UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO - P5

Completely Fair Scheduler

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Contabilizando tempo das Threads

Para definir a justiça utilizamos o seguinte critério: o tempo de CPU (tempo em que cada Thread permanece na CPU) deve ser semelhante para Threads que nasceram juntas. No caso do jantar dos filósofos, onde cada uma das 5 Threads é criada simultaneamente, o tempo de CPU de cada Thread esperado, considerando-se justiça, deve ser praticamente igual, independente do número de CPUs do sistema.

Para testar este aspecto, modificamos a classe Thread a fim de contabilizar o tempo total de CPU para cada objeto Thread. Ao final da execução do programa, imprimimos o resultado.

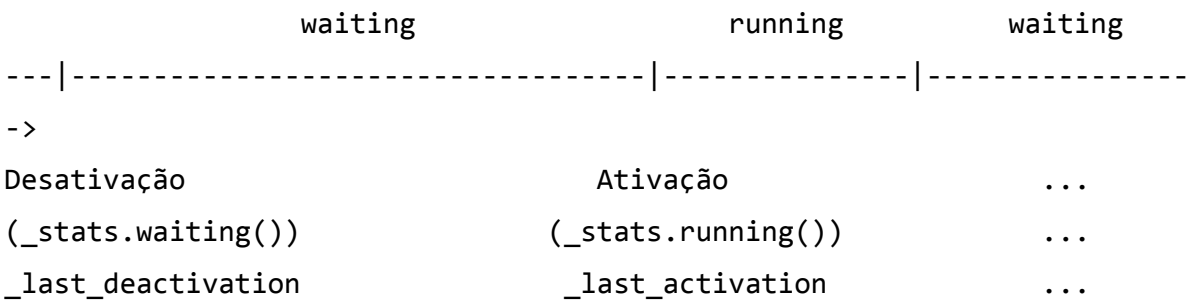
Exemplo de execução do parallel_philosophers_dinner com 4 CPUs contabilizando tempo de CPU de cada Thread:

```
The Philosopher's Dinner:
Philosophers are alive and hungry!

                                done[1]

                           \                /

                    done[1]                  done[2]

                           /                \

                      done[0]      |      done[3]


The dinner is served ...
Philosopher 0 total running time: 6.086s
Philosopher 1 total running time: 8.759s
Philosopher 2 total running time: 8.880s
Philosopher 3 total running time: 8.510s
Philosopher 4 total running time: 5.918s
Philosopher 0 ate 10 times
Philosopher 1 ate 10 times
Philosopher 2 ate 10 times
Philosopher 3 ate 10 times
Philosopher 4 ate 10 times
The end!
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
```

Pode-se notar que as Threads 1, 2 e 3 possuem tempo semelhante de CPU, porém as Threads 0 e 4 possuem tempos bem menores. Isso acontece porque as Thread 0 e 4 estão compartilhando a CPU 0, enquanto as demais monopolizam sua própria CPU.
O objetivo das modificações a seguir, portanto, foi igualar o tempo de CPU de cada Thread utilizando uma política de *fair scheduling*, ocasionalmente migrando Threads para rebalanceamento de carga, mas sem quebrar drasticamente a CPU affinity.

Ciclo de vida e Prioridade Dinâmica

Para a contabilização das estatísticas de tempo de execução e de espera das Threads, definimos uma nomenclatura para os eventos pertinentes do ciclo de vida da Thread. Uma *ativação* ocorre quando uma Thread entra na CPU, enquanto uma *desativação* acontece quando ela deixa uma CPU.
Ao ser ativada ou desativada estatísticas pertinentes de tempo de uso e de espera são atualizadas. A prioridade dinâmica é calculada com base nestas estatísticas.
Exemplo do ciclo de vida de Thread e atualização de estatísticas:



