INE5424 - SO II - P2: Parallel Idle Threads

Alunos:

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

Tarefa

O design atual do EPOS exige que sempre exista uma `thread` para ser executada. Para tal, na versão `monocore`, foi criada uma `Thread Idle`. Esta `thread` é executada sempre que não há outras `threads` para serem executadas, colocando a CPU em modo de baixo consumo de energia até que um evento externo ocorra. Agora, no cenário de `multicore`, o mesmo princípio deve ser preservado ou, alternativamente, o sistema deve ser reprojetado. Esta etapa do projeto deve conduzi-lo até a função `main()` da aplicação mantendo todos os demais cores ativos. Os testes devem ser executados com o QEMU emulando 8 CPUs.

Solução

Implementando o programa de teste

O programa de teste apenas causa uma espera de meio-segundo que permite cada um dos `idle threads` executarem antes do programa terminar.

Para poder visualizar quais `idle threads` estão executando num determinado momento, a versão inicial do método `Thread::idle()` foi modificada para imprimir `Machine::cpu_id()` enquanto estiver no `idle loop`:

```
while(_thread_count > 1) {

    db<Thread>(WRN) << Machine::cpu_id(); // identificando a CPU em idle.

    CPU::int_enable();
    CPU::halt();
```

```
}
```

Para a execução com apenas uma CPU, a saída do programa é a seguinte:

```
Setting up this machine as follows:
  Processor:    IA32 at 2498 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261800 Kbytes [0x00000000:0x0ffaa000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        19456 bytes
  APP code:     17536 bytes    data: 480 bytes
  CPU count:    1

Esperando...
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000...TCHAU!
The last thread has exited!
Rebooting the machine ...
```

Observe que os zeros são impressos dentro do `idle loop` porque existe apenas uma CPU (cpu_id = 0) executando em `idle` .

Executando com duas CPUs

Ainda usando a versão inicial do EPOS disponibilizada pelo professor para P2, vamos executar o programa de testes com duas CPUs:

```
Setting up this machine as follows:
  Processor:    IA32 at 797 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261784 Kbytes [0x00000000:0x0ffa6000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
```

```
  Setup:        20992 bytes
  APP code:     17504 bytes    data: 512 bytes
  CPU count:    2

Esperando...
1
Esperando...
IC::exc_gpf(cs=8,ip=0x00003f50,fl=2)
The running thread will now be termin1ated!
10010101010101010010101010101001010010101010101010101010101110101010101010101
00101010101010101010101010101010101010101010101010101010101010101010101010101010
10101010110101010101010101010101010101010101010101010101010101010101010101010101
...
10101010101010101010101010101010101010101010101010101010101010101010010101010101010101
01010101010101010101010101010101010101010101010101010101010101010101010101010101
01010101010101010101010101011001010101010101010101010101010101010101010101010101
01010101010101010101010101...TCHAU!
00101010101010101010101010101010101010101010101010101010101010010101010101010101010101
01010101010101010101010101010101010101010101010101010101010101010101010101010101
...
01010101010101010101010101010101010101010101010101010101010101010101010101010101
01010101010100101010101010110101010101010101010101010101010101010101010101010101
01010101010101010101010101010101IC::exc_pf[address=0x66333038](cs=8,ip=0x00001b23,
fl=46,err=PR)
The running thread will now be terminated!
IC::exc_pf[address=0x53e5895d](cs=8,ip=0x00002065,fl=2,err=)
The running thread will now be terminated!
IC::exc_pf[address=0x53e58971](cs=8,ip=0x00002dac,fl=97,err=)
The running thread will now be terminated!
IC::exc_pf[address=0x53e58971](cs=8,ip=0x00002dac,fl=97,err=)
...
IC::exc_pf[address=0x53e58971](cs=8,ip=0x00002dac,fl=97,err=)
The running thread will now be terminated!
IC::exc_pf[address=0x53e58971](cs=8,ip=0x00002dac,fl=97,err=)
ThThe last thread has exited!
Rebooting the machine ...
e
```

Na saída do programa acima, existem alguns pontos importantes para se observar:

- O programa foi iniciado e executado duas vezes concorrentemente, uma vez em cada CPU. Observe o texto `Esperando...` sendo impresso duas.
- Logo no inicio da execução do programa, abaixo do segundo `Esperando...`, ocorreu um `general protection fault` (GPF - indicado pelo texto `IC::exc_gpf`). A função do EPOS que cuida de GPFs tenta terminar a execução do programa, mas ele continua executando.
- Logo após o GPF, os `idle threads` das duas CPUs ficam executando em seu `idle loop`, o que é observado através da alternância entre zeros e uns impressos na saída.
- Dado o tempo ainda maior que o meio-segundo de espera programado, a execução do programa termina em ambas as CPUs.
- Antes de terminar a execução, porém, vê-se uma séria de `page faults` (PFs).

