrada */
    while (senha[i]!=senhaProxima) skip;
    secao critica;
    senhaProxima++; /* Prot. Saida */
    secao nao critica;
  }
}
```

- Ocorrerá algum problema se o algoritmo rodar o laço do while muitas vezes? Ou ainda, se o valor de n for muito grande?

Bakery – Objetivo e ideia do algoritmo

- Propor uma alternativa ao algoritmo Ticket

  Não depender da instrução `Fetch-and-Add`

  Não ter que implementar um protocolo de acesso à seção crítica no protocolo de entrada

- Os processos que chegam olham as senhas de todos os processos já presentes e escolhem uma nova maior do que todas atuais

  Ainda tem o problema de que dois processos podem escolher mesma senha, mas não precisa mais ler e incrementar uma variável global (abole o `Fetch-and-Add`)

Protocolos de entrada e saída

- CSEntrada para o processo CS[i]

```
<senha[i]=max(senha[1:n])+1;>
for (j=1 to n st j!=i)
   <await (senha[j]==0 or senha[i]<senha[j];)>
```

- CSSaida para o processo CS[i]

```
senha[i]=0
```

Algoritmo completo


```
int senha[1:n]=([n] 0);

process CS[i=1 to n] {
   while (true) {
      <senha[i]=max(senha[1:n])+1;> /* Prot. entrada */
      for [j=1 to n st j!=i]
         <await (senha[j]==0 or senha[i]<senha[j]);>
      secao critica;
      senha[i]=0; /* Prot saida */
      secao nao critica;
   }
}
```

- Há entrada garantida porque uma hora todos os processos na seção crítica terão que sair dela (`senha[j]=0`) ou porque uma hora a `senha[i]` vai ser a menor
- Problemas?

Algoritmo completo


```
int senha[1:n]=([n] 0);

process CS[i=1 to n] {
   while (true) {
      <senha[i]=max(senha[1:n])+1;> /* Prot. entrada */
      for [j=1 to n st j!=i]
         <await (senha[j]==0 or senha[i]<senha[j]);>
      secao critica;
      senha[i]=0; /* Prot saida */
      secao nao critica;
   }
}
```

- Há 2 operações atômicas definidas com < e >

  `<senha[i]=max(senha[1:n])+1;>`

  `<await (senha[j]==0 or senha[i]<senha[j]);>`

Algoritmo completo


```
int senha[1:n]=([n] 0);

process CS[i=1 to n] {
   while (true) {
      <senha[i]=max(senha[1:n])+1;> /* Prot. entrada */
      for [j=1 to n st j!=i]
         <await (senha[j]==0 or senha[i]<senha[j]);>
      secao critica;
      senha[i]=0; /* Prot saida */
      secao nao critica;
   }
}
```

- Há 2 operações atômicas definidas com < e >

  Não há instrução atômica para calcular o máximo de vários números :(

Algoritmo completo


```
int senha[1:n]=([n] 0);

process CS[i=1 to n] {
   while (true) {
      <senha[i]=max(senha[1:n])+1;> /* Prot. entrada */
      for [j=1 to n st j!=i]
         <await (senha[j]==0 or senha[i]<senha[j]);>
      secao critica;
      senha[i]=0; /* Prot saida */
      secao nao critica;
   }
}
```

- Há 2 operações atômicas definidas com < e >

  A condição do `await` não respeita a propriedade no máximo uma vez :(

Algoritmo completo


```
int senha[1:n]=([n] 0);

process CS[i=1 to n] {
   while (true) {
      <senha[i]=max(senha[1:n])+1;> /* Prot. entrada */
      for [j=1 to n st j!=i]
         <await (senha[j]==0 or senha[i]<senha[j]);>
      secao critica;
      senha[i]=0; /* Prot saida */
      secao nao critica;
   }
}
```

- O limite de tamanho da variável `senha` pode ser um problema?

Soluções para os problemas das ações atômicas

- Seguir a mesma ideia do Tie Breaker e implementar o protocolo de acesso à seção crítica
- Bolar um algoritmo para 2 processos e generalizar para n

Protocolos para 2 processos

- Protocolo de entrada para CS1:

