Relatório V de Sistemas Operacionais II - INE5424
Destruição de Objetos do Sistema

Bruno Izaias Bonotto - 16104059

Fabíola Maria Kretzer - 16100725

João Vicente Souto - 16105151

1. Introdução

Se não todas, grande parte das aplicações computacionais precisam lidar com o gerenciamento de memória, seja diretamente através de manipulação de endereços (geralmente controlada pelo Sistema Operacional (SO)) ou indiretamente (o gerenciamento de memória é abstraído por algumas linguagens de programação e até mesmo pelos SOs).

Para manter a consistência dos dados, de maneira geral o SO deve manter a semântica proposta pela linguagem de programação na qual o usuário está implementando sua aplicação. Caso contrário, podem ocorrer efeitos colaterais indesejáveis, como por exemplo: em uma aplicação qualquer é criado um objeto dinamicamente, ou seja, é alocado um endereço de memória válido para o objeto. O SO precisa garantir que este dado permaneça válido enquanto o programa o disser, pois caso o SO destrua o objeto e posteriormente a aplicação tente ler ou escrever naquele endereço, haverá uma inconsistência, o que inviabiliza o uso do SO.

Atualmente, a versão didática do EPOS possui uma implementação que não trata cuidadosamente de algumas operações de deleção e, portanto, não garante completamente as propriedades discutidas anteriormente.

Com a finalidade de aplicar e revisar os conhecimentos adquiridos ao longo da disciplina, este trabalho tem como objetivo revisar e melhorar a implementação dos destrutores de algumas classes do EPOS. Ainda, garantir que certas inconsistências geradas por aplicações não comprometa o funcionamento do SO.

2. Soluções Utilizadas

a. ~Thread

Foram identificadas diversas inconsistências ao analisar-se o destrutor da classe *Thread*. Primeiramente, ao deletar uma *thread*, seu *link* era buscado em todas as filas comuns as *threads*, garantindo que ela fosse removida das estruturas. Tais operações eram desnecessárias uma vez que em uma *thread* poderia estar em apenas uma fila simultaneamente. Como solução, para cada estado em que uma *thread* pode ser deletada existe um tratamento diferenciado, o que é possível de ser realizado com uma simples estrutura de seleção (*switch*).

Identificou-se na implementação anterior, que ao deletar-se uma *thread* suspensa e que estava aguardando pela finalização de outra (*join*), esta última por sua vez, teria um ponteiro (*_joining*) inconsistente, apontando para uma região de memória inválida quando fosse liberar a *thread* que a estava aguardando. Assim, os métodos *exit()* e destrutor de objetos do tipo *thread* tentariam acordar uma *thread* que já não existia mais no sistema. Para solucionar este problema, foi criado um atributo na classe *Thread* (*_waiting_join*), servindo de ponteiro para a *thread* de interesse do *join*. Sendo assim, ao deletar-se uma *thread* que esteja esperando outra terminar, é possível remover a inconsistência mencionada ao anular ambos os ponteiros *_joining* e *_waiting_join*.

Por último, foi criado um *assert* para garantir que uma *thread* não possa se auto deletar. Essa foi uma solução minimalista em termos de modificação no SO, uma vez que para permitir auto-deleção seria necessário implementar novos métodos para realizar a troca de contexto, já que um deles conteria memória a ser desalocada. Outro ponto importante levado em consideração nesta solução foi a necessidade de desalocar a pilha (*_stack*) ao final do destrutor, o que geraria outro conflito na troca de contexto.

```
Thread::~Thread()
{
    lock();
    ...
    assert(_state != RUNNING);
```

```
    switch (_state)
    {
    case READY:
        _ready.remove(this);
        _thread_count--;
        break;

    case SUSPENDED:
        if (_waiting_join) //! Libera thread da responsabilidade de me acordar
            _waiting_join->_joining = 0;

        _suspended.remove(this);
        _thread_count--;
        break;

    case WAITING:
        _waiting->remove(this);
        _thread_count--;
        break;

    default: //! FINISHING
        break;
    }

    if(_joining) {
        _joining->_waiting_join = 0;
        _joining->resume();
    }
    unlock();
    kfree(_stack);
}
```

b. ~Alarm

O destrutor da classe *Alarm* já estava consistente com o sistema e a linguagem. Ao deletar-se um alarme, é realizada a verificação na fila de alarmes pendentes (*_request*) a fim de removê-lo (utilizando a função *remove()*) e não permitir a execução de um alarme inexistente. Além disso, conclui-se que a implementação está correta devido ao fato de que a fila *_request* não necessita

conter alguém, nem mesmo o *link* para o alarme que deseja-se deletar, pois a função *remove()* garante a consistência na fila.

Por fim, notou-se que um alarme ser deletado durante a execução de um *handler* associado à ele (já que todo *handler* pode ser preemptado) não ocasiona problema ao SO, já que o *handler* não possui mais vínculo com o alarme quando é executado. Assim, é responsabilidade do usuário deletar tanto o *handler* quanto o alarme.

```
Alarm::~Alarm()
{
  lock();
  db<Alarm>(TRC) << "~Alarm(this=" << this << ")" << endl;

  _request.remove(this);
  unlock();
}
```

c. ~Handler

A implementação dos destrutores das especializações da classe *Handler* estão corretos ao não deletar os objetos/funções recebidos em seus construtores. Eles possuem apenas as responsabilidades de encapsular e executar certas operações sobre estes objetos/funções. Portanto, deletar um *handler* associado a um alarme e não deletar o objeto passado pelo construtor deve ser responsabilidade do usuário, caso contrário, o SO engessaria o uso de *handlers* no sistema e degenerando o desempenho devido à necessidade de verificações e medidas preventivas em casos de erros introduzidos pelo usuário.

d. ~Synchronizer

Também não foi identificado nenhum problema com o destrutor do *Synchronizer*. Ele lida com as duas situações de deleção possíveis corretamente. A primeira, onde exista uma *thread*, ou mais, parada na fila de sincronização e, neste caso todas são liberadas, ou seja, inseridas na fila *_ready*. A segunda, quando não

há nenhuma *thread* a ser acordada, as funções que utilizam a fila de sincronização tratam do caso em que ela está vazia, garantindo a consistência do SO.

```
~Synchronizer_Common() { begin_atomic(); wakeup_all(); }

void wakeup_all() { Thread::wakeup_all(&_queue); }

void Thread::wakeup_all(Queue * q)
{
    …
    while(!q->empty()) {
        …
    }
    …
}
```

3. Conclusão

Inicialmente, a versão didática do EPOS não tinha todos os destrutores corretamente implementados e de forma correspondente à linguagem C++. Dessa maneira, era possível algumas situações específicas de utilização dos recursos do SO onde o programador de software aplicativo poderia ter vários problemas. Este sentido, com o objetivo de garantir a consistência do EPOS, diversas verificações e correções em relação à deleção de objetos foram necessárias.

Tomados os devidos cuidados com cada fila de *threads*, alarmes pendentes, *handlers* e sincronizadores, é possível que a probabilidade de erros relacionados ao mau gerenciamento de memória sejam mínimos. No entanto, para garantir que erros não sejam possíveis faz-se necessário uma verificação formal do SO, o que não é simples e, portanto, completamente fora do escopo da disciplina. Além disso, para avaliações de desempenho, são necessários testes mais específicos do que simples aplicações, tais como as utilizadas na versão didática do EPOS. No entanto, como a disciplina se propõe a desenvolver os conceitos de SO e como ele lida com características físicas do *hardware*, avaliações de desempenho e provas de corretude não se fazem necessárias.