UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO II

Bloking Thread Joining

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Thread Joining

A operação join, comum em bibliotecas de gerenciamento de threads, bloqueia uma determinada Thread que deseja esperar o término de outra Thread. Por exemplo, dada uma Thread A, que no meio de sua execução deseja que uma outra Thread B finalize sua execução, para então continuar. A Thread A deve dar um Join na Thread B, pois assim a Thread A poderá “esperar” até que a Thread B finalize sua execução para somente depois continuar. Caso a Thread B já tenha terminado sua execução a Thread A continua sua execução normalmente, sem nenhum bloqueio. A operação join comumente também retorna o status de retorno da thread *joined*.

Implementação Atual

A implementação original do EPOS implementava o join através do método de espera ocupada (*spinning* ou *busy waiting*), ou seja, a Thread *joiner* continuamente checava o status da thread *joined*. Caso a Thread *joined* tivesse o status de encerrada, então a Thread *joiner* pode desbloquear e consumir o valor de retorno da outra Thread. Caso contrário, a Thread *joiner* simplesmente invocava o *yield*, que fazia a thread atual voltar para o estado e para a fila ready escalonava outra Thread para tomar a execução. Quando a Thread *joiner* voltasse a ser escalonada, a verificação ocorria novamente e o processo se repetia.

Esta implementação simples, embora funcionasse, tinha o problema de ocupar a CPU com verificações desnecessárias. Uma solução mais elegante seria colocar a thread *joiner* em espera e só acordá-la quando a thread *joined* terminasse sua execução.

Descrição da Solução

Com o objetivo de eliminar a necessidade de realizar constantes verificações, com isso eliminando a espera ocupada da CPU, pelo fim da Thread aguardada foi criado um atributo Queue _joining; em Thread a fim de armazenar as Threads que aguardam seu fim. Essa fila sofre um aumento na chamada do método Thread::join() e uma redução, por completo, com Thread::exit().

Thread::join() Esse método tem como objetivo pausar a execução da Thread em execução até a finalização da Thread que sofreu a chamada de join(). Um exemplo: caso tenhamos duas Threads ***x,y*** e no contexto de execução de ***x*** houver a chamada **y**.join(), ***x*** deve aguardar o fim da execução de ***y***.

Para atingirmos esse objetivo nosso método join() reutiliza, por possuir a mesma semântica de espera do exercício 1, o método wait() presente na Thread que recebe uma fila como parâmetro, no caso deste exercício a fila _joining, e tem o papel de alterar o estado da Thread em execução para WAITING, adiciona-la na fila e escalonar a próxima Thread que esteja READY. Com isso, ao final da execução temos armazenado na fila _joining da Thread joiner (thread que possuí threads que aguardam sua finalização) a Thread joined (thread que aguarda a finalização de outra thread).

```
class Thread

{

...

protected:

      Queue _joining; // 1

      Thread * _joined; // 2

      static void wait(Queue * waiting_queue); // 3

      void wakeup(); // 4

      void wakeup_joiners(); // 5

...

}
```

(1) Fila de joiners. Onde são salvas as referências para as threads que estão esperando pelo término da thread atual. Optamos por utilizar a mesma estrutura de fila do escalonador, com prioridade, para que as threads sejam acordadas de acordo com sua prioridade ao chamar o *exit* da thread joined. Isss só faz diferença, porém, se o escalonador for acionado via uma interrupção durante a operação wakeup_joiners, antes que todas sejam acordadas e inseridas na fila *ready*.

(2) Caso esta Thread esteja esperando por outra, guarda o ponteiro da thread *joined.* Este ponteiro é mantido para manipulações de alguns casos especiais, como por exemplo, a deleção da thread *joiner*, descrito nas próximas seções deste relatório.

(3) Faz a Thread que está rodando entrar para o estado WAITING, se colocar na fila parâmetro e outra Thread da fila ready ser escalonada para executar. Este método foi implementado na solução do exercício passado e reutilizado neste.

(4) Faz a Thread mudar seu estado para READY e se colocar na fila ready. Este método foi implementado na solução do exercício passado e reutilizado neste.

(5) Acorda todas as Threads da fila _joining, invocando o método wakeup de cada uma.

O método Thread::join() é responsável por colocar a thread *joiner* na fila da thread *joined*, mas apenas caso a thread joined não esteja encerrando (FINISHING). Neste caso, a thread joiner pode continuar sua execução e consumir o valor de retorno da thread pela qual estava esperando, pois esta já terminou sua execução.