```
senha1=senha2+1;
while (senha2 != 0 and senha1 > senha2) skip;
```

- Protocolo de entrada para CS2:

```
senha2=senha1+1;
while (senha1 != 0 and senha2 > senha1) skip;
```

- Problemas?

Protocolos para 2 processos

- Protocolo de entrada para CS1:

```
senha1=senha2+1;
while (senha2 != 0 and senha1 > senha2) skip;
```

- Protocolo de entrada para CS2:

```
senha2=senha1+1;
while (senha1 != 0 and senha2 > senha1) skip;
```

- Garante exclusão mútua?
- O problema é que o `while` está deixando ambos os processos passarem se as senhas forem iguais. Como corrigir isso?

Protocolos para 2 processos

- Protocolo de entrada para CS1:

```
senha1=senha2+1;
while (senha2 != 0 and senha1 > senha2) skip;
```

- Protocolo de entrada para CS2:

```
senha2=senha1+1;
while (senha1 != 0 and senha2 >= senha1) skip;
```

- Problemas?

Protocolos para 2 processos

- Protocolo de entrada para CS1:

```
senha1=senha2+1;
while (senha2 != 0 and senha1 > senha2) skip;
```

- Protocolo de entrada para CS2:

```
senha2=senha1+1;
while (senha1 != 0 and senha2 >= senha1) skip;
```

- O problema agora é que se os processos começam o protocolo de entrada, as senhas não mudam imediatamente de valor. Elas ainda são 0 por causa do protocolo de saída
- A solução é obrigar a mudança do valor logo no início do protocolo de entrada. Como fazer isso?

Protocolos para 2 processos

- Protocolo de entrada para CS1:

```
senha1=1;
senha1=senha2+1;
while (senha2 != 0 and senha1 > senha2) skip;
```

- Protocolo de entrada para CS2:

```
senha2=1;
senha2=senha1+1;
while (senha1 != 0 and senha2 >= senha1) skip;
```

- Como generalizar agora para n processos? Os protocolos tem que ser simétricos para que possa colocar dentro de um loop mas eles não são simétricos :((> no CS1 e >= no CS2). Como resolver?

Protocolos para 2 processos

- Protocolo de entrada para CS1:

```
senha1=1;
senha1=senha2+1;
while (senha2 != 0 and senha1 > senha2) skip;
```

- Protocolo de entrada para CS2:

```
senha2=1;
senha2=senha1+1;
while (senha1 != 0 and senha2 >= senha1) skip;
```

- É necessário comparar os dois valores das senhas e se forem iguais o desempate deve ser feito pelo id do processo

Protocolos para n processos

- Criar uma nova operação para comparação

```
(a,b) > (c,d) == true , se (a>c) or (if a==c e b>d)
                == false, caso contrario
```

Protocolos para n processos

- Protocolo de entrada para CS1:

```
senha1=1;
senha1=senha2+1;
while (senha2 != 0 and (senha1,1) > (senha2,2)) skip;
```

- Protocolo de entrada para CS2:

```
senha2=1;
senha2=senha1+1;
while (senha1 != 0 and (senha2,2) > (senha1,1)) skip;
```

Algoritmo completo para n processos


```
int senha[1:n] = ([n] 0);

process CS [i=1 to n] {
   while (true) {
      senha[i]=1;senha[i]=max(senha[1:n])+1; /* P. entrada */
      for [j=1 to n st j!=i]
         while (senha[j] !=0 and
               (senha[i],i) > (senha[j],j)) skip;
      secao critica;
      senha[i]=0; /* P. saida */
      secao nao critica;
   }
}
```

Barreiras de sincronização

Objetivo

- Necessidade de garantir que todos os processos cheguem em um determinado ponto antes de prosseguir
- Útil para algoritmos iterativos
- Como implementar com os comandos já conhecidos para definir processos?

Objetivo


```
while (true) {
   co [i=1 to n]
      codigo da tarefa i;
   oc
}
```

- Lembrando da propriedade do co, neste exemplo é possível que o algoritmo passe para uma próxima iteração antes de todos terminarem a iteração atual?
- Problemas?

Objetivo


```
process Worker [i=1 to n] {
   while (true) {
      codigo da tarefa i;
      espere todas as tarefas n terminarem;
   }
}
```

- No exemplo acima, há uma barreira de sincronização no fim dos loops
- Mais eficiente mas são necessários protocolos para garantir a sincronização

Ideia geral da primeira solução – Contador compartilhado

- Ter uma variável `contador` que armazena quantos processos já terminaram
- Como seria o algoritmo?

Algoritmo para a primeira solução


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
   }
}
```

- Problemas?

