UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO I

Blocking Thread Synchronization

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Implementação atual: Busy Waiting

A versão do EPOS dada no enunciado do exercício resolvia o problema de sincronização de Threads utilizando uma estratégia conhecida como *Busy Waiting*. Nesta abordagem, a Thread que deseja utilizar a seção crítica do código continuamente checa uma variável por uma condição. Caso essa condição seja aceita, a Thread continua sua execução, caso contrário, cede a vez para outra Thread executar. Esse comportamento pode ser visto através da implementação do método `Semaphore::p`:

```
void Semaphore::p()
{
    ...
    fdec(_value);
    while(_value < 0)
        sleep();
}
```

O método `sleep`, neste caso, chama `yield`, que cede sua vez de execução a outra Thread, alterando seu estado e se colocando na fila `ready`. Assim que esta Thread for escalonada novamente, a variável `_value` será checada novamente e voltará a dormir caso a condição seja positiva. Entretanto, esta checagem ocorrerá indefinidamente até que outra Thread incremente o valor de `_value` para que possa continuar sua execução.

O problema desta implementação está no uso contínuo de processamento por Threads inativas. Essa estratégia, por exemplo, pode levar a um alto consumo da CPU mesmo por sistemas que deveriam estar totalmente ociosos. Uma vez que a variável de sincronismo é continuamente checada, o processador nunca pára de trabalhar.

Além do mais, a versão implementada no enunciado deste exercício possui um bug e não funciona para o caso em que mais de uma Thread esteja bloqueada por um semáforo. O caso em que isso acontece é o seguinte: A Thread A está em execução, de posse do recurso único de um semáforo, enquanto as Threads B e C foram bloqueadas por este mesmo semáforo. Neste ponto, o valor interno do semáforo é -2 (foi inicializado com 1, e cada Thread decrementou seu valor uma vez). Quando a Thread A liberar o semáforo e incrementar seu valor, este passará a ser -1. Quando a Thread B ou C executar, porém, a checagem `while(_value < 0)` continuará a ser verdadeira e ambas continuarão dormindo, quando na verdade uma das duas deveria ser acordada.

Nova Implementação: uso de filas

A estratégia adotada por esta equipe para resolver o problema de consumo da abordagem *Busy Waiting* foi o uso de filas. Sempre que uma Thread chamar o método sleep, ela será posta em uma nova fila associada ao dispositivo de sincronismo que a bloqueou, colocando uma nova Thread em execução e mudando seu estado para *WAITING*.

A primeira mudança foi no método `p()` de semáforo que passou a checar somente uma vez a variável de controle, mudando o `while` por um `if` e fazendo esta checagem de forma atômica (dentro de um begin / end_atomic):

```
-- semaphore.cc: --
void Semaphore::p()
```

```
{
    ...
    begin_atomic();
    fdec(_value);
    if(_value < 0)
        sleep(); // implicit end_atomic();
    else
        end_atomic();

}
```

As mudanças subsequentes foram feitas em Synchronizer_Common e Thread, para gerenciar a inserção e remoção da Thread na fila de *WAITING*. O problema passa a ser relacionado com a fila em questão e seu gerenciamento. Três abordagens diferentes foram testadas e estão descritas abaixo. A abordagem adotada de fato foi a última delas.

Primeira abordagem: suspender a Thread

A primeira abordagem foi a de reutilizar o estado `SUSPENDED` e os métodos `suspend()` e `resume()` de Thread. Ou seja, sempre que uma thread chamar sleep, ela entrará em modo suspenso, alterando seu estado para *SUSPENDED*, colocando-se na fila de suspensos e dando lugar para outra Thread executar. O `wakeup` neste caso seria somente uma chamada ao `resume` da thread correspondente.

Esta não é uma boa abordagem, uma vez que perdemos a capacidade de separar as Threads que foram suspensas de fato daquelas que estão esperando por um sincronismo. Um caso onde esta abordagem não funciona, por exemplo, é um em que a Thread 1 foi bloqueada pelo semáforo A (e foi suspensa), a Thread 2 foi suspensa via API e a Thread 3, em execução, liberou um recurso do semáforo A. Dependendo do caso, qualquer uma das Threads 1 ou 2 poderiam ser acordadas, já que ambas estão na mesma fila de suspensos, o que não é correto, uma vez que somente a Thread 1 está bloqueada pelo semáforo A.

Segunda ideia: nova fila no escalonador (waiting queue)

A nova abordagem implementada para resolver os problemas descritos na abordagem anterior foi a de criar outra fila única e outro estado para as Threads. Neste caso, o estado `WAITING` e a fila `_waiting`. Ou seja, quando sleep é chamado, a Thread em execução mudaria seu estado e passaria para a fila waiting, dando lugar para outra Thread executar. Agora conseguimos separar as Threads suspensas daquelas que estão dormindo (waiting).

Esta abordagem, porém, é outra má ideia. Imagine a situação em que dois semáforos, A e B são inicializados. A Thread 1 foi bloqueada pelo semáforo A e a Thread 2 foi bloqueada pelo semáforo B. A Thread 3, em execução, liberou um recurso do semáforo A. Neste momento, ambas as Threads 1 e 2 estão na fila de waiting e um wakeup será chamado, acordando uma das Threads. Como as Threads estão na mesma fila, ambas podem ser acordadas, o que não é o comportamento correto, visto que somente a Thread 1 poderia ser acordada.

Terceira estratégia: uma fila para cada Synchronizer

A última estratégia, aquela que foi adotada de fato, foi a criação de uma fila para cada dispositivo de sincronismo:

```
-- synchronizer.h: --
...
class Synchronizer_Common
{
private:
    Thread::Queue _waiting;
    ...
```

Deste modo, cada semáforo, mutex ou condition possui uma fila associada que guardará a referência das Threads que estão esperando por sincronismo. Quando sleep é chamado, a Thread mudará seu estado para *WAITING*, se colocará na fila associada ao dispositivo de sincronismo que a bloqueou e dará lugar para outra Thread executar.
O método wakeup acordará uma das Threads da fila membro do dispositivo de sincronismo que foi liberado e a colocará no estado e na fila *READY*.
Do mesmo modo, wakeup_all() acordará todas as Threads que estão na fila de waiting do Synchronizer correspondente.
Todas as modificação realizadas podem ser encontradas no arquivo *diff* anexado a este relatório.