As observações acima mostram os seguintes problemas com a versão do EPOS disponilizada para este trabalho:

- A inicialização do sistema não está tomando o cuidado de iniciar a execução do programa em apenas uma CPU.
- A GPF indica acesso de memória fora do seguimento, o que deve estar ocorrendo quando as CPUs tentam acesso concorrente a `heap` do sistema, ou a outros recursos compartilhados.
- Ambos as CPUs provavelmente estavam tentando desalocar memória, ou outros resources, ao término da execução do programa, o que deve ter causado as PFs.

As próximas seções irão mostrar mudanças progressivas sobre a versão inicial do código para resolver estes problemas.

Modificando a inicialização do `thread` principal e dos `idle threads`

Como vimos anteriormente, ambas as CPUs estavam executando o `thread` principal. As alterações abaixo em `init_frst.cc` fazem com que apenas a BSP inicie o `thread` principal. As PAs vão iniciar apenas um `idle thread`:

```
Thread * first;
if (Machine::cpu_id() == 0) {
  // If EPOS is not a kernel, then adjust the application entry point to __epos_app_entry,
  // which will directly call main(). In this case, _init will have already been called,
  // before Init_Application, to construct main()'s global objects.
  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);
}
```

Além das mudanças acima, também foi colocado uma barreira em `init_first.cc` logo após a instanciação dos `threads` para garantir que nenhum `thread` venha iniciar sua execução antes das outras:

```
  db<Init>(INF) << "INIT ends here!" << endl;

  db<Init, Thread>(WRN) << "Dispatching the first thread: " << first << " on CPU: " << Machine::cpu_id() << endl;

  This_Thread::not_booting();

  Machine::smp_barrier(); // todas as threads iniciarão juntas

  first->_context->load();
}
```

Observe abaixo a saída do programa de teste com duas CPUs:

```
Setting up this machine as follows:
  Processor:    IA32 at 822 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261784 Kbytes [0x00000000:0x0ffa6000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21792 bytes
  APP code:     18256 bytes    data: 544 bytes
```

```
  CPU count:    2

Esperand1o na CPU 0...
1
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111...
```

Assim como esperado, depois das alterações em `init_first.cc`, somente a CPU 0 está rodando a `thread` principal, como mostrado pelo texto `Esperand1o na CPU 0...`. Porém, observe que somente a CPU 1 está executando o `idle thread`. A CPU 0 também deveria estar executando em seu `idle thread`, mas esta já terminou sua execução neste ponto. A proxima seção vai cuidar desta questão.

Modificando a política de escalonamento para suportar mais de uma CPU

A configuração atual do `Scheduler` utiliza a política `round-robin`, que é adequada para apenas uma CPU. Porém, no caso SMP, é preciso que `Scheduler.chosen()` retorne um `thread` diferente para cada CPU. Portanto, é necessário alterar a configuração de `Scheduler` para utilizar uma política adequada em SMP.

Para tanto, modificou-se `Schedeling_Queue` - a classe base de `Scheduler` - para usar `Scheduling_Multilist` como sua base. Esta última permite políticas que alocam uma lista de `threads` para cada CPU:

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

Observe que `Scheduling_Multilist` não estava disponível na versão disponilizada para o trabalho P2, mas foi copiada do EPOS 2.

Também foi necessário implementar uma política nova, a qual chamamos de `uniform distribution` porque aloca um `idle thread` para cada CPU e também distribui igualmente a alocação de `threads` normais entre as CPUs:

```
// Uniform Distribution
class UD: public Priority
{
public:
    enum {
        MAIN   = 0,
        NORMAL = 1,
        IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
    };

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

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

public:
    UD(int p = NORMAL): Priority(p), _queue(((_priority == IDLE) || (_priority == MAIN)) ? Machine::cpu_id() : ++_next_queue %= Machine::

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

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

private:
      volatile unsigned int _queue;
      static volatile unsigned int _next_queue;
};
```