Algoritmo para a primeira solução


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
   }
}
```

- Ações atômicas com < e >

```
<contador=contador+1;>

<await (contador == n);>
```

Algoritmo para a primeira solução


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
   }
}
```

- Ações atômicas com < e > (Se o hardware tiver Fetch-and-add)

  ```
  FA(contador,1); while (contador != n) skip;
  ```

- É suficiente?

Algoritmo para a primeira solução


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
   }
}
```

- contador tem que ser 0 sempre no início de cada iteração

  ou seja, sempre que todos processos passarem pela barreira

  e antes de algum processo tentar incrementar

Algoritmo para a primeira solução


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
   }
}
```

- contador está sempre mudando e sempre sendo verificado (cache, contenção de memória)

Contador compartilhado


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
   }
}
```

- Como resolver o problema de zerar o contador?

  Ter certeza que os `n` processos só vão começar quando `contador` for zero

  Ideias?

Contador compartilhado


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
      <contador=contador-1;>
      <await (contador == 0);>
   }
}
```

- Mas não resolve o problema de cache e necessidade do FA

Contador compartilhado


```
int contador=0;

process Worker[i=1 to n] {
   while (true) {
      codigo da tarefa i;
      <contador=contador+1;>
      <await (contador == n);>
      <contador=contador-1;>
      <await (contador == 0);>
   }
}
```

- Útil se a máquina tem `FA` ou similar
- Útil se a máquina tem atualização eficiente de cache
- Útil se `n` é pequeno (não por causa do limite do `int` – Verdade?)

Flags e coordenadores

- Para resolver o problema da contenção de memória

  Cada processo atualiza sua variável ao invés de um só (`contador`)

  `count = (arrive[1] + ... + arrive[n])`

Flags e coordenadores

- `<contador=contador+1;>` $\rightarrow$ `arrive[i]=1`
- Ótimo! Não precisa de `FA` :)

Flags e coordenadores

- `<await (contador == n);>` $\rightarrow$ `<await ((arrive[0] + ... + arrive[n]) == n);>`
- Problema?

Flags e coordenadores

- `<await (contador == n);>` $\rightarrow$ `<await ((arrive[0] + ... + arrive[n]) == n);>`
- Problema?

  Contenção de memória :(

  Está fazendo a soma o tempo todo para diversos processos :(

Flags e coordenadores

- `<await (contador == n);>` $\rightarrow$ `<await ((arrive[0] + ... + arrive[n]) == n)`
- Ideias?

  Ter uma variável por processo para marcar que eles podem continuar

  `<await (continue[i] == 1);>`

  Quem vai fazer `continue[i] = 1`?

Flags e coordenadores

- A solução é ter um processo especial (coordenador) que:
  - Espera todos os `arrive` serem 1
  - Faz todos os `continue` serem 1
- Tentem implementar essas duas tarefas do coordenador

Flags e coordenadores

- Algoritmo do coordenador

```
for [i = 1 to n] <await (arrive[i] == 1);>
for [i = 1 to n] continue[i] = 1;
```

- Há o risco de `arrive[i]` mudar depois que o coordenador passou por `i` no laço do `for`?

Flags e coordenadores

- Processo Worker

```
arrive[i] = 1;
<await (continue[i] == 1);>
```

- Processo Coordenador

```
for [i = 1 to n] <await (arrive[i] == 1);>
for [i = 1 to n] continue[i] = 1;
```

- Sem FA e sem contenção de memória
- Pode substituir `await` por `while` (PNMUV)
- Falta zerar as variáveis (Tentem fazer)