UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO

Parallel Idle Threads

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Enunciado

O design atual do EPOS exige que sempre existe 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.

Os principais critérios de avaliação para a atividade são:

- Criação de uma thread Idle para cada core;
- Ajustes no escalonador para que todos os cores possam acessar a fila Ready (única, em um modelo global de escalonamento).

Prelúdio

Atomizando o Output Stream

O primeiro passo para a resolução de todas as tarefas multicores foi atomizar o output stream. Ao rodar o EPOS com mais de uma CPU e com informações de debug ativadas, todas as CPUs tentam escrever na tela ao mesmo tempo, originando informações muitas vezes não legíveis e difíceis de identificar qual CPU as originou.

Desta forma, atomizamos o output stream de forma que, ao começar a escrever, a CPU adquira acesso único ao output, e só a libere ao terminar de escrever. Enquanto isso, outras CPUs que requisitem escrita ficarão bloqueadas esperando sua vez.

Implementamos este comportamento na classe *OStream* do EPOS, utilizando o método *take* ao imprimir uma nova linha no output (Begl) e o método *release* ao imprimir um fim de linha no output (Endl).

A estratégia utilizada foi o uso de um busy waiting inspirado no Spin lock do EPOS. Tivemos que utilizar o método CPU::cas, que utiliza uma instrução assembly atômica para multicores, *compare and exchange*, que associa um owner ao output atual e bloqueia o acesso a todas as CPUs não-owners.

ostream.h

```
class OStream
{
private:
```

```
    volatile int _owner;
   …
public:
    OStream(): _base(10), _error(false), _owner(-1) {}
    …

    OStream & operator<<(const Begl & begl) {
        take();
        return *this << "[cpu=" << _owner << "]    ";
    }

    OStream & operator<<(const Endl & endl) {
        print("\n");
        _base = 10;
        release();
        return *this;
    }
```

ostream.cc

```
#include <machine.h>

void OStream::take()
{
    // We cannot use Spin lock here

    int me = Machine::cpu_id();

    // Compare and exchange:
    // Atomically compare _owner and -1. If they're equal, replace
_owner by 'me' and return the new value of '_owner'
    // Otherwise don't replace anything and return the current value of
'_owner'
    while(CPU::cas(_owner, -1, me) != me);
}

void OStream::release()
{
    // -1 means: no body 'owns' the output stream
    _owner = -1;
}
```

Inicialização

O ligamento de todas as CPUs e algumas inicializações de sistema e de componentes de hardware acontece no *pc_setup*, que já estava completo na versão do enunciado deste trabalho. Após este setup, todas as CPUs caem na função *call_next*, que desloca o fluxo de

execução para a próxima etapa de inicialização. No nosso caso, todas as CPUs criam um objeto Init_System e passam a executá-lo.

A ordem de inicialização que cada CPU percorre, portanto, é a seguinte:

pc_setup → Init_System → Init_Application → Init_First

A partir de Init_System foram realizadas as seguintes modificações:

init_system.cc

O Init_System é responsável por inicializar a CPU, Machine, a System Heap e as abstrações do sistema. A primeira função utilizada, *Machine::smp_init* é a responsável por setar as informações de ids das CPUs e número de CPUs no sistema. Essas informações foram perdidas após a liberação da memória associada ao *pc_setup*.

Todas as inicializações foram atribuídas somente para a CPU 0 executar.