Utilizou-se um assert para especificar a pré-condição de que a thread *joiner* e *joined* não podem ser iguais (join em si mesmo). Este caso é detalhado na seção final deste relatório.

Outro caso especial verificado neste método por meio de um assert é o joining cíclico (não transitivo). Este problema também está detalhado na seção final.

if(_state != FINISHING) -> passou a checar somente uma vez o estado, mudando o `while` por um `if` , chamando o método wait() e passando o endereço da fila joining.

```
int Thread::join()

{

    lock();


    db<Thread>(TRC) << "Thread::join(this=" << this << ",state=" <<
_state << ")" << endl;


    // A Thread should not join itself

    assert(this != _running);


    // Avoid cyclic joining (transitive cyclic joining untreated)

    assert(this->_joined != _running);


    if(_state != FINISHING) {

        _running->_joined = this;

        Thread::wait(&_joining); // implicit unlock()

    } else

        unlock();


    return *reinterpret_cast<int *>(_stack);

}
```

O valor de retorno da thread é salvo em sua pilha ao chamar o método exit(). Nesta ocasião também são acordadas as threads que estavam esperando pela thread que está

encerrando. Estas threads acordadas voltam a executar e consome o status de retorno da thread *joined.*

Thread::wakeup(): Tem como objetivo “acordar”, ou seja, se preparar para quando solicitado ser posta em execução. Para isso, ao final da execução do wakeup a Thread deve estar no estado READY e estar presente na fila das outras Threads com o estado READY. Este método foi reaproveitado da solução do exercício anterior e não foi modificado.

```
void Thread::wakeup()

{

    lock();


    db<Thread>(TRC) << "Thread::wakeup(this=" << this << ")" << endl;


   _state = READY;

   _ready.insert(&_link);


   unlock();

}
```

Thread::wakeup_joiners(): Acorda todas as Threads joiners da thread atual. O objetivo é alcançado através da chamada de wakeup() para cada Thread presente na lista _joining. Além disso, as referências à thread *joined* de cada thread acordada são limpas (`joiner->_joined`).

```
void Thread::wakeup_joiners()

{

    lock();


    db<Thread>(TRC) << "Thread::wakeup_joiners(this=" << this << ")" <<
endl;
```

```
    Thread * joiner;

    while(!_joining.empty()) {
        joiner = _joining.remove()->object();
        joiner->_joined = 0;
        joiner->wakeup(); // implicit unlock()
    }

    unlock();
}
```

Thread::exit(int status) A primeira tarefa do método exit é acordar todas as threads *joiners* da thread que está encerrando. Faz-se isso através do método wakeup_joiners da Thread que está rodando. Porém, temos que salvar o status no stack da Thread antes de fazer isso, para evitar que a Thread que está encerrando seja preemptada e uma Thread *joiner* seja executada, consumindo o valor de retorno da Thread *joined* antes de ele ser alocado. Essa preempção pode ocorrer imediatamente após o unlock dentro do método wakeup_joiners e antes do lock (2).

```
void Thread::exit(int status)
{
    lock();

    db<Thread>(TRC) << "Thread::exit(status=" << status << ")
[running=" << running() << "]" << endl;

    // 1
    *reinterpret_cast<int *>(_running->_stack) = status;
    _running->wakeup_joiners(); // implicit unlock();

    lock(); // 2
```

```
    while(_ready.empty() && !_suspended.empty())
        idle(); // implicit unlock();

...
}
```

```
void Thread::wait(Queue * waiting_queue)
{
    lock();

    db<Thread>(TRC) << "Thread::wait(running=" << _running << ")" <<
endl;

    if(!_ready.empty()) {
        Thread * prev = _running;
        prev->_state = WAITING;
        waiting_queue->insert(&prev->_link);

        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        Thread::dispatch(prev, _running);
    } else
        Thread::idle();

    unlock();
}
```

O método **Thread::wait()** é exatamente o mesmo utilizado no exercício anterior. Sua função é alterar o status de uma thread para WAITING e colocá-la numa fila qualquer, passada como parâmetro. O fato de enviar a fila de espera como parâmetro, apesar de parecer um pouco estranho em um primeiro momento, facilita o reuso do método e este exercício provou exatamente este ponto: enquanto no exercício anterior o método era usado por mecanismos de sincronismo (Synchronizer) para sincronizar threads, neste ele é usado para *thread joining*. Sua semântica, também, é muito legível: “entre em modo de espera e se inclua nesta fila de espera”.

