UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO - P4

CPU Affinity Scheduling

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Critério de escalonamento Particionado

O novo critério de escalonamento para este trabalho passa a ser particionado, ou seja, ao contrário do escalonamento global, onde só existia uma fila de prontos para as Threads, no escalonamento particionado cada CPU terá sua fila própria de Threads prontas. Também dá-se a esta estratégia de escalonamento o nome de CPU Affinity.

Desta forma, uma Thread criada é colocada na fila de uma CPU qualquer, permanecerá somente na fila desta CPU, a não ser que migre. Migração de Threads, porém, não está no escopo deste trabalho, portanto, em nosso caso, uma Thread passará todo seu ciclo de vida somente na fila ou no poder da mesma CPU.

Para a implementação desta nova estratégia, foi utilizada a mesma classe *Scheduler*, desta vez estendendo a classe *Scheduling_Multilist.* Esta última precisa de alguns métodos de controle definidos no novo critério de escalonamento criado, o *PRR*.

A escolha da fila para a qual novos elementos inseridos são enviados é feita através do uso da função T::choose_queue(), sendo T um parâmetro template para o novo critério *PRR*. Esta estratégia foi utilizada já que o critério por si não tem acesso à lista ou ao escalonador. Este método é utilizado no momento de criação do critério. Cabe ao método choose_queue(), então, fazer o balanceamento de carga para os elementos inseridos na fila conforme este critério.

scheduler.h

```
namespace Scheduling_Criteria {
    ...
    // Partitioned Round-Robin
    // The typename T must have a class method choose_queue() to
    // designate the queue id for new criterion objects
    template<typename T>
    class PRR: public RR
    {
    public:
        static const unsigned int QUEUES = Traits<Machine>::CPUS;
        static const bool partitioned = true;

    public:
        PRR(int p = NORMAL): RR(p) {
            if (_priority == IDLE || _priority == MAIN)
                _queue = Machine::cpu_id();
            else
                _queue = T::choose_queue();
        }
```

```
        static unsigned int current_queue() { return Machine::cpu_id();
}
        unsigned int queue() const { return _queue; }

    private:
        unsigned int _queue;
    };
}

template<typename T>
class Scheduling_Queue<T, Scheduling_Criteria::PRR>:
public Scheduling_Multilist<T> {};
```

traits.h

```
template<> struct Traits<Thread>: public Traits<void>
{
    static const bool smp = Traits<System>::multicore;

    typedef Scheduling_Criteria::PRR<Thread> Criterion;
    static const unsigned int QUANTUM = 100000; // us

    static const bool trace_idle = true;
};
```

Balanceamento de carga (Criação de Threads)

Com o uso do novo critério de escalonamento particionado, foi necessária a implementação do método *choose_queue()* em Thread. Este método é o responsável por realizar o balanceamento de carga das Threads criadas, ou seja, inserir uma nova Thread na CPU menos ocupada no momento da criação.

thread.h

```
class Thread {
    ...
protected:
 static const bool partitioned =
Traits<Thread>::Criterion::partitioned;

 unsigned int cpu_owner() { return _link.rank().queue(); }

 static unsigned int choose_queue() {
   assert(partitioned);
   return _scheduler.underloaded_queue();
 }
```

list.h

```
class Scheduling_Multilist
{
    ...

    unsigned int underloaded_queue() {
        unsigned int queue = 0;
        unsigned int queue_size = _list[queue].size();

        for(unsigned int i = 1; i < Q; i++)
            if (_list[i].size() < queue_size) {
                queue = i;
                queue_size = _list[i].size();
            }

        return queue;
    }
```

A criação da Thread permanece igual. O critério será criado no construtor da Thread e o ID da cpu destino será associado ao critério neste momento. Em constructor_prolog(), ao inserir a nova Thread no escalonador, este usará a variável _queue de critério para definir em qual fila (CPU) colocar a nova Thread.

A única alteração na construção da Thread em si está na mudança da chamada de *reschedule()* ao novo *reschedule(cpu_id)*, que invocará um reescalonamento na CPU dona da nova Thread.

thread.cc

```
// Methods
void Thread::constructor_prolog(unsigned int stack_size)
{
    lock();
```

```
    _thread_count++;
    _scheduler.insert(this);

    _stack = new (SYSTEM) char[stack_size];
}

void Thread::constructor_epilog(const Log_Addr & entry, unsigned int
stack_size)
{
    ...

    if((_state != READY) && (_state != RUNNING))
        _scheduler.suspend(this);

    if(preemptive && (_state == READY) && (_link.rank() != IDLE))
        reschedule(cpu_owner());
    else
        if((_state == RUNNING) || (_link.rank() != IDLE)) // Keep
interrupts disabled during init_first()
            unlock(false);
        else
            unlock();
}
```

Causando reescalonamento na CPU alvo

Foi necessária a criação de um mecanismo de envio de mensagens entre CPU, mais especificamente para uma CPU qualquer poder invocar o reescalonador de outra CPU qualquer.