Como no código acima se usa `Machine::n_cpus()` para distribuir `threads` entre as várias filas, foi preciso inicializar `_n_cpus` chamando `smp_init()`, como mostrado abaixo:

```
void PC::init()
{
    db<Init, PC>(TRC) << "PC::init()" << endl;

    if(Traits<PC_IC>::enabled)
        PC_IC::init();
```

```
    if(Traits<PC_PCI>::enabled)
        PC_PCI::init();

    if(Traits<PC_Timer>::enabled)
        PC_Timer::init();

    if(Traits<PC_Scratchpad>::enabled)
        PC_Scratchpad::init();

    if(smp) {
      System_Info<PC> * si = reinterpret_cast<System_Info<PC> *>(Memory_Map<PC>::SYS_INFO);
      smp_init(si->bm.n_cpus);
    }
}
```

Observe na saida abaixo que agora ambos `idle threads` estão executando:

```
Setting up this machine as follows:
  Processor:    IA32 at 543 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261784 Kbytes [0x00000000:0x0ffa6000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21120 bytes
  APP code:     18240 bytes    data: 544 bytes
  CPU count:    2

1Esper1ando na CPU 0...
10101010001010101010010100101010101010101010101010101010101010101010101010100110101010
10101010101010101010101001010110101010100110101010101010101010101010101001010101010101
01010101000101010100101010101010101010101010101001010101010101010100101010101010101010
01010101010101010101010101010101010101010101011010100101010101010101010101010101010101
01010101010100110111010100101010101010101001101010101010101010101010111010101010100101
01010101010101110101010101010101010101010100101010101000101010100101010101010101010110
11010101010101001101010111010101001010100101010100101010110101010101001010101010100
11010101010101000101010101010101001010101010101001101010101010101010101010101010101
01010101010101001101010101010101010010101001010101011010101010101010101010101010
10101010101010101001101010101010101010101010010101010010101010101010101010010010
10100100101010101001010101010101010101010100101010101010101010101010100110101010
```

```
...
```

Apesar do progresso, a execução do sistema ainda não está terminando depois de meio-segundo de espera do programa de teste. Este problema será resolvido na próxima seção.

Modificando `Thread::idle`

Para que o programa termine sua execução e o sistema seja "desligado" é preciso que os `idle threads` parem de executar quando não houver mais `threads` normais para executar no sistema. Para tanto, `Thread::idle` foi modificado como mostrado abaixo:

```
int Thread::idle()
{
    while(_thread_count > Machine::n_cpus()) { // someone else besides idle
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(this=" << running() << ")" << endl;

        db<Thread>(WRN) << Machine::cpu_id();  // identificando a CPU em idle.
        CPU::int_enable();
        CPU::halt();
    }
    ...
```

Observe que no código acima, verificamos se o número de `threads` em execução é maior que o número de CPUs. Neste caso, os `idle threads` continuam a executar normalmente. Caso contrário, o sistema pode desligar porque não faz sentido continuar rodando apenas `idle threads` em todas as CPUs.

A saída abaixo mostra o resultado esperado:

```
Setting up this machine as follows:
  Processor:    IA32 at 543 MHz (BUS clock = 125 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261784 Kbytes [0x00000000:0x0ffa6000]
  PCI aperture: 44996 Kbytes [0xfc000000:0xfebf1000]
  Node Id:      will get from the network!
  Setup:        21120 bytes
```

```
  APP code:     18240 bytes    data: 544 bytes
  CPU count:    2

1Esper1ando na CPU 0...
10101010001010101010010100101010101010101010101010101010101010101010100110101010
10101010101010101010101001010110101010100110101010101010101010101010010101010101
01010101000101010100101010101010101010101010100101010101010101010010101010101010
01010101010101010101010101010101010101010101011010100101010101010101010101010101
01010101010100110111010100101010101010100110101010101010101010101011010101010100
01010101010101110101010101010101010101010100101010101001010101010010101010101010
11010101010101001101010111010101001010100101010100101010111010101010010101010100
11010101010101000101010101010100101010101010100110101010101010101010101010101010
01010101010101001101010101010101010100101010010101010111010101010101010101010101
10101010101010101001101010101010101010101010010101001010101010101010101010010010
10100100101010101001010101010101010101010100101010101010101010101010101001101010
10101010100101010101010011010101010101010111010101001010101010101010101010101010
01010101010101...TCHAU!
The last thread has exited on CPU 0...
Rebooting the machine on CPU 0...
The last thread has exited on CPU 1...
Rebooting the machine on CPU 1...
```

Próximos Passos

- Executar com 4+ CPUs; resolver problemas de inicialização.
- Modificar o timer para cada CPU, para evitar contenção no escalonamento.