UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO V

System Object Destruction

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Synchronizer

Deleção do Sincronizador

Na deleção de uma estrutura sincronizadora liberamos, com **wakeup_all**, todas as Threads que estão aguardando um recurso compartilhado. A deleção de um mecanismo de sincronização é de responsabilidade do programador. Portanto, ao deletá-lo, o programador indica que não deseja mais realizar um sincronismo sobre o código e as Threads podem continuar suas execuções.

```
 ~Synchronizer_Common() {
      db<Synchronizer>(TRC) << "~Synchronizer_Common(this=" << this <<
")" << endl;
      wakeup_all();
}
```

Thread

Auto deleção

Estipulando uma pré-condição de que, no momento de deleção de uma Thread, ela não se auto-delete utilizamos um assert para checar tal condição.

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();

      db<Thread>(TRC) << "~Thread(this=" << this
```

```
                    << ",state=" << _state
                    << ",priority=" << _link.rank()
                    << ",thread_count=" << _thread_count
                    << ",stack={b=" << reinterpret_cast<void *>(_stack)
                    << ",context={b=" << _context
                    << "," << *_context << "})" << endl;

        // Auto deletion is not allowed
        assert(this != _running);

        // 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);

        if (_state != FINISHING)
            _thread_count--;

        _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.

Alarm

Quando o destrutor de Alarm é invocado, remove-se este alarme da fila *_request*.

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

	db<Alarm>(TRC) << "~Alarm(this=" << this << ")" << endl;

	_request.remove(this);

	unlock();
}
```

Deleção do Handler

Considere o seguinte caso:

```
Function_Handler *handler_a = new Function_Handler(&func_a);
Alarm alarm_a(2000000, handler_a, iterations);
delete handler_a;
```

Ao disparar, o alarme em questão tentará executar a função handler, que foi deletada. Isso ocasionará um erro por tentar executar uma área da heap que foi desalocada. Este caso, porém, só acontece quando o programador explicitamente iniciar um Handler e destruí-lo após associá-lo a um alarme que ainda não disparou. Julgamos que trata-se de um caso específico causado unicamente pelo programador, portanto não tratamos este caso.