UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO VII

Scheduler Isolation

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

1- What are the policies implemented by the original multilevel scheduler? What is the inter-level policy? And what is the intra-level?
Intra-level: **round-robin**. Threads de uma mesma prioridade são escalonados para executar na ordem de entrada por um *quantun* de tempo pré-determinado. Quando este tempo acabar, uma nova Thread é escalonada.

Inter-level: **priority**. Threads com nível de prioridade maior são escalonadas por primeiro. Enquanto ativa, threads com prioridade mais baixa não tomarão a CPU.

2- How is it that EPOS implements a multilevel scheduler with a single Ready Queue?
A fila já implementa controle de prioridade através de *rank* de elemento de lista. Desta forma, ao inserir uma Thread na lista, ela é posta na posição exata correspondente à sua prioridade. Ou seja, Threads com maior prioridade são postas no começo da fila e, portanto, escalonadas por primeiro quando o escalonador retirar um membro da fila para executar.

3- Could you implement other policies using the same strategy?
Sim, bastaria alterar a política de inserção / remoção de elementos da fila ready. Mas, para isso, seria necessário alterar o tipo da fila.

Refatoração da classe Thread usando Escalonador

Na implementação antiga do EPOS, o escalonamento estava espalhado pela classe Thread. Quando um novo elemento deveria ser suspenso, resumido, acordado ou escalonado, por exemplo, cabia à Thread manipular as filas e gerenciar o escalonamento.
Este aspecto foi isolado da classe Thread através da criação de uma nova classe Scheduler, que encapsula o gerenciamento de filas e política de escalonamento.

Definição e Inicialização do Scheduler

types.h: Definição do novo tipo Scheduler (template)

```
// Abstractions
...
template<typename>
class Scheduler;
```

thread.h: Criação do escalonador de Threads

```
class Thread
{
private:
    ...
    static Scheduler<Thread> _scheduler;
```

thread.cc: Inicialização do escalonador de Threads

```
// Class attributes
```

```
...

Scheduler<Thread> Thread::_scheduler;
```

A API escolhida para a classe Scheduler foi a descrita no artigo de Fröhlich (2013)[1].
A próxima seção trata das modificações realizadas na Thread para o uso correto do novo escalonador e a seguinte trata da implementação da classe Scheduler.

Modificações em Thread.cc

O escalonamento de Threads, que antes estava espalhado pela classe Thread passa a ser encapsulada pelo Scheduler. Desta forma, as filas ready, suspended e a referência para a thread running são retiradas de Thread.
Os métodos de Thread foram alterados para manter a política de escalonamento anterior, porém usando o novo Scheduler<Thread>. As alterações foram as seguintes:

thread.h

```
class Thread {
...
// Class attributes
...

//Thread* volatile Thread::_running;
//Thread::Queue Thread::_ready;
//Thread::Queue Thread::_suspended;
Scheduler<Thread> Thread::_scheduler;
```

constructor_epilog

```
void Thread::constructor_epilog(const Log_Addr & entry, unsigned int
stack_size)
{
    db<Thread>(TRC) << "Thread(entry=" << entry
                    << ",state=" << _state
                    << ",priority=" << _link.rank()
                    << ",stack={b=" << reinterpret_cast<void *>(_stack)
                    << ",s=" << stack_size
                    << "},context={b=" << _context
                    << "," << *_context << "}) => " << this << endl;

    switch(_state) {
        case RUNNING:
            _scheduler.insert(this);
            break;
        case READY:
            //_ready.insert(&_link);
            _scheduler.insert(this);
            break;
        case SUSPENDED:
            //_suspended.insert(&_link);
            break;
        case WAITING:
            break;
        case FINISHING:
            break;
    }

    if(preemptive && (_state == READY) && (_link.rank() != IDLE))
        reschedule();
    else
        unlock();
}
```

~Thread()

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

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

    // The running thread cannot delete itself!
    assert(_state != RUNNING);
```

```
    switch(_state) {
    case RUNNING:  // For switch completion only: the running thread
would have deleted itself! Stack wouldn't have been released!
        exit(-1);
        break;
    case READY:
        //_ready.remove(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case SUSPENDED:
        //_suspended.remove(this);
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case WAITING:
        _waiting->remove(this);
        _scheduler.resume(this);
        _scheduler.remove(this);
        _thread_count--;
        break;
    case FINISHING: // Already called exit()
        break;
    }

    if(_joining)
        _joining->resume();

    unlock();

    delete _stack;
}
```

pass()

```
void Thread::pass()
{
    lock();

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

    Thread * prev = running();
    // prev->_state = READY;  dispatch() already setting states
    //_ready.insert(&prev->_link);
    // next->_state = RUNNING; // dispatch() already setting states
    //_ready.remove(this);
    //_state = RUNNING;
```

```
    //_running = this;

    dispatch(prev, _scheduler.choose(this));
    unlock();
}
```

suspend()

```
void Thread::suspend()
{
    lock();

    db<Thread>(TRC) << "Thread::suspend(this=" << this << ",
running="<< running() << ")" << endl;

    //if(running() != this)
        //_ready.remove(this);
    //_suspended.insert(&_link); // TODO: remove suspended queue?

