INE5424 - SO II - P4: CPU Affinity Scheduling

Alunos:

- Glaucia de Pádua da Silva - 09232087
- Quenio Cesar Machado dos Santos - 14100868

Verificando Tempo de Execução em CPUs

Para melhor entender como se dá a alocação das CPUs para cada política de escalonamento, modificamos a classe `Thread` para capturar o tempo de execução dos `threads` em cada CPU. Também modificamos o programa "jantar dos filósofos" para imprimir os tempos ao encerrar sua execução.

As mudanças na classe `Thread` estão a seguir:

```
class Thread
{
   ...
   typedef Scheduler_Timer Timer;

public:
   ...
   int total_tick(int cpu_id) { return _total_tick_[cpu_id]; }

private:
   ...
   Timer::Tick _tick_count_, _total_tick_[Traits<Build>::CPUS];
   ...
}
...
void Thread::dispatch(Thread * prev, Thread * next, bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer_->reset();
```

```
    }

    if(prev != next) {
        next->_tick_count_ = Timer::tick_count();
        if (prev->_state_ == RUNNING || prev->_state_ == FINISHING) {
          prev->_total_tick_[Machine::cpu_id()] += (next->_tick_count_ - prev->_tick_count_);
        }

        if(prev->_state_ == RUNNING) {
            prev->_state_ = READY;
        }
        next->_state_ = RUNNING;

        spinUnlock();

        CPU::switch_context(&prev->_context_, next->_context_);
    } else {
      spinUnlock();
    }
```

```
    CPU::int_enable();
}
```

No código acima, observe que o método `tick_count(cpu_id)` irá retornar o tempo em milisegundos que um `Thread` levou para executar numa determinada CPU. O monitoramento e cálculo do tempo é feito no método `dispatch()`.

O programa do "jantar dos filósofos" utiliza o método `tick_count()` para apresentar os tempos de todos os `threads` - um por "filósofo" - ao final de sua execução. Veja abaixo:

```
int main()
{
  ...
  cout << "The dinner is served ..." << endl;
  table.unlock();

  Timer::Tick total_cpu_tick[Traits<Build>::CPUS];
  for(int i = 0; i < 5; i++) {
      int ret = phil[i]->join();
```

```
    table.lock();
    Display::position(20 + i, 0);
    cout << "Philosopher " << i;
    Timer::Tick total_thread_tick = 0;
    for (int cpu_id = 0; cpu_id < Machine::n_cpus(); cpu_id++) {
      Timer::Tick tick_per_cpu = phil[i]->total_tick(cpu_id);
      total_thread_tick += tick_per_cpu;
      total_cpu_tick[cpu_id] += tick_per_cpu;
      cout << " | " << cpu_id << ": " << Spaced(tick_per_cpu);
    }
    cout << " | T: " << Spaced(total_thread_tick) << endl;
    table.unlock();
}
table.lock();
Display::position(25, 0);
cout << "CPU Totals   ";
Timer::Tick total_tick = 0;
for (int cpu_id = 0; cpu_id < Machine::n_cpus(); cpu_id++) {
  Timer::Tick tick_per_cpu = total_cpu_tick[cpu_id];
  total_tick += tick_per_cpu;
  cout << " | " << cpu_id << ": " << Spaced(tick_per_cpu);
}
cout << " | T: " << Spaced(total_tick) << endl << endl;
table.unlock();
...
}
```

Alocação de CPU com Multi-Head Round-Robin

No trabalho P3, nós implementamos o round-robin usando uma fila `multi-head` para que as CPUs pudessem compartilhar a fila de `threads`. Isto fez com que os `threads` fossem executados em todas as CPUs, como se vê na saída abaixo:

```
qemu-system-i386 -smp 4 -m 262144k -nographic -no-reboot -fda  phil_dinner.img | tee phil_dinner.out
Setting up this machine as follows:
  Processor:    IA32 at 4002 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261752 Kbytes [0x00000000:0x0ff9e000]
```

```
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21344 bytes
  APP code:     32608 bytes    data: 640 bytes
  CPU count:    4

The Philosopher's Dinner:
Philosophers are alive and hungry!
even numbers: 15000-3

                                done   2

                           \               /


                     done   3               done   3


                           /               \

                               done   0 |     done   1


 The dinner is served ...
 Philosopher 0 | 0:  1235 | 1:   190 | 2:  4096 | 3:   110 | T:  5631
 Philosopher 1 | 0:  1364 | 1:   138 | 2:  3862 | 3:   241 | T:  5605
 Philosopher 2 | 0:  3359 | 1:   110 | 2:  2598 | 3:   110 | T:  6177
 Philosopher 3 | 0:  1495 | 1:   107 | 2:  1292 | 3:  1480 | T:  4374
 Philosopher 4 | 0:  1200 | 1:   290 | 2:  1358 | 3:  2028 | T:  4876
 CPU Totals    | 0:  8653 | 1:   835 | 2: 13206 | 3:  3969 | T: 26663
```

Observamos o seguinte nesta saída:

- Cada um dos `threads` foi executado em todas as quatro CPUs disponíveis.
- O tempo total de execução de cada `thread` variou bastante.
- O tempo de uso de cada CPU também variou.
- Tempo total de execução em todas as CPUs foi 26s.

