MAC0422 - Sistemas Operacionais

Daniel Macêdo Batista

IME - USP, 5 de Outubro de 2020

Roteiro

Barreiras de sincronização

Barreiras de sincronização

Objetivo

- Necessidade de garantir que todos os processos (threads) cheguem em um determinado ponto antes de prosseguir
- Útil para algoritmos iterativos

Objetivo


```
Thread 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 laços
- Mais eficiente do que criar processos (threads) a cada laço mas são necessários protocolos para garantir a sincronização

Ideia da primeira solução – Contador compartilhado

- Ter uma variável `contador` que armazena quantas threads já terminaram
- Como seria o algoritmo?

Algoritmo para a primeira solução


```
int contador=0;

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

- Problemas?

Algoritmo para a primeira solução


```
int contador=0;

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

- Ações atômicas com < e >

  ```
  <contador=contador+1;>
  ```

- Espera ocupada

  ```
  while (contador != n) skip;
  ```

Algoritmo para a primeira solução


```
int contador=0;

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

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

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

  Vamos considerar que o skip deixa a thread dormindo por um tempo
- É suficiente?

Algoritmo para a primeira solução


```
int contador=0;

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

- contador tem que ser 0 sempre no início de cada iteração
  - ou seja, sempre que todas as threads passarem pela barreira
    - e antes de alguma thread tentar incrementar

Algoritmo para a primeira solução


```
int contador=0;

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

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

Contador compartilhado


```
int contador=0;

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

- Como resolver o problema de zerar o contador?

  Ter certeza que as `n` threads só vão começar quando `contador` for zero
- É possível resolver com dois contadores se alternando a cada rodada (a rodada tem que ser variável local de cada thread) – Mas não resolve o problema de cache e a necessidade do `FA`.

Contador compartilhado

- Útil se a máquina tem `FA` ou similar
- Útil se a máquina tem atualização eficiente de cache
- Útil se `n` é pequeno

Flags e coordenadores

- Para resolver o problema da contenção de memória

  Cada thread 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

- `while (contador != n) skip;` $\rightarrow$ `while ((arrive[0] + ... + arrive[n]) != n) skip;`
- Problema?

  Contenção de memória :(

  Está fazendo a soma o tempo todo para diversas threads :(

Flags e coordenadores

- `while (contador != n) skip;` $\rightarrow$ `while ((arrive[0] + ... + arrive[n]) != n) skip;`
- Ideias?

  Ter uma variável por thread para marcar que todas as demais podem continuar

  `while (continue[i] == 0) skip;`

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

Flags e coordenadores

- A solução é ter uma thread especial (coordenador) que:
  - Espera todos os `arrive` serem 1
  - Faz todos os `continue` serem 1