init_system.cc

```
   Init_System() {
      // Set n_cpus and cpu_id:
      Machine::smp_init(System::info()->bm.n_cpus);

      db<Init>(TRC) << "Init_System(n_cpus=" << Machine::n_cpus()
            << ",cpu_id=" << Machine::cpu_id() << ")" << endl;

      Machine::smp_barrier();

      if (Machine::cpu_id() == 0) {
            // Initialize the processor
            db<Init>(INF) << "Initializing the CPU: " << endl;
            CPU::init();
            db<Init>(INF) << "done!" << endl;

            // Initialize System's heap
            db<Init>(INF) << "Initializing system's heap: " << endl;
            if(Traits<System>::multiheap) {
                  System::_heap_segment = new (&System::_preheap[0])
Segment(HEAP_SIZE, Segment::Flags::SYS);
                  System::_heap = new
(&System::_preheap[sizeof(Segment)])
Heap(Address_Space(MMU::current()).attach(System::_heap_segment,
Memory_Map<Machine>::SYS_HEAP), System::_heap_segment->size());
            } else
                  System::_heap = new (&System::_preheap[0])
Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE);
            db<Init>(INF) << "done!" << endl;

            // Initialize the machine
```

```
            db<Init>(INF) << "Initializing the machine: " << endl;
            Machine::init();
            db<Init>(INF) << "done!" << endl;

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

            // Randomize the Random Numbers Generator's seed
            if(Traits<Random>::enabled) {
                  db<Init>(INF) << "Randomizing the Random Numbers
Generator's seed: " << endl;
                  if(Traits<TSC>::enabled)
                  Random::seed(TSC::time_stamp());

                  if(!Traits<TSC>::enabled)
                  db<Init>(WRN) << "Due to lack of entropy, Random is a
pseudo random numbers generator!" << endl;
                  db<Init>(INF) << "done!" << endl;
            }
      }
      Machine::smp_barrier();
}
```

init_application.cc

Init_Application é responsável por alocar a Heap de aplicação do EPOS, representada pela variável estática *Application::_heap*. Esta heap de aplicação é única para o sistema, portanto foi adicionada uma condição para que somente a CPU de inicialização a execute, fazendo a heap de aplicação ser iniciada somente uma vez.

init_application.cc

```
 Init_Application() {
      if (Machine::cpu_id() == 0) {
            db<Init>(TRC) << "Init_Application()" << endl;

            // Initialize Application's heap
            db<Init>(INF) << "Initializing application's heap: " <<
endl;
            if(Traits<System>::multiheap) { // Heap in data segment
arranged by SETUP
                  char * stack = MMU::align_page(&_end);
                  char * heap = stack +
MMU::align_page(Traits<Application>::STACK_SIZE);
```

```
                Application::_heap = new (&Application::_preheap[0])
Heap(heap, HEAP_SIZE);
            } else
                for(unsigned int frames = MMU::allocable(); frames;
frames = MMU::allocable())
                System::_heap->free(MMU::alloc(frames), frames *
sizeof(MMU::Page));
            db<Init>(INF) << "done!" << endl;
      }
}
```

init_first.cc

Init_First é responsável por criar as threads iniciais de cada CPU e carregar seus respectivos contextos para iniciar sua execução. A variável *Thread * first* representa a primeira Thread que será carrega por cada CPU, imediatamente após o término de Init_First.

Com múltiplas CPUs, devemos criar a thread Main e carregar seu contexto em somente uma CPU. Escolhemos a CPU 0 para isso, e continuamos a usar a estratégia SPMD: caso a CPU atual seja a 0, associamos *first* à Thread contendo a main para ser executa.

Cada CPU deve ter uma Thread para executar. Neste caso, criamos uma Thread Idle para cada CPU. Caso a CPU seja a 0, porém, uma Thread Idle é criada e colocada no escalonador, caso contrário, para as demais CPUs, a Thread Idle é criada e associada à variável *first*. Após a finalização de Init_First, a CPU 0 terá a Main Thread carregada e as demais terão Threads Idles carregadas.

A ordem de inicialização das Threads aqui é importante. Quando uma Thread é criada, ela é imediatamente inserida no escalonador, que continua sendo um único, global para todas as CPUs. A primeira thread inserida no escalonador se torna o *_chosen*, a Thread que está atualmente rodando. Mas cada CPU agora possui uma Thread ativa. Desta forma, modificamos o escalonador para guardar uma referência *_chosen* associada a cada CPU. Mais sobre este ponto na seção “Multiple Scheduler”.