CPU-Bound

O problema com a execução acima é que, ao migrar a execução de cada `thread` entre as CPUs, o escalonador está gerando uma série de `cache misses`. Os `caches` das CPUs são invalidados toda vez que o `working set` de um `thread` precisa migrar para outra CPU. Isto causa a queda no desempenho de execução.

Para sanar este problema, implementamos uma versão do round-robin que é `CPU-bound`:

```
...
template<typename T, typename R = typename T::Criterion>
class Scheduling_Queue: public Scheduling_Multilist<T> {};
...
class CPU_Bound
{
public:
  static const unsigned int QUEUES = Traits<Machine>::CPUS;

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

public:
  CPU_Bound(unsigned int queue = current_queue()): _queue_(queue) {}

  const volatile unsigned int & queue() const volatile { return _queue_; }

private:
  volatile unsigned int _queue_;
};
...
class CPU_Bound_RR: public RR, public CPU_Bound
{
public:
  CPU_Bound_RR(int p = NORMAL, unsigned int queue = current_queue())
    : RR(p), CPU_Bound(queue) {}
};
...
template<> struct Traits<Thread>: public Traits<void>
{
  ...
```

```
    typedef Scheduling_Criteria::CPU_Bound_RR Criterion;
    ...
  };
  ...
```

Observe acima que `Scheduling_Queue` agora é uma multi-lista. Isto permite que cada CPU tenha sua própria fila. Quando for entregar o próximo `thread` a `Scheduler`, a classe `Scheduling_Multilist` vai usar o método `current_queue()` de `CPU_Bound` para escolher a fila da CPU corrente.

Além disso, observe também que o método `queue()` diz em que fila (ou CPU) o `Thread` foi alocado. Por `default`, `CPU_Bound_RR` vai colocar um `thread` novo na mesma CPU do `thread` que o criou.

A saída abaixo mostra o compartamento de `CPU_Bound_RR`:

```
qemu-system-i386 -smp 4 -m 262144k -nographic -no-reboot -fda  phil_dinner.img | tee phil_dinner.out
Setting up this machine as follows:
  Processor:    IA32 at 452 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261752 Kbytes [0x00000000:0x0ff9e000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21344 bytes
  APP code:     32976 bytes    data: 704 bytes
  CPU count:    4

The Philosopher's Dinner:
Philosophers are alive and hungry!
even numbers: 15000-0

                                done   0

                          \                /

                    done   0                done   0

                          /                \
```

```
                        done   0 |     done   0


The dinner is served ...
Philosopher 0 | 0:  5488 | 1:     0 | 2:     0 | 3:     0 | T:  5488
Philosopher 1 | 0:  5353 | 1:     0 | 2:     0 | 3:     0 | T:  5353
Philosopher 2 | 0:  5175 | 1:     0 | 2:     0 | 3:     0 | T:  5175
Philosopher 3 | 0:  5642 | 1:     0 | 2:     0 | 3:     0 | T:  5642
Philosopher 4 | 0:  5094 | 1:     0 | 2:     0 | 3:     0 | T:  5094
CPU Totals    | 0: 26752 | 1:     0 | 2:     0 | 3:     0 | T: 26752

The end!
The last thread has exited!
Rebooting the machine ...
```

Observe na saída acima que todos os threads acabaram executando somente na CPU zero. Isto se deu porque foi a partir da CPU zero que os demais `threads` foram criados, de acordo com o compartamento `default` de `CPU_Bound_RR`.

No entanto, é possível alterar em qual CPU um `thread` será executado:

```
int main()
{
    table.lock();
    Display::clear();
    Display::position(0, 0);
    cout << "The Philosopher's Dinner:" << endl;

    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;

    phil[0] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 0)), // CPU 0
      &philosopher, 0,  5, 32);
    phil[1] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 1)), // CPU 1
      &philosopher, 1, 10, 44);
    phil[2] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 2)), // CPU 2
```

```
      &philosopher, 2, 16, 39);
    phil[3] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 3)), // CPU 3
      &philosopher, 3, 16, 24);
    phil[4] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 3)), // CPU 3 - novamente
      &philosopher, 4, 10, 20);

    cout << "Philosophers are alive and hungry!" << endl;
    ...
}
```

Observe a seguinte alocação para os `threads` do "filósofos":

- `thread` zero na CPU zero;
- `thread` um na CPU um;
- `thread` dois na CPU dois;
- `threads` três e quatro na CPU três.