Isso é necessário devido ao seguinte cenário. Imagine que a CPU 0 criou uma Thread que foi colocada na fila de prontos da CPU 1. Neste momento a CPU 1 está no loop Idle. A CPU 1 continuará executando a função Idle até que o quantum acabe um reescalonamento seja acionado. Mas seria mais interessante que, imediatamente após a inclusão da nova Thread na fila de prontos, a CPU 1 fosse avisada e um reescalonamento ocorresse, assim ganhamos desempenho.

A forma deste envio de mensagem é através de interrupções. Criamos uma nova interrupção associada ao número IC::INT_RESCHEDULER e passamos o *time_slicer* como handler. Assim, quando a interrupção acontecer em uma CPU qualquer, o time_slicer será acionado.

thread_init.cc

```
void Thread::init() {
    ...

    if(Criterion::partitioned && (Machine::cpu_id() == 0)) {
        IC::int_vector(IC::INT_RESCHEDULER, time_slicer);
        IC::enable(IC::INT_RESCHEDULER);
    }
}
```

O próximo passo é a geração da interrupção. Fazemos isso através do método *IC::ipi_send(cpu_id, interruption_id)*, que envia uma interrupção qualquer à uma CPU. Desta forma, criamos um novo método *reschedule* em Thread que recebe por parâmetro a CPU onde o reescalonamento deve acontecer. No corpo do método, geramos a interrupção *IC::INT_RESCHEDULER* na CPU alvo, onde o *time_slicer* (handler da interrupção) será acionado.

thread.h

```
Thread {
...
protected:
    static void reschedule(unsigned int target_cpu);
```

thread.cc

```
void Thread::reschedule(unsigned int target_cpu) {
    db<Thread>(TRC) << "Thread::reschedule(target_cpu=" << target_cpu
<< ")" << endl;

    if (target_cpu != Machine::cpu_id()) {
        IC::ipi_send(target_cpu, IC::INT_RESCHEDULER);
        unlock(); // Since reschedule() also has an implicit unlock()
    } else
        reschedule();
}
```

Fazemos também uma checagem para que, caso a CPU atual seja igual à CPU alvo, um *reschedule* padrão seja chamado, já que não precisamos utilizar interrupção neste caso. Também utilizamos um *unlock()* para manter o comportamento do reschedule padrão, já que trocamos as ocorrências de chamada a *reschedule* (que já consideravam um unlock implícito) para *reschedule(cpu_id)*.

Novas chamadas a *reschedule(cpu_id)*

Todas as chamadas a *reschedule()* foram revisadas para levar em conta o fato de que sua invocação deve ser realizada tendo como alvo a CPU com afinidade à Thread manipulada no momento.

resume()

Em *Thread::resume()*, a CPU que invocou o método para resumir uma Thread qualquer pode não ser a CPU com afinidade àquela Thread. Portanto, trocamos o método *reschedule()* ao novo *reschedule(cpu_id)* passando o id da CPU dona da Thread.

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

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

    if(_state == SUSPENDED) {
        _state = READY;
        _scheduler.resume(this);

        if(preemptive)
            reschedule(cpu_owner());
    } else {
        db<Thread>(WRN) << "Resume called for unsuspended object!" <<
endl;

        unlock();
    }
}
```

wakeup()

O método *wakeup()*, responsável por acordar uma Thread e retorná-la à fila READY, teve a invocação de *reschedule*() modificada para ocorrer na CPU com afinidade à Thread que está sendo acordada.

```
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;
        _scheduler.resume(t);

        if(preemptive)
            reschedule(t->cpu_owner());
    } else
        unlock();
}
```

wakeup_all()

A mesma justificativa de *wakeup()* vale para *wakeup_all()*. Para cada Thread acordada um *reschedule()* será invocado para a CPU dona desta Thread.

```
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());

    if(!q->empty())
        while(!q->empty()) {
            Thread * t = q->remove()->object();
            t->_state = READY;
            t->_waiting = 0;
            _scheduler.resume(t);

            if(preemptive) {
                reschedule(t->cpu_owner());
                lock();
            }
         }
    else
        unlock();
}
```

priority()

O método setter *priority()* altera a prioridade da Thread construindo um novo critério e atribuindo ao rank do link da Thread. Como um novo critério será construído, uma nova fila será escolhida para o critério o que pode fazer com que a Thread migre (troque de CPU). Para garantir que a antiga CPU dona da Thread e a nova sejam avisadas, utilizamos o *reschedule(cpu_id)*.

thread.cc

```
void Thread::priority(const Priority & c)
{
    lock();

    db<Thread>(TRC) << "Thread::priority(this=" << this << ",prio=" <<
c << ")" << endl;

    unsigned int prev_owner = cpu_owner();

    _link.rank(Criterion(c));

    if(_state != RUNNING) {
        _scheduler.remove(this);
        _scheduler.insert(this);
    }

    if(preemptive) {
        reschedule(prev_owner); // implicit unlock()
        lock();
        reschedule(cpu_owner());
    }
}
```