    _scheduler.suspend(this);  // Might trigger a scheduler.choose()
(if this == running)
    _state = SUSPENDED;

    //if(running() == this) {
        //_running = _ready.remove()->object();
        //_running->_state = RUNNING;
        dispatch(this, _scheduler.chosen());
    //}

    unlock();
}
```

resume()

```
void Thread::resume()
{
    lock();

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

   //_suspended.remove(this);
   //_ready.insert(&_link);
   if (_state == SUSPENDED) {
       _state = READY;
       _scheduler.resume(this);
   }

   unlock();
}
```

yield()

```
void Thread::yield()
{
    lock();

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

    //prev->_state = READY;
    //_ready.insert(&prev->_link);

    //next->_state = RUNNING;

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

    Thread * prev = running();
    dispatch(prev, _scheduler.choose());

    unlock();
}
```

exit()

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

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

    Thread * prev = running();
    *reinterpret_cast<int *>(prev->_stack) = status;
    prev->_state = FINISHING;
    _scheduler.remove(prev);

    _thread_count--;

    if(prev->_joining) {
        prev->_joining->resume();
        prev->_joining = 0;
    }

    lock();

    //_running = _ready.remove()->object();
    //_running->_state = RUNNING;
    dispatch(prev, _scheduler.choose());
```

```
    unlock();
}
```

sleep()

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

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

    Thread * prev = running();
    _scheduler.suspend(prev); // Triggers a scheduler.choose()
    prev->_state = WAITING;
    prev->_waiting = q;
    q->insert(&prev->_link);

    //_running = _ready.remove()->object();
    //_running->_state = RUNNING;
    dispatch(prev, _scheduler.chosen());

    unlock();
}
```

wakeup()

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

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

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

    unlock();

    if(preemptive)
        reschedule();
```

```
}
```

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);
        _scheduler.resume(t);
    }

    unlock();

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

dispatch()

```
void Thread::dispatch(Thread * prev, Thread * next)
{
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;

        db<Thread>(TRC) << "Thread::dispatch(prev=" << prev << ",next="
<< next << ")" << endl;
        db<Thread>(INF) << "prev={" << prev << ",ctx=" <<
*prev->_context << "}" << endl;
        db<Thread>(INF) << "next={" << next << ",ctx=" <<
*next->_context << "}" << endl;

        CPU::switch_context(&prev->_context, next->_context);
    }

    unlock();
}
```

Remoção da fila *suspended*

A fila de suspensos não é necessária no contexto atual do EPOS. Ambos os métodos *suspend* e *resume* são métodos de objetos Thread. O suspend altera o estado da Thread para SUSPENDED e a coloca na fila suspended, enquanto o resume altera seu estado para READY e insere-a na fila ready.
Notou-se que a fila suspended de Thread nunca é usada. Também é um fato que, se alguém está resumindo uma thread, este alguém possui a referência para a Thread suspensa, então não se faz mais necessário salvar a referência para as Threads suspensas.
É responsabilidade do objeto externo à Thread que realizou um suspend em uma determinada Thread manter a referência desta para invocar o resume nela no futuro. Esta característica já era obrigatória devido à API do EPOS e foi reforçada com a remoção da fila *suspended* da classe Thread.

Reescalonamento (preempção)

A estratégia de preempção foi mantida: um timer com o *quantun* de tempo é iniciado e a cada chamada um *Thread::reschedule()* é invocado que, por sua vez, invoca a *Thread::yield()* e invoca o *_scheduler.choose()*, acionando o escalonador.

Implementação do Scheduler

A implementação da classe Scheduler foi baseada na API descrita no artigo de Fröhlich (2013)[1].