Este ciclo de vida e os métodos correspondentes foram mapeados para a função *dispatch*. Considera-se que a Thread *prev* esteja deixando a CPU (desativando) e a Thread *next* esteja tomando a CPU (ativando). Os métodos *_stats.waiting* e *_stats.running* são responsáveis por atualizar internamente as estatísticas da Thread.

```
void Thread::dispatch(Thread * prev, Thread * next, bool charge)
{
    ...

    prev->_stats.waiting();
    next->_stats.running();
```

```
    prev->update_priority();

    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;

        next->update_priority();

        if (   prev->link()->rank() != IDLE
            && prev->_stats.activations() >= ATTEMPTS_BEFORE_MIGRATE)
        {
            prev->migrate();
        }

        db<Thread>(TRC) << "Thread::dispatch(prev=" << prev << ",next="
<< next << ")" << endl;

        if(smp)
            _lock.release();

        CPU::switch_context(&prev->_context, next->_context);
    } else {
        if(smp)
            _lock.release();
    }

    CPU::int_enable();
}
```

Stats

A finalidade da nova classe Stats criada é contabilizar as estatísticas do ciclo de vida de um Thread. Nela são guardadas as informações de média de tempo de execução e de espera e, a partir destes, são calculadas as novas prioridades da Thread.

A classe é utilizando através dos métodos *running()* e *waiting()*. O primeiro sinaliza uma *ativação* da Thread, ou seja, sempre que a Thread entrar na CPU este método é invocado, atualizando informações de tempo médio de espera e salvando o momento atual como a última ativação. O método *waiting()*, por sua vez, sinaliza uma *desativação* da Thread, atualizando informações de tempo de execução e salvando o momento atual como a última desativação.

Além disso, a classe Stats salva o tempo total de execução da Thread em cada CPU, para fins de depuração e verificação da justiça do escalonamento, principalmente no contexto desta atividade.

A construção do Stats é definida de tal forma que novas Threads começam no estado WAITING. O método first_init(), porém, é utilizado pelas primeiras Threads que serão lançadas, logo após a inicialização do sistema. Ele é chamado em init_first(), imediatamente antes do contexto da primeira Thread ser carregado.

thread.h

```
class Thread {
    ...
    class Stats {
    public:
        typedef TSC::Time_Stamp Time_Stamp;
        typedef RTC::Microsecond Microsecond;

        enum State {
            RUNNING, WAITING
        };

    public:
        Stats() : _last_deactivation(now()),
                  _last_activation(0),
                  _total_running_time{0},
                  _waiting_mean(0),
                  _running_mean(0),
                  _state(WAITING),
                  _activations(0) {}

        static Microsecond now() {
            return IA32_TSC::time_stamp() * 1000000 / CPU::clock();
        }

        void first_init() {
            _last_deactivation = now();
            _last_activation = _last_deactivation;
            _state = RUNNING;
        }

        void waiting() {
            assert(_state == RUNNING);
            _state = WAITING;
            _last_deactivation = now();
            _running_mean = (_running_mean + running_time()) / 2;
            _total_running_time[Machine::cpu_id()] +=
_last_deactivation - _last_activation;
        }
```

```
        void running() {
            assert(_state == WAITING);
            _state = RUNNING;
            _last_activation = now();
            _waiting_mean = (_waiting_mean + waiting_time()) / 2;
            _activations++;
        }

        static Microsecond ideal_waiting_time() {
            if (Thread::count() == 0)
                return 0;

            float run_proportion = Machine::n_cpus() / Thread::count();
            float wait_proportion = 1 - run_proportion;

            return wait_proportion * QUANTUM;
        }

        unsigned int activations() { return _activations; }

        void reset_activations() { _activations = 0; }

        Microsecond waiting_mean() const { return _waiting_mean; }

        Microsecond running_mean() const { return _running_mean; }

        Microsecond waiting_time() const {
            return _last_activation - _last_deactivation;
        }

        Microsecond running_time() const {
            return now() - _last_activation;
        }

        Microsecond * total_running_time() {
            return _total_running_time;
        }

        Microsecond total_running_time_all() const {
            Microsecond total_time = 0;
            for (unsigned int i = 0; i < Traits<Build>::CPUS; i++) {
                total_time += _total_running_time[i];
            }
            return total_time;
        }

    protected:
        Microsecond _last_deactivation; // last time thread has left
the CPU
```

```
        Microsecond _last_activation;   // last time thread has
obtained the CPU
        Microsecond _total_running_time[Traits<Build>::CPUS];
        Microsecond _waiting_mean;
        Microsecond _running_mean;
        State _state;
        unsigned int _activations;
    };
}
```