Ao executar o "jantar dos filósofos" após estas alterações:

```
qemu-system-i386 -smp 4 -m 262144k -nographic -no-reboot -fda  phil_dinner.img | tee phil_dinner.out
Setting up this machine as follows:
  Processor:    IA32 at 422 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261752 Kbytes [0x00000000:0x0ff9e000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21344 bytes
  APP code:     33296 bytes    data: 704 bytes
  CPU count:    4

The Philosopher's Dinner:
Philosophers are alive and hungry!
even numbers: 15000-3

                                  done   0
```

```
                          \               /

                    done   3               done   1

                          /               \

                         done   3 |     done   2

The dinner is served ...
Philosopher 0 | 0:  1166 | 1:     0 | 2:     0 | 3:     0 | T:  1166
Philosopher 1 | 0:     0 | 1:  1071 | 2:     0 | 3:     0 | T:  1071
Philosopher 2 | 0:     0 | 1:     0 | 2:  1108 | 3:     0 | T:  1108
Philosopher 3 | 0:     0 | 1:     0 | 2:     0 | 3:  6797 | T:  6797
Philosopher 4 | 0:     0 | 1:     0 | 2:     0 | 3:  6421 | T:  6421
CPU Totals    | 0:  1166 | 1:  1071 | 2:  1108 | 3: 13218 | T: 16563

The end!
The last thread has exited!
Rebooting the machine ...
```

Na saída acima, vale observar que:

- A alocação de CPUs para cada `thread` foi respeitada pelo escalonador.
- O tempo total de execução dos `threads` 0, 1 e 2 diminuiu bastante quando comparado à execução anterior. Isto ocorreu provavelmente porque não houve troca de contexto e `cache misses`.
- Entretanto, os `threads` 3 e 4, que compartilharam a mesma CPU, continuaram levando mais tempo. Provavelmente devido as trocas de contexto e `cache invalidation`.
- O tempo total de todas as CPUs diminuiu de 26s para 16.5s.

Demora na Iniciação dos `Threads`

Agora vamos observar o tempo que leva para cada `thread` iniciar na CPU em que foi alocado. Para isto, foi feita mais uma modificação no "jantar dos

filósofos":

```
...
volatile Timer::Tick s[5];
...
int philosopher(int n, int l, int c)
{
    s[n] = (Timer::tick_count() - s[n]);
    ...
}
...
int main()
{
    ...
    s[0] = Timer::tick_count(0);
    phil[0] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 0)),
      &philosopher, 0,  5, 32);

    s[1] = Timer::tick_count(1);
    phil[1] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 1)),
      &philosopher, 1, 10, 44);

    s[2] = Timer::tick_count(2);
    phil[2] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 2)),
      &philosopher, 2, 16, 39);

    s[3] = Timer::tick_count(3);
    phil[3] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 3)),
      &philosopher, 3, 16, 24);

    s[4] = Timer::tick_count(3);
    phil[4] = new Thread(
      Thread::Configuration(Thread::READY, Thread::Criterion(Thread::NORMAL, 3)),
      &philosopher, 4, 10, 20);
```

```
    cout << "Philosophers are alive and hungry!" << endl;
    ...
}
```

O vetor `s` acima vai guardar o tempo que levou para iniciar cada um dos `threads`. Na saída abaixo, temos mais uma execução do "jantar dos filófosos" com `CPU_Bound_RR`, mas agora com o tempo de iniciação:

```
qemu-system-i386 -smp 4 -m 262144k -nographic -no-reboot -fda  phil_dinner.img | tee phil_dinner.out
Setting up this machine as follows:
  Processor:    IA32 at 926 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261752 Kbytes [0x00000000:0x0ff9e000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21344 bytes
  APP code:     33488 bytes    data: 704 bytes
  CPU count:    4

The Philosopher's Dinner:
Philosophers are alive and hungry!
even numbers: 15000-3

                                done   0

                            \               /


                      done   3                done   1

                            /               \


                         done   3 |     done   2


The dinner is served ...
Philosopher 0 | 0:   828 | 1:     0 | 2:     0 | 3:     0 | T:   828 | S: 3
Philosopher 1 | 0:     0 | 1:   762 | 2:     0 | 3:     0 | T:   762 | S: 94
```

```
Philosopher 2 | 0:     0 | 1:     0 | 2:   864 | 3:     0 | T:   864 | S: 95
Philosopher 3 | 0:     0 | 1:     0 | 2:     0 | 3:  4635 | T:  4635 | S: 95
Philosopher 4 | 0:     0 | 1:     0 | 2:     0 | 3:  4721 | T:  4721 | S: 98
CPU Totals    | 0:   828 | 1:   762 | 2:   864 | 3:  9356 | T: 11810

The end!
The last thread has exited!
Rebooting the machine ...
```

