Relatório I de Sistemas Operacionais II - INE5424
Implementação Alternativa de Bloqueio para *Thread Joining*

Bruno Izaias Bonotto - 16104059
Fabíola Maria Kretzer - 16100725
João Vicente Souto - 16105151

1. INTRODUÇÃO

A utilização de paralelismo em sistemas operacionais demanda uma série de recursos que o torne viável. Determinadas funções de sincronização de processos e threads permitem a execução correta de determinadas atividades paralelamente. Uma delas, comumentemente chama de *join*, é executada por um processo ou *thread* quando o mesmo necessita aguardar a finalização das atividades delegadas à um outro processo ou *thread*.

Neste sentido, a versão didática do Sistema Operacional Paralelo Embarcado (*Embedded Parallel Operating System* - EPOS) conta com uma implementação simplificada dessa função *join()* para os objetos do tipo *Thread*. Esta implementação, portanto, é responsável por verificar se a *thread* solicitada já finalizou suas tarefas (que é possível apenas verificando o estado atual em que o objeto se encontra). Dessa forma, o objeto solicitante verifica o estado atual da *thread* e caso ela não esteja em *FINISHING* este perde o processador (é colocado novamente na fila de *threads* escalonáveis) dando lugar à outra *thread*. No entanto, a *thread* que perdeu o processador pode ser novamente selecionada para executar, mesmo que sua atividade seja simplesmente verificar se o estado da *thread* solicitada tenha sido alterado, o que pode ocorrer diversas vezes durante o tempo de vida das *threads*.

Procurando estimular os conhecimentos adquiridos ao longo das disciplinas, este trabalho demanda um estudo sobre o processo de bloqueio de threads com objetivo final de implementar uma versão alternativa do método *join()*, de forma que esse não permita uma *thread* ser novamente escalonada a menos que de fato possa usar o processador de forma eficiente, ou seja, uma implementação que bloqueie os solicitantes de *join* enquanto os solicitados não finalizarem suas atividades.

2. SOLUÇÃO UTILIZADA

A solução adotada para este trabalho consiste na inclusão de uma fila de *threads* no objeto *Thread* (*_waiting_exit*). Esta fila é utilizada para armazenar as n *threads* que

solicitaram o *join()* à uma determinada *thread*. Sendo assim, para garantir que as n *threads* não vão ser novamente escalonadas até que possam de fato fazer uso relevante do processador, o método *join()* bloqueia a solicitante, inserindo-a na fila de espera por finalização e a suspende. Por fim, quando uma *thread* finaliza suas tarefas e executa o método *exit()* (onde altera seu estado para *FINISHING*), ela então libera todas as n *threads* que estão aguardando sua finalização.

3. IMPLEMENTAÇÃO

a. thread.h

Para prover o bloqueio de n *threads* que desejam esperar o término de uma *thread* qualquer, foi atribuído uma fila privada a cada *thread* para armazenar o *link* das *threads* solicitantes (*_waiting_exit*). Buscando não corromper as demais filas das *threads*, se fez uso do mesmo *link* utilizado na implementação do exercício anterior (*_sync_link*), resultando em uma fila ordenada por chegada, no entanto, a ordem é irrelevante nesta implementação.

Como a fila *_waiting_exit* é privada, fez-se necessário a criação de uma função (*freedom()*) que permitisse liberar todas as *threads* que estão bloqueadas, deve-se também ao fato de que *exit()* é uma função definida estaticamente, impossibilitando o uso direto de atributos não-estáticos do objeto.

i. Código:

```
...
class Thread
{
    …
    typedef Thread::Synchronized_Queue Queue

    ...
    void freedom();

    ...
    Synchronized_Queue _waiting_exit;
};
...
```

b. thread.cc

As funções *exit()* e *join()* no sistema operacional EPOS buscam ter as mesmas características da biblioteca *POSIX*, implementada no *Linux*. Entretanto, a implementação tinha uma abordagem ingênua, de modo que uma *thread,* ao realizar a função *join()*, era frequentemente escalonada apenas para verificar o estado da *thread* de interesse, caso fosse diferente de *FINISHING*, era retirada do processador e colocada novamente na fila *_ready*, apta a ser escalonada novamente. Tal abordagem pode vir a degradar o desempenho, onde o desperdício de processamento na troca de contexto ocupa ciclos de relógio que poderiam ser utilizados por outras *threads* do sistema.

Desta forma, para melhorar o desempenho da função *join(),* foi proposta as modificações citadas anteriormente, onde caso o estado da *thread* de interesse seja diferente de *FINISHING*, entregamos a responsabilidade de avisar a *thread* solicitante para a *thread* solicitada, inserindo a primeira na lista *_waiting_exit* e bloqueando-a através da função *suspend()*.

Já a solução encontrada para a função *exit()* foi, primeiramente, liberar todas as *threads* bloqueadas através da função *freedom()*, possibilitando o escalonamento de quem estivesse esperando. Em seguida, são realizadas algumas verificações para que o sistema operacional mantenha-se executando corretamente. A primeira consiste em executar a função *idle()* caso existam *threads* no sistema mas nenhuma que possa executar no momento. A existência de ao menos uma *thread* na fila *_ready* permite a troca de contexto. E caso ambas as filas sejam vazias, dependendo da configuração de *reboot* do sistema, ou ele para a CPU (*halt()*) ou finaliza o próprio sistema (*reboot()*).

Por fim, para cuidar do caso em que houve a deleção de uma *thread*, a função *freedom()* verifica se o *link* a ser removido da fila é válido antes de liberar a *thread* de fato. De forma semelhante, o destrutor da *thread* passa agora a liberar todas as *threads* que esperam por essa para executar.

i. Código:

```
...
int Thread::join()
{
    ...
    if (_state != FINISHING)
    {
        _waiting_exit.insert(&(_running->_sync_link));
        _running->suspend();
    }
    ...
```

```
}
…
int Thread::exit(int status)
{
    lock();
    …
    _running->freedom();

    while (_ready.empty() && !_suspend.empty())
        idle();

    lock();

    if (!_ready.empty())
    {...}
    else if (reboot)
    {   …
        Machine::reboot();
    }
    else
    {   …
        CPU::halt();
    }

    unlock();
}

void Thread::freedom()
{
    while (!_running->waiting_exit.empty())
    {
        auto link = _running->_waiting_exit.remove();

        if (link != 0)
            link->object()->resume();
    }
}
...

Thread::~Thread()
{
    ...
    if (_state != FINISHING)
        freedom();
    ...
}
```

...

4. CONCLUSÃO

A implementação à priori dos mecanismos de *join()* e *exit()* do sistema operacional EPOS em sua versão educacional não estava completamente de acordo com o padrão *Posix*. No caso do *join()*, o simples fato de remover o processador da *thread* que aguarda a finalização de outra permite que a CPU possa ser utilizada apenas para checagem de uma determinada variável, sendo que existem formas de implementação que permitem o uso da CPU de forma mais eficiente ao escalonar *threads* que realmente possam fazer bom uso do hardware. Uma dessas alternativas foi utilizada como solução para o problema este trabalho. Ao bloquear as threads que solicitaram join() e desbloqueando-as apenas na finalização da thread solicitada, o processador não é alocado para threads que, no final das contas, não pode executar. Dessa forma otimiza-se o uso dos recursos da CPU e reduz-se o tempo de execução.

No entanto, para avaliar de fatos os benefícios dessa nova abordagem, seriam necessários diversos testes que pudessem extrair informações relevantes. Análises estatísticas de tempo de uso da CPU poderiam ser bons indicadores sobre os reais ganhos com essa abordagem.