init_first.cc

```
Init_First() {
    ...
    first->_stats.first_init();
    first->_context->load();
}
```

Atualização da Prioridade

A atualização da prioridade dinâmica das Threads é feita com base no seu tempo médio de espera em comparação com o tempo ideal (teórico) de espera. O tempo ideal de espera é calculado considerando-se que, em um processador totalmente justo, o tempo de execução seja igualmente distribuído entre as Threads.

A diferença entre o tempo ideal de espera e a média atual de espera da Thread é a diferença de prioridade que será atribuída à Thread. Se a Thread esperou mais do que o ideal, esta diferença será negativa e a prioridade será aumentada. Se a Thread esperou menos do que o ideal, esta diferença será positiva e a prioridade será rebaixada. A base sobre o qual este valor será incrementado é o valor de NORMAL.

thread.h

```
void Thread::update_priority() {
    if (link()->rank() == IDLE)
        return;

    int new_priority = NORMAL + Stats::ideal_waiting_time() -
_stats.waiting_mean();

    assert(new_priority < IDLE);
    assert(new_priority > 0);

    db<Thread>(TRC) << "Thread::update_priority[this=" << this
        << ",old_priority=" << link()->rank()
        << ",new_priority=" << new_priority
```

```
        << "]" << endl;

    link()->rank(Criterion(new_priority, cpu_owner()));
}
```

Migração da Thread

O método *migrate* é responsável por migrar a Thread para a CPU mais disponível no momento. O id da CPU com menos carga no momento é obtido com *less_user_cpu()*, que será explicado adiante. Para migrar a Thread, seu critério é criado novamente com a prioridade antiga e a nova CPU. Caso a Thread esteja no estado READY, ela é retirada e reinserida no escalonador. Fazemos isso para que a Thread de fato saia da fila da CPU atual e seja inserida na nova.

Caso a Thread não esteja READY, a reinserção não ocorre. Considere o caso em que ela foi posta para dormir, por exemplo. O migrate pode ter sido chamado neste caso e a Thread foi inserida na fila de um semáforo e retirada da fila ready da CPU. Neste caso, alteramos o valor de CPU (fila atual) da Thread, mas não a reinserimos, já que isto ocasionaria na Thread sendo posta na fila do escalonador em um estado de espera.

```
void Thread::migrate() {
      db<Thread>(TRC) << "Thread::migrate(this=" << this
                  << ",prev_cpu=" << cpu_owner() << ")" << endl;

      if (_state == READY)
          _scheduler.remove(this);

      link()->rank(Criterion(link()->rank(), less_used_cpu()));

      if (_state == READY)
            _scheduler.insert(this);

      db<Thread>(TRC) << "Thread::migrate(this=" << this
                  << ",target_cpu=" << this->cpu_owner() << ")" <<
endl;

      _stats.reset_activations();
}
```

O método *less_used_cpu()* é responsável por retornar o ID da CPU com menos carga no momento. Definimos esta CPU usando o tempo médio de execução das Threads Idles. A