É possível observar que:

- A execução do `thread` zero na CPU zero é quase imediata.
- Todos os outros `threads` demoram em torno de 100ms para iniciar sua execução, o que equivale ao `QUANTUM` do escalonador.

Forçando Escalonamento em Outra CPU

Fica claro na última saída da seção anterior que é preciso mudar a implementação para que os `threads` de 1 a 4 possam iniciar sua execução imediatamente.

As seguintes alterações resolvem este problema:

```
...
void Thread::init()
{
    // Thread::init() deve ser chamado somente pelo BSP.
    assert(Machine::cpu_id() == 0);

    if(Criterion::timed)
        _timer_ = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

    Thread::init_rescheduler();
}

// Deve ser chamado pelas APs via System::init_rescheduler()
void Thread::init_rescheduler()
{
  CPU::int_disable();
```

```
  // Habilita escalonamento para ipi_send(cpu_id, IC::INT_RESCHEDULER)
  IC::int_vector(IC::INT_RESCHEDULER, time_slicer);

  // Para x86 APIC, na verdade, não há necessidade de habilitar interrupções individualmente.
  // Chamando aqui para manter o padrão e para suportar outras arquiteturas.
  IC::enable(IC::INT_RESCHEDULER);

  CPU::int_enable();
}
...
void System::init()
{
    if(Traits<Alarm>::enabled)
        Alarm::init();

    if(Traits<Thread>::enabled)
        Thread::init();
}
```

```
void System::init_rescheduler()
{
  // Deve ser chamado somente pelas APs.
  assert(Machine::cpu_id() > 0);

  if(Traits<Thread>::enabled)
    Thread::init_rescheduler();
}
...
Init_System() {
    db<Init>(TRC) << "Init_System()" << endl;

    Machine::smp_barrier();

    // Only the boot CPU runs INIT_SYSTEM fully
    if(Machine::cpu_id() != 0) {
        // Wait until the boot CPU has initialized the machine
        Machine::smp_barrier();

        // For IA-32, timer is CPU-local. What about other SMPs?
```

```
    Timer::init();

    if(Traits<Thread>::enabled)
      System::init_rescheduler();

    Machine::smp_barrier();
    return;
}

...

Machine::smp_barrier(); // signalizes "machine ready" to other CPUs

// Initialize system abstractions
db<Init>(INF) << "Initializing system abstractions: " << endl;
System::init();
db<Init>(INF) << "done!" << endl;

...

    Machine::smp_barrier();

    // Initialization continues at init_first
}
...
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)) {
        if (_link_.rank().queue() != Machine::cpu_id())
          reschedule_cpu(_link_.rank().queue());
        else {
          reschedule(); // implicit unlock
          return;
        }
    }

    unlock();
```

```
}
...
void Thread::reschedule_cpu(int cpu_id)
{
  IC::ipi_send(cpu_id, IC::INT_RESCHEDULER);
}
...
static const unsigned int IRQS = 16;
static const unsigned int HARD_INT = 32;
static const unsigned int SOFT_INT = HARD_INT + IRQS;
...
enum {
    INT_FIRST_HARD  = HARD_INT,
    INT_TIMER    = HARD_INT + IRQ_TIMER,
    INT_KEYBOARD    = HARD_INT + IRQ_KEYBOARD,
    INT_LAST_HARD   = HARD_INT + IRQ_LAST,
    INT_RESCHEDULER = SOFT_INT,
    INT_SYSCALL
};
...
```

No código acima, vemos que a interrupção `IC::RESCHEDULER` foi abilitada para executar `Thread::time_slicer()`, ou seja, força um escalonamento. Esta interrupção é disparada de uma CPU para outra quando a primeira cria um `thread` que foi alocado à segunda.