init_first.cc

```
 Init_First() {

      ...

      Thread * first;
      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::READY, Thread::IDLE),
&Thread::idle);
      }

      db<Init>(INF) << "Idle function: " <<
static_cast<Log_Addr>(&Thread::idle) << endl;

      db<Init>(INF) << "done!" << endl;

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

      db<Init, Thread>(INF) << "Dispatching the first thread: " <<
first << endl;

      This_Thread::not_booting();

      Machine::smp_barrier();

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

Ao final do código, antes de cada CPU carregar seu contexto, utilizamos uma barreira para garantir que o processo de inicialização terminou e que todas as CPUs carreguem seus novos contextos mais ou menos ao mesmo tempo.

Execução

Multiple Scheduler

Na implementação do enunciado do EPOS, a função *Thread::running()* é somente um forward para *scheduler.chosen()*. Com uma arquitetura monocore, isto é o suficiente, já que temos somente uma Thread rodando por vez. Porém, com multicore, múltiplas Threads estão em execução ao mesmo tempo e devemos ser capazes de dizer qual Thread está rodando em cada CPU.

Para sermos capazes de fazer isso, criamos novas classes para o escalonamento multicore. Essa nova classe Multiple_Scheduler opera da mesma forma que o Scheduler comum, porém *_chosen* é um array do tamanho do número de CPUs no sistema, que mantém uma referência para cada Thread rodando em cada CPU.

Ou seja, a fila *ready* do escalonador foi mantida inalterada, porém ao usar um *choose*, *insert*, *remove* etc, especificamos o ID da CPU que realizou a operação, assim o escalonador mantém as referências corretas de cada CPU associada a cada Thread.

scheduler.h

```
// Multiple Scheduling_Queue
template<typename T, unsigned int Size = Traits<Build>::CPUS, typename
R = typename T::Criterion>
class Multiple_Scheduling_Queue: public Multiple_Scheduling_List<T,
Size> {};

// Multiple Scheduler
template<typename T, unsigned int Size = Traits<Build>::CPUS>
class Multiple_Scheduler: public Multiple_Scheduling_Queue<T, Size>
{
private:
    typedef Multiple_Scheduling_Queue<T, Size> Base;

public:
    typedef typename T::Criterion Criterion;
    typedef Multiple_Scheduling_List<T, Size, Criterion> Queue;
    typedef typename Queue::Element Element;

public:
    Multiple_Scheduler() {}

    unsigned int schedulables() { return Base::size(); }

    T * volatile chosen(unsigned int consumer_id = Machine::cpu_id()) {
        return const_cast<T *
volatile>(Base::chosen(consumer_id)->object());
    }

    void insert(T * obj, unsigned int consumer_id = Machine::cpu_id())
{
        Base::insert(obj->link(), consumer_id);
    }

    T * remove(T * obj, unsigned int consumer_id = Machine::cpu_id()) {
        return Base::remove(obj->link(), consumer_id) ? obj : 0;
    }
```

```
    void suspend(T * obj, unsigned int consumer_id = Machine::cpu_id())
{
        Base::remove(obj->link(), consumer_id);
    }

    void resume(T * obj, unsigned int consumer_id = Machine::cpu_id())
{
        Base::insert(obj->link(), consumer_id);
    }

    T * choose(unsigned int consumer_id = Machine::cpu_id()) {
        T * obj = Base::choose(consumer_id)->object();
        db<Multiple_Scheduler>(TRC) << obj << endl;

        return obj;
    }

    T * choose_another(unsigned int consumer_id = Machine::cpu_id()) {
        T * obj = Base::choose_another(consumer_id)->object();

        db<Multiple_Scheduler>(TRC) << obj << endl;

        return obj;
    }

    T * choose(T * obj, unsigned int consumer_id = Machine::cpu_id()) {
        if(!Base::choose(obj->link(), consumer_id))
            obj = 0;

        db<Multiple_Scheduler>(TRC) << obj << endl;

        return obj;
    }
```

O funcionamento geral da nova classe *Multiple_Scheduler* foi mantido. As mudanças significativas se encontram no *Multiple_Scheduling_List*, descrito abaixo. As modificações aqui foram a inclusão do parâmetro *consumer_id* em todo método, tendo como valor default o ID da CPU atual. Desta forma, não precisamos modificar a classe Thread, já que, por padrão, enviamos o Machine::cpu_id(). Este valor é *forwarded* para a *Multiple_Scheduling_List* utilizada como base para este escalonador.

Outro ponto importante é a inclusão do parâmetro template Size, que tem como padrão o valor de número de CPUs do sistema (Traits<Build>::CPUS). Optamos por enviar este valor via template, já que o conhecemos em tempo de compilação e assim podemos definir o array de *_chosen (_chosen[Size])* de *Multiple_Scheduling_List* em seu próprio stack (e não via *new* ou utilizando uma lista, como seria feito caso contrário).

list.h

```
// Doubly-Linked, Scