Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
INE5424 - Sistemas Operacionais II
Professor: Antônio Augusto Medeiros Fröhlich
Estagiário de Docência: Mateus Krepsky Ludwich
Grupo: 01
Alunos: Alisson Granemann Abreu matrícula: 11100854
Maria Eloísa Costa matrícula: 10200630
Osvaldo Edmundo Schwerz da Rocha matrícula: 10103132

E5: System Object Destruction

EPOS abstracts system entities as ordinary C++ objects, thus promoting usability. At some points, however, the semantics of operating system and programming language do not seem to fit together easily. Consider for instance the case in which a system abstraction (say thread), created by the application programmer with new, is destroyed by the operating system (e.g. exit). Or the counterpart situation: the application programmer deleting a thread that is currently being manipulated by the operating system.

A careful binding between programming language and operating system must be defined to prevent undesirable side effects.

To do

Modify the implementation of this didactic version of OpenEPOS such that the delete C++ operation preserves system consistency at the same time it honors the programming language semantics.

Análise

Sempre que trabalhamos com uma thread, algumas questões precisam ser analisadas cuidadosamente para que não deixemos o sistema de forma inconsistente, seja movendo-as de um lado para outro, como também destruíndo-as. É interessante sabermos:

- Quais threads realmente estão rodando no sistema;

- Determinar se as threads estão efetuando exatamente o esperado;

- Quando é seguro usar os resultados obtidos por uma thread;

e talvez os pontos mais importantes para este exercício, visto que os outros pontos foram tratados ao longo dos exercícios anteriores:

- Garantir que todos os recursos associados a uma thread foram liberados adequadamente;

- Garantir que uma thread não vai tentar acessar dados de um objeto que já tenha sido destruído.

Thread

Quando uma thread é destruída é importante verificar todas as filas as quais ela possa estar vinculada. Para tanto, fizemos a seguinte alteração no destrutor da thread, garantindo que uma thread que esteja em execução, não possa se destruir:

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

    db<Thread>(TRC) << "~Thread(this=" << this
                    << ",state=" << _state
                    << ",priority=" << _link.rank()
                    << ",stack={b=" << _stack
                    << ",context={b=" << _context
                    << "," << *_context << "})" << endl;

      assert(_state != RUNNING); // Eu não posso me deletar se estou
fazendo algo!

      switch(_state) {
```

```
                case READY:
                        _ready.remove(this);
                        _thread_count--;
                        break;
                case SUSPENDED:
                        _suspended.remove(this);
                        _thread_count--;
                        break;
                case WAITING:
                        _waiting->remove(this);
                        _thread_count--;
                        break;
                case FINISHING: // Já chamei exit, só saindo do sistema;
                        break;

        }

        if(_joining)
            _joining->resume();

        unlock();

        kfree(_stack);
    }
```

Colocamos a restrição de assert para o estado de Running, visto que a thread pode não ter terminado as alterações que estaria fazendo, podendo deixar informações do sistema inconsistentes para as próximas threads que precisem desses dados.

Running também está como opção do switch/case por ser uma opção de _state, caso contrário o próprio compilador não aceita compilar o código alegando a falta de um dos estados possíveis de thread.

Seguimos o pressuposto que, ao criar uma thread com a chamada new, o operador delete deve ser chamado a fim de terminá-la pelo próprio usuário. Desta forma, o usuário fica responsável pela deleção de threads criadas por ele.

Synchronizers

Também foi necessário analisar a destruição synchronizers, pois é preciso garatir a integridade da fila de threads. A solução usada já existia no synchronizer.h com a chamada de um wakeup_all() no destrutor de um synchronizer (mutex, semáforo ou condition), para caso ainda exista uma thread na fila e seu estado não fique pendente. É importante ressaltar que a destruição de uma thread em execução foi tratada na própria thread. Um ponto importante também verificado foi se, em algum momento é verificada a situação da fila do synchronizer, pois ela não pode estar vazia quando for chamado wakeup/wakeup_all. Isso já era feito no método dentro de thread.cc.

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

void Thread::wakeup_all(Queue * q)
{
    db<Thread>(TRC) << "Thread::wakeup_all(running=" << running() <<
",q=" << q << ")" << endl;

    // lock() must be called before entering this method
    assert(locked());

    while(!q->empty()) {
        Thread * t = q->remove()->object();
        t->_state = READY;
        t->_waiting = 0;
        _ready.insert(&t->_link);
    }

    unlock();

    if(preemptive)
        reschedule();
}
```

Alarm

O problema na deleção de um alarme está na verificação da fila _request, onde um alarm pode possuir referência. O destrutor possui já a chamada do método remove() para remover o alarm da fila, porém não havia a verificação se esta fila, _request, estava ou não vazia no momento que o destrutor era chamado. Foi adicionado um assert a fim de garantir que a fila _request não estará vazia quando for chamado o remove().

```
Alarm::~Alarm()
{
       assert(!_request.empty());

    lock();

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

    _request.remove(this);

    unlock();
}
```

Handler

Outra preocupação que encontramos foi o que poderia acontecer caso o objeto fosse deletado antes mesmo do próprio handler? Neste caso, consideramos que o sistema operacional não criaria tal situação e que seria um erro do próprio usuário efetuar tal operação.