Abaixo, veja o tempo de iniciação dos `threads` com a solução acima:

```
qemu-system-i386 -smp 4 -m 262144k -nographic -no-reboot -fda  phil_dinner.img | tee phil_dinner.out
Setting up this machine as follows:
  Processor:    IA32 at 758 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261752 Kbytes [0x00000000:0x0ff9e000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21344 bytes
  APP code:     33632 bytes    data: 704 bytes
  CPU count:    4

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

```
even numbers: 15000-3

                                done   0

                            \               /

                        done   3               done   1

                            /               \

                           done   3 |     done   2


The dinner is served ...
Philosopher 0 | 0:   765 | 1:     0 | 2:     0 | 3:     0 | T:   765 | S: 8
Philosopher 1 | 0:     0 | 1:   744 | 2:     0 | 3:     0 | T:   744 | S: 1
Philosopher 2 | 0:     0 | 1:     0 | 2:   726 | 3:     0 | T:   726 | S: 1
Philosopher 3 | 0:     0 | 1:     0 | 2:     0 | 3:  2663 | T:  2663 | S: 1
Philosopher 4 | 0:     0 | 1:     0 | 2:     0 | 3:  4275 | T:  4275 | S: 1
CPU Totals    | 0:   765 | 1:   744 | 2:   726 | 3:  6938 | T:  9173

The end!
The last thread has exited!
Rebooting the machine ...
```

Observe na saída acima que os `threads` de 1 a 4 agora iniciam sua execução imediatamente.

Distribuição Automática entre as CPUs

A implementação `CPU_Bound_RR` exige que o criador do `Thread` especifique em qual CPU este deve rodar. Abaixo vemos uma implementação que distribui os `threads` automaticamente entre as CPUs disponíveis:

```
...
template<> struct Traits<Thread>: public Traits<void>
```

```
{
    ...
    typedef Scheduling_Criteria::CPU_Distribution_RR Criterion;
    ...
};
...
class CPU_Distribution: public CPU_Bound
{
public:
  CPU_Distribution(unsigned int queue = next_queue()): CPU_Bound(queue) {}

protected:
  static unsigned int next_queue()
  {
    CPU::finc(_next_queue_);
    return (_next_queue_-1) % Machine::n_cpus();
  }

private:
  static volatile unsigned int _next_queue_;
};
...
class CPU_Distribution_RR: public RR, public CPU_Distribution
{
public:
  CPU_Distribution_RR(int p = NORMAL)
    : RR(p), CPU_Distribution(p == MAIN || p == IDLE ? Machine::cpu_id() : next_queue()) {}
};
...
```

A classe `CPU_Distribution` herda de `CPU_Bound` e implementa o método `next_queue()` que irá distribuir os `threads` entre as CPUs. A classe `CPU_Distribution_RR` vai certificar-se que `threads` do tipo `MAIN` e `IDLE` sempre serão alocados a CPU que os criou.

Veja abaixo a saída com a implementação de `CPU_Distribution_RR`:

```
qemu-system-i386 -smp 4 -m 262144k -nographic -no-reboot -fda  phil_dinner.img | tee phil_dinner.out
Setting up this machine as follows:
  Processor:    IA32 at 1074 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
```

```
  User memory:  261752 Kbytes [0x00000000:0x0ff9e000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21344 bytes
  APP code:     33776 bytes    data: 704 bytes
  CPU count:    4

The Philosopher's Dinner:
Philosophers are alive and hungry!
even numbers: 15000-0

                                done   0

                           \               /


                      done   0               done   1


                           /               \

                          done   3 |     done   2


The dinner is served ...
Philosopher 0 | 0:  4296 | 1:     0 | 2:     0 | 3:     0 | T:  4296 | S: 6
Philosopher 1 | 0:     0 | 1:   770 | 2:     0 | 3:     0 | T:   770 | S: 2
Philosopher 2 | 0:     0 | 1:     0 | 2:   821 | 3:     0 | T:   821 | S: 1
Philosopher 3 | 0:     0 | 1:     0 | 2:     0 | 3:   775 | T:   775 | S: 1
Philosopher 4 | 0:  4027 | 1:     0 | 2:     0 | 3:     0 | T:  4027 | S: 6
CPU Totals    | 0:  8323 | 1:   770 | 2:   821 | 3:   775 | T: 10689

The end!
The last thread has exited!
Rebooting the machine ...
```

Conclusão

A alocação de `threads` a CPUs permite otimizar sua execução através da conservação de `cache` e outros recursos da CPU. Esta alocação pode ser feita pelo desenvolvedor da aplicação ou de forma automática pelo sistema.

Como um `thread` pode ser alocado para uma CPU diferente daquela que o criou, é necessário implementar um mecanismo de comunicação com a outra CPU para forçar o escalonamento do novo `thread`.

Através de uma fila de múltiplas listas, a alocação de `threads` a CPUs pode ser combinada com várias políticas de escalonamento, tais como round-robin e prioridade.