Uso do Scheduling_List e Criterion

A versão antiga do EPOS já continha em lists.h a definição de uma lista chamada *Scheduling_List*. Esta é uma lista ordenada de elementos baseada em um rank, denominado, neste caso, de Criterion. Criterion é um template type passado como parâmetro para o Scheduling_List que é usado para definir o ordenamento dos elementos escalonáveis. Ou seja, o tipo de elemento que se deseja escalonar deve possuir um tipo chamado Criterion (T::Criterion).
Os elementos de lista escalonáveis devem retornar um rank do tipo Criterion. Este rank será convertido para int e usado pela lista para comparar os elementos e definir a ordenação da lista.
Desta forma, para definir critérios (políticas) diferentes de escalonamento, basta criar um tipo qualquer que implemente o operador int() e definir este tipo com o nome Criterion no objeto a ser escalonado. O elemento de lista deste objeto também deve retornar um rank compatível com este critério. Então, ao ser escalonado, o objeto será inserido na *Scheduling_List* que usará o critério de escalonamento especificado.

No caso da Thread, decidimos usar o critério já existente (prioridade da thread). Para tornar o objeto compatível com o escalonador, fizemos estas modificações:

thread.h

```
class Thread {
public:
...
   // Thread Criterion for scheduling (1)
    typedef Priority Criterion;

    // Thread Queue and compatible Scheduler Element (2)
    typedef Ordered_Queue<Thread, Criterion,
Scheduler<Thread>::Element> Queue;
...
protected:
    Queue::Element _link;
```

(1) - Definimos o critério como sendo a Prioridade da Thread (unsigned int). Como este tipo já é um inteiro, não precisamos definir um novo operator int() para ele. As prioridades originais de Thread foram mantidas.

(2) - O tipo de fila padrão da Thread foi mantido. O tipo de rank da fila, porém, foi alterado para Criterion e o tipo do link da lista foi definido como o Elemento de link de lista do escalonador (definido mais adiante).

Assim, o atributo *_link* da Thread passa a ser compatível com o escalonador. Ao inserir uma thread no escalonador, seu _link será usado e, a partir dele, o critério especificado.

Implementação do Scheduler

Como toda gerência de escalonamento já está implementada em *Scheduling_List*, a implementação do Scheduler simplesmente delega para *Scheduling_List* as chamadas de funções, através de composição. Enquanto a API da lista trabalha com o link do objeto escalonado, a API do Scheduler trabalho com o objeto escalonado por si. O link do objeto escalonável é passado para lista invocando-se seu método *link()*. O tipo deste link, alterado na classe Thread, é definido em (1).

scheduler.h

```
#ifndef __scheduler_h
#define __scheduler_h

#include <system/config.h>
#include <utility/list.h>

__BEGIN_SYS

/**
```

```
* Typename T must implement T::link() and have a T::Criterion type wich
implements de operator int()
*/
template<typename T>
class Scheduler
{
protected:
    Scheduling_List<T> _list;

public:
    typedef typename Scheduling_List<T>::Element Element; // (1)

public:
    Scheduler() {
        db<Scheduler>(TRC) << "Scheduler()" << endl;
    }

    void insert(T * obj) {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
"]::insert(" << obj << ")" << endl;
        _list.insert(obj->link());
    }

    T * remove(T * obj) {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
"]::remove(" << obj << ")" << endl;
        return _list.remove(obj->link()) ? obj : 0;
    }

    T * suspend(T * obj) {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
"]::suspend(" << obj << ")" << endl;
        return _list.remove(obj->link()) ? obj : 0;
    }

    void resume(T * obj) {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
"]::resume(" << obj << ")" << endl;
        _list.insert(obj->link());
    }

    T  * chosen() {
        return _list.chosen()->object();
    }

    T * choose() {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
",queue size=" << _list.size()
            << "]::choose() ..." << endl;
```

```
        T * chosen = _list.choose()->object();

        db<Scheduler>(TRC) << "    Scheduler new chosen => " << chosen
<< endl;

        return chosen;
    }

    T * choose(T * obj) {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
",queue size=" << _list.size()
            << "]::choose(obj=" << obj << ") => ";

        T * chosen = _list.choose(obj->link())->object();

        db<Scheduler>(TRC) << "    Scheduler new chosen => " << chosen
<< endl;

        return chosen;
    }

    T * choose_another() {
        db<Scheduler>(TRC) << "Scheduler[chosen=" << chosen() <<
"]::choose_another() => ";

        T * chosen = _list.choose_another()->object();

        db<Scheduler>(TRC) << "    Scheduler new chosen => " << chosen
<< endl;

        return chosen;
    }
};

__END_SYS

#endif
```

REFERÊNCIAS

[1]Giovani Gracioli, Antônio Augusto Fröhlich, Sebastian Fischmeister and Rodolfo Pellizzoni, Implementation and Evaluation of Global and Partitioned Scheduling in a Real-Time OS, In: Real-Time Systems, 49(6):669-714, 2013.