INE5424 - SO II - P3: Parallel Philosopher's Dinner

Alunos:

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

Multihead Queue & Round-Robin

A classe `Scheduling_Queue`, que é a classe base de `Scheduler`, agora herda da classe `Multihead_Scheduling_List`. Esta última permite `Scheduler` ter várias cabeças em sua fila. Com as modificações feitas na classe `Round-Robin`, cada cabeça corresponderá a uma CPU. Desta forma, `Scheduler` poderá distribuir a alocação de `threads` entre as CPUs disponíveis. Veja abaixo, a nova classe base de `Scheduling_Queue`:

```
// Scheduling_Queue
template<typename T, typename R = typename T::Criterion>
class Scheduling_Queue: public Multihead_Scheduling_List<T> {};

// Scheduler
// Objects subject to scheduling by Scheduler must declare a type "Criterion"
// that will be used as the scheduling queue sorting criterion (viz, through
// operators <, >, and ==) and must also define a method "link" to export the
// list element pointing to the object being handled.
template<typename T>
class Scheduler: public Scheduling_Queue<T>
{
  ...
```

Para que `Scheduler` funcione com uma `Multihead_Scheduling_List`, foi preciso alterar a política round-robin para definir que o número de cabeças na lista corresponde ao número de CPUs e também qual a cabeça da fila corresponde a CPU em execução. Veja abaixo:

```
// Round-Robin
class RR: public Priority
{
```

```
public:
    enum {
        MAIN   = 0,
        NORMAL = 1,
        IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
    };

    static const bool timed = true;
    static const bool dynamic = false;
    static const bool preemptive = true;

    static const unsigned int HEADS = Traits<Build>::CPUS;

    static int current_head() { return Machine::cpu_id(); }

public:
    RR(int p = NORMAL): Priority(p) {}
};
```

Observe que `HEADS` acima é a constante que informa a `Scheduler` o número de cabeças necessárias (uma para cada CPU), enquanto `current_head()` vai informar qual é a cabeça para a CPU em execução.

Spin Lock & Escalonamento

Para que o escalonamento de `threads` funcione corretamente num ambiente multi-processado, foi preciso modificar o mecanismo de `locking` na classe `Thread`. Desabilitar as interrupções do processador corrente não é suficiente, pois as outras CPUs poderiam executar as seções críticas da classe `Thread`.

Para solucionar este problema, foi utilizado a classe `Spin`, que implementa um `spin lock` utilizando a primitiva de sincronização `cas - compare-and-swap`. Na arquitetura IA32, a primitiva `cas` é implementada pela instrução `lock cmpxchgl` que garante atomicidade e coerência de cache entre todas as CPUs do sistema.

Veja abaixo as alterações na classe `Thread` referente ao métodos `lock()` e `unlock()`:

```
class Thread
{
```

```
    ...
    static void lock()
    {
      CPU::int_disable();
      spinLock();
    }

    static void unlock()
    {
      spinUnlock();
      CPU::int_enable();
    }
    ...
private:
    static void spinLock() { if (smp) _spinLock.acquire(); }
    static void spinUnlock() { if (smp) _spinLock.release(); }

 protected:
    static Spin _spinLock;
    ...
};
```

Além das alterações acima, também foi preciso modificar o método `Thread::dispatch()` para que este libere o `spin lock` antes de mudar de contexto:

```
void Thread::dispatch(Thread * prev, Thread * next, bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();
    }

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

        spinUnlock();

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

```
    } else {
      spinUnlock();
    }

    CPU::int_enable();
}
```

Sem as alterações acima, todas as CPUs ficariam bloqueadas pelo `spin lock` e nenhum `thread` poderia executar, ou seja, `thread starvation`.

Correção em `Heap`

No método `enter()` da classe `Heap_Wrapper`, moveu-se a desabilitação das interrupções para antes de adquirir o `spin lock`. Isto evita que haja uma troca de contexto para outro `thread` sem a liberação da `spin lock`, o que poderia bloquear todas CPUs que estejam alocando memória na `heap`. Veja abaixo:

```
class Heap_Wrapper ...
{
  ...
  void enter() {
      CPU::int_disable();
      _lock.acquire();
  }
  ...
}
```

Correção em `init_first.cc`

Em `init_first`, quando criando os `idle threads` das APs, marcando o `idle thread` como `RUNNING`, visto que inicialmente seu contexto será carregado em `init_first`, ao invés de ser escalonado:

```
if(Machine::cpu_id() == 0) {
    first = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING, Thread::MAIN), reinterpret_cast<int (*)()>(__epos_app_entry));
```

```
    // Idle thread creation must succeed main, thus avoiding implicit rescheduling.
    new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::IDLE), &Thread::idle);
} else
    first = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING, Thread::IDLE), &Thread::idle);
```

Correção em `init_system.cc`

Foi feita a equalização das barreiras em `init_system.cc` entre a BSP e as APs para garantir que todas as CPUs:

```
--- src/init/init_system.cc    (revision 3782)
+++ src/init/init_system.cc    (working copy)
         // 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();
             Machine::smp_barrier();
@@ -66,6 +66,8 @@
             db<Init>(INF) << "done!" << endl;
         }

+        Machine::smp_barrier();
+
         // Initialization continues at init_first
     }
 };
```

Término da Execução

Uma vez que todos os `threads` terminarem sua execução, somente os `idle threads` estarão rodando nas CPUs. Por esta razão, o termino da execução do sistema se dá em `Thread::idle()`. Aqui somente a BSP irá desligar a máquina. As outras CPUs vão apenas desativar-se - `halt()` - mas antes desabilitam suas interrupções para evitar que sejam acordadas novamente.

```
int Thread::idle()
{
    while(_thread_count > Machine::n_cpus()) { // someone else besides idles
        if(Traits<Thread>::trace_idle)
          db<Thread>(WRN) << 'i' << Machine::cpu_id();

        CPU::int_enable();
        CPU::halt();
    }

    CPU::int_disable();
    if(Machine::cpu_id() == 0) {
        db<Thread>(WRN) << "The last thread has exited!" << endl;
        if(reboot) {
            db<Thread>(WRN) << "Rebooting the machine ..." << endl;
            Machine::reboot();
        } else
            db<Thread>(WRN) << "Halting the machine ..." << endl;
    }
    CPU::halt();

    return 0;
}
```

Verificação dos mecanismos de `locking` do EPOS

Foi feita uma verificação dos mecanismos de `locking` de todo a sistema para guarantir seu funcionamento num ambiente multi-processado. Veja a seguir nas sub-seções abaixo.

`Mutex`

O método `lock()` de `Mutex` utiliza a primitiva atomica `tsl - test-and-set -` que vai retornar o valor atual de `_lock` e depois marcá-lo. Na arquitetura IA32, `tsl` é implementada pela instrução `lock xchg` que garante atomicidade e coerência de cache entre todas as CPUs do sistema. Veja baixo:

```
void Mutex::lock()
```

```
{
    db<Synchronizer>(TRC) << "Mutex::lock(this=" << this << ")" << endl;

    begin_atomic();
    if(tsl(_locked))
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}

void Mutex::unlock()
{
    db<Synchronizer>(TRC) << "Mutex::unlock(this=" << this << ")" << endl;

    begin_atomic();
    if(_queue.empty()) {
        _locked = false;
        end_atomic();
    } else
        wakeup(); // implicit end_atomic()
}
```

Além disso, `begin_atomic()` e `end_atomic()`, implementados respectivamente por `Thread::lock()` e `Thread::unlock()` em `synchronizer.h`, protegem a execução de `sleep()` e `wakeup()`, e do acesso a `_queue` de espera.

Semaphore

Da mesma forma que no caso do `Mutex`, `Semaphore` utiliza `begin_atomic()` e `end_atomic()` para proteger seus métodos `p()` e `v()`.

Para incrementar seu contador, `Semaphore` utiliza as primitivas atômicas `finc()` e `fdec()` que garantem a coerência de cache entre todas as CPUs, pois ambas utilizam a instrução `lock xadd` da arquitetura IA32.

Veja a seguir de `p()` e `v()`:

```
void Semaphore::p()
{
    db<Synchronizer>(TRC) << "Semaphore::p(this=" << this << ",value=" << _value << ")" << endl;
```

```
    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}

void Semaphore::v()
{
    db<Synchronizer>(TRC) << "Semaphore::v(this=" << this << ",value=" << _value << ")" << endl;

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else
        end_atomic();
}
```

Alarm

O handler de interrupção que dispara `Alarm::handler()` é implementado em `PC_Timer`. Este somente vai disparar os alarmes na BSP:

```
void PC_Timer::int_handler(const Interrupt_Id & i)
{
    ...

    if((!Traits<System>::multicore || (Traits<System>::multicore && (Machine::cpu_id() == 0))) && _channels[ALARM]) {
        _channels[ALARM]->_current[0] = _channels[ALARM]->_initial;
        _channels[ALARM]->_handler(i);
    }

    ...
}
```

Portanto, poderia-se pensar que não haveria a necessidade de usar um `spin lock` para os alarmes, mas `Thread::lock()/unlock()` são usados para processar os `requests` de alarmes porque é preciso haver sincronização com os outros métodos de `Alarm`:

```
void Alarm::handler(const IC::Interrupt_Id & i)
{
    lock();

    _elapsed++;
    Alarm * alarm = 0;

    if(!_request.empty()) {
        if(_request.head()->promote() <= 0) { // rank can be negative whenever multiple handlers get created for the same time tick
            Queue::Element * e = _request.remove();
            alarm = e->object();
            if(alarm->_times != INFINITE)
                alarm->_times--;
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
        }
    }

    unlock();

    if(alarm) {
        (*alarm->_handler)();
    }
}
```

O Jantar dos Filósofos em 8 CPUs

Dada as verificações e alterações acima, podemos agora fazer algumas modificações no programa "jantar dos filófosos" para demonstrar a execução adequada em múltiplos processadores.

Primeiramente, vamos alterar a configuração para rodar o programa em 8 CPUs:

```
--- include/system/traits.h    (revision 3782)
+++ include/system/traits.h    (working copy)
```

```
@@ -28,7 +28,7 @@
     enum {Legacy};
     static const unsigned int MODEL = Legacy;

-    static const unsigned int CPUS = 1;
+    static const unsigned int CPUS = 8;
     static const unsigned int NODES = 1; // > 1 => NETWORKING
 };
```

A seguir, vamos alterar as mensagens "thinking" e "eating" para também imprimir o `id` da CPU:

```
table.lock();
Display::position(l, c);
cout << "thinking " << Machine::cpu_id();
table.unlock();
...
table.lock();
Display::position(l, c);
cout << " eating  " << Machine::cpu_id();
table.unlock();
```

Para garantir a plena utilização de todas as CPUs, vamos implementar um algoritmo a ser executado quando o "filófoso" estiver "comendo":

```
bool is_even(int n)
{
  bool result = true;
  while (n > 0) {
      n--;
      result = !result;
  }
  return result;
}
...
int count_even_numbers()
{
  int n = 0;
  for(int i = 0; i < 30000; i++) {
    if (is_even(i)) n++;
```

```
  }
  return n;
}
...
int philosopher(int n, int l, int c)
{
  ...
  chopstick[first]->p();    // get first chopstick
  chopstick[second]->p();   // get second chopstick

  table.lock();
  Display::position(l, c);
  cout << " eating  " << Machine::cpu_id();
  table.unlock();

  n = count_even_numbers();
  table.lock();
  Display::position(3, 0);
  cout << "even numbers: " << n << "-" << Machine::cpu_id();
  table.unlock();

  chopstick[first]->v();    // release first chopstick
  chopstick[second]->v();   // release second chopstick
  ...
}
```

E também outro algoritmo para quando o "filósofo" estiver "pensando":

```
int count_odd_numbers()
{
  int *n = new (SYSTEM) int;

  *n = 0;
  for(int i = 0; i < 20000; i++) {
    if (!is_even(i)) (*n)++;
  }

  int result = *n;
  delete n;
```

```
  return result;
}
...
int philosopher(int n, int l, int c)
{
  ...
  table.lock();
  Display::position(l, c);
  cout << "thinking " << Machine::cpu_id();
  table.unlock();

  int n = count_odd_numbers();
  table.lock();
  Display::position(3, 0);
  cout << " odd numbers: " << n << "-" << Machine::cpu_id();
  table.unlock();
  ...
}
```

Observe acima que também estamos testando a `heap` com as alocações de `n` em `count_odd_numbers()`.

Após executar o programa com as modificações acima, observou-se que este roda de maneira confiável inúmeras vezes, sempre distribuindo a execução entre as várias CPUs:

```
Setting up this machine as follows:
  Processor:    IA32 at 3098 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261688 Kbytes [0x00000000:0x0ff8e000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21376 bytes
  APP code:     31536 bytes    data: 672 bytes
  CPU count:    8
[2J[000;000HThe Philosopher's Dinner:
Philosophers are alive and hungry!
[007;044H/[013;044H\[016;035H|[013;027H/[007;027H\[019;000HThe dinner is served ...
[005;032Hthinking 0[010;044Hthinking 0[016;039Hthinking 0[016;024Hthinking 2[010;020Hthinking 1[003;000H odd numbers: 10000-5[005;032H ea
[021;000HPhilosopher 1 ate 4 times
```

```
[022;000HPhilosopher 2 ate 4 times
[023;000HPhilosopher 3 ate 4 times
[003;000Heven numbers: 15000-3[010;020H  done   3[024;000HPhilosopher 4 ate 4 times
The end!
The last thread has exited!
Rebooting the machine ...
```

Aqui está a captura do estado final de uma execução:

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

                                done   6

                           \               /


                     done   3               done   5

                           /               \


                     done   1 |     done   4


The dinner is served ...
Philosopher 0 ate 4 times
Philosopher 1 ate 4 times
Philosopher 2 ate 4 times
Philosopher 3 ate 4 times
Philosopher 4 ate 4 times
The end!
The last thread has exited!
Rebooting the machine ...
```