Relatório – Exclusão Mútua

Marcelo Fernandes de Morais Filho
Diogo Nogueira Knop

Engenharia Eletrônica - UTFPR
10 de setembro de 2018

1. Sem Coordenação

Ao executarmos várias threads que incrementam a mesma variável, encontramos uma saída onde o resultado final é muito menor do que o esperado:

```
Sum should be 10000000 and is 2940360

real    0m0,047s
user    0m0,152s
sys     0m0,012s
```

Isso se deve ao fato de que a operação de incremento não é atômica, fazendo com que várias threads façam a leitura do mesmo valor da memória e não a incrementem no tempo correto, fazendo com que outras threads acabem lendo um valor antigo e realizando o mesmo incremento que já foi feito anteriormente.

2. Solução Ingênua

Nesta tentativa de solução, uma flag de *busy* é adicionada para realizar o bloqueio da região crítica enquanto uma das threads a está utilizando. Um problema ocorre, porém, na situação onde após a verificação da flag *busy* como livre e antes de esta ser setada novamente, um escalonamento inoportuno pode causar com que outras threads também passem pela condição e entrem na região crítica ao mesmo tempo.

3. Alternância

Esta tentativa utiliza uma variável *turn* com a intenção de designar turnos para cada processo. A sequencialidade deste método, porém, acaba fazendo mau uso do recurso de escalonamento, pois enquanto uma das threads esteja em seu turno, todas as outras threads escalonadas ficarão travadas. Isso faz com que a maior parte dos recursos do processador seja gasta inutilmente, fazendo com que este método seja muito mais lento do que os outros.

```
Turn: 0, Sum: 0
Turn: 0, Sum: 100
Turn: 0, Sum: 200
...
real    29m11,871s
```

4. Instrução TSL

Esta tentativa utiliza um método parecido com a solução ingênua, porém utilizando uma instrução atômica para realizar a verificação. Esta estrutura *Test-and-set Lock* impede a ocorrência de condições de disputa que ocorriam na solução ingênua, onde uma verificação não atômica permitia acessessos simultâneos à área crítica.

Este método acaba ainda sendo ineficiente, porém, pois grande parte do tempo acaba sendo gasto na verificação, fazendo com que threads fiquem inativas por bastante tempo.

```
Sum should be 10000000 and is 10000000

real    0m41,304s
```

5. Instrução XCHG

Nesta tentativa, a estrutura *Test-and-set Lock* é novamente utilizada, porém com a instrução *XCHG* da plataforma intel. Os resultados encontrados são parecidos, com o valor correto de soma porém também com uma grande ineficiência temporal.

```
Sum should be 10000000 and is 10000000

real    0m50,456s
```

6. Semáforo

Uma solução mais inteligente é a utilização de semáforos, onde ao invés de desperdiçarmos tempo em situações de *idle* quando threads estão bloqueadas, podemos setar um semáforo que bloqueia a thread e impede seu posterior escalonamento até que este semáforo seja desbloqueado. Isso faz com que a execução se torne muito mais eficiente, pois apenas threads ativas são escalonadas e realizam seu trabalho em um período muito menor do que nos métodos anteriores.

```
Sum should be 10000000 and is 10000000

real    0m1,867s
```

7. Mutex

Um caso específico dos semáforos é o mutex, que diz respeito a um semáforo binário, ou seja, que só assume dois possíveis valores. Esta *exclusão mútua* é uma solução minimalista para situações onde um semáforo mais simplificado é o suficiente para a realização do processo de exclusividade da região crítica.

```
Sum should be 10000000 and is 10000000

real    0m1,092
```

Isso se reflete na precisão do resultado (que é o esperado) e no tempo de execução, ainda menor do que o com semáforos (que já era muito mais eficiente do que outros métodos).