Casos Especiais

Joining em si mesmo

Uma Thread pode dar join em si mesmo. Imagine a situação em a Thread A esteja rodando e, no meio de sua exeução, dê thread em si mesmo. Neste caso, não faz sentido fazer a Thread “esperar por si mesma”. Caso esta situação não seja tratada, porém, a Thread entrará no estado WAITING e se colocará em sua própria fila joining. Neste caso, esta Thread nunca voltará a executar, pois não cairá no método exit() e, portanto, seus joiners (que incluem a si própria) nunca serão acordados.

Uma medida que poderia ser tomada é não fazer nada quando uma thread dá join em si mesma, ou seja, simplesmente deixar sua execução continuar. Porém, neste caso, o que o método join deve retornar? Por padrão o join retorna o valor de retorno da Thread, mas a Thread que deu join em si mesma ainda não terminou e não possui valor de retorno. Podemos retornar lixo ou estipular um valor de retorno neste caso, porém, serão situações que fogem à regra.

Levando isto em conta, preferimos tratar este caso com um simples *assert* dentro do método *join*. Ele checará que a thread *joiner* não é igual a thread *joined*:

```
int Thread::join()
{
   ...
    // A Thread should not join itself
    assert(this != _running);
```

O assert não trata o caso, mas, dependendo das configurações de compilação, avisam o usuário em tempo de execução de que uma pré-condição não foi atendida. O assert também pode ser usado por ferramentas de análise estática para verificar se este erro pode ocorrer.

Joining Cíclico

Joining cíclico pode ocorrer se uma Thread A dar join na Thread B e a Thread B dar join na Thread A, diretamente ou transitivamente. Neste caso, ambas as threads entrarão no modo

de espera e nas filas da thread pela qual estão esperando. Como ambas não serão mais executadas, nenhuma delas cairá no exit() e não ocorrerá a chamada de wakeup_joiners. Esta configuração caracteriza uma situação de deadlock. Neste caso, outra Thread não envolvida no deadlock e com estado pronto será posta em execução ou o sistema entrará no modo idle caso contrário.
Verificamos apenas o caso de joining cíclico direto e não transitivo, que é quando a Thread A deu join na Thread B e a Thread B deu join na Thread A. Esta verificação foi introduzida no método join na forma de um assert:

```
int Thread::join()

{

    ...

    // Avoid cyclic joining (transitive cyclic joining untreated)

    assert(this->_joined != _running);
```

O assert não trata o erro, mas informa ao usuário de que o problema ocorreu e também pode ser utilizado por ferramentas de análise estática.
Joininig cíclico transitivo (por exemplo, Thread A deu join na B, que deu join na C, que deu join na A) não foi tratado pois trata-se de uma verificação particularmente custosa. Optamos por não introduzir esta complexidade na biblioteca de Threads do EPOS e deixar este controle à cargo do usuário.

Deleção da Thread Joiner
Imagine que a Thread A dê join na Thread B que começa a executar. Por algum motivo, a Thread A foi deletada. Neste momento, o ponteiro para Thread A continua existindo na fila *joining* da Thread B. Este valor pode ocasionar um erro de acesso quando consumido.
Para tratar essa situação, criamos o atributo *joined* para cada Thread. Este guarda um ponteiro para a Thread *joined* pela qual a thread atual está esperando. Desta forma, no destrutor da Thread A do exemplo anterior, podemos remover a thread joiner (A) da thread joined (B):

```
Thread::~Thread()
{
    lock();

    ...

    // No Threads should be waiting for a deleted one
    assert(_joining.empty());

    // If deleted thread is joining another one, remove it from the
list of that thread
    if (_joined)
```

```
        _joined->_joining.remove(&_link);

    _ready.remove(this);
    _suspended.remove(this);

    unlock();

    kfree(_stack);
}
```

Deleção da Thread Joined

Imagine que a Thread A dê join na Thread B. Suponha que a Thread B seja deletada por algum motivo, antes de conseguir exitar. Nesta situação, a Thread A, que estava na fila *joining* da Thread B, não voltará a ser escalonada, pois o método wakeup_joiners da Thread B não será executado (isso acontece dentro do exit()). Não podemos, também, simplesmente acordar todos os *joiners* de uma thread no momento de sua deleção, pois não há valor de status de retorno para ser passado. Neste cenário, a Thread foi destruída antes de ser exitada, portanto sua execução não terminou completamente e seu valor de retorno não está disponível. Chegamos ao mesmo paradoxo do caso “Joining em si mesmo”.

Nesta situação, optamos por utilizar um assert para estipular a pré-condição de que, no momento de deleção de uma Thread, nenhuma outra Thread esteja em sua fila *joining*. O tratamento deste erro fica a carga do programador.