CPU cuja Idle rodou mais nos últimos ciclos é a CPU com menos carga. Obtemos o ID desta CPU iterando o array de Idles do sistema.

```
unsigned int Thread::less_used_cpu() {
      unsigned int cpu = 0;
      Stats::Microsecond max_mean = 0;

      db<Thread>(TRC) << "Thread::less_used_cpu()" << endl;

      for (unsigned int i = 0; i < Traits<Build>::CPUS; i++) {
          Stats::Microsecond mean = _idles[i]->_stats.running_mean();
          db<Thread>(TRC) << "     idle[" << i << "] | running_mean = "
<< mean << endl;

         if (mean >= max_mean) {
            cpu = i;
            max_mean = mean;
         }
      }

      db<Thread>(TRC) << "   less_used_cpu = [" << cpu << "] |
running_mean = " << max_mean << endl;

      return cpu;
}
```

As referências às Idles são mantidas da seguinte forma:

thread.h

```
Thread {
    ...
    static Thread * _idles[Traits<Build>::CPUS];
```

thread.cc

```
Thread * Thread::_idles[];

// Methods
void Thread::constructor_prolog(unsigned int stack_size)
{
    ...
    if (_link.rank() == IDLE){
      _idles[Machine::cpu_id()] = this;
    }
    ...
}
```

Novo critério de escalonamento: CFS

Foi criado um novo critério de escalonamento, o CFS. Nele foram redefinidos os valores de prioridade para a metade do valor máximo (IDLE) para que haja uma margem suficiente para trabalharmos com adição ou subtração de prioridade. Todas as prioridades passam a ter o mesmo valor, já que estamos buscando um escalonamento justo. Apenas a IDLE é mantida com a mais baixa prioridade, já que continuamos querendo este comportamento associado a ela.

A MAIN também começa com um valor diferente, para podemos utilizar o condicional dentro do construtor do CFS, atribuindo a MAIN à CPU atual, já que optamos por continuar criando a MAIN na CPU 0 em um primeiro momento, mas podendo migrá-la após a inicialização do sistemas e após a atualização de sua prioridade.

Uma outra opção de construtor também foi definido para podermos criar um novo critério com determinada prioridade setando a fila de nosso interesse.

scheduler.h

```
namespace Scheduling_Criteria
{
    ...
    // Completely Fair Scheduler
    // The typename T must have a class method choose_queue() to
    // designate the queue id for new criterion objects
    template<typename T>
    class CFS: public RR
    {
    public:
        enum {
            IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1,
            MAIN   = IDLE / 2 - 1,
            HIGH   = IDLE / 2,
            NORMAL = IDLE / 2,
            LOW    = IDLE / 2
        };

        static const unsigned int QUEUES = Traits<Machine>::CPUS;
        static const bool partitioned = true;

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

    public:
        CFS(int p = NORMAL): RR(p), _changes(0) {
```

```
            if (_priority == IDLE || _priority == MAIN)
                _queue = Machine::cpu_id();
            else
                _queue = T::choose_queue();
        }

        CFS(int p, unsigned int target_queue): RR(p), _changes(0) {
            _queue = target_queue;
        }

    protected:
        unsigned int _queue;
        unsigned int _changes;
    };
}

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

Resultados

```
The Philosopher's Dinner:
Philosophers are alive and hungry!


                              done[3]

                         \                /


                  done[0]                 done[2]


                         /                \


                         done[3]     |     done[1]


The dinner is served ...
Philosopher 0 ate 10 times  time: 11187539
Philosopher 1 ate 10 times  time: 11371883
Philosopher 2 ate 10 times  time: 10324760
Philosopher 3 ate 10 times  time: 9698329
Philosopher 4 ate 10 times  time: 11478121
```

```
Philosopher    C_0       C_1         C_2         C_3

0           0                 419877           0           10779517
1           0                 0                11386432          0
2           0                 10342471         0                 0
3           0                 0                0                 9767779
4           11518845          0                0                 0

Philosopher 0 total running time: 11199394
Philosopher 1 total running time: 11386432
Philosopher 2 total running time: 10342471
Philosopher 3 total running time: 9767779
Philosopher 4 total running time: 11518845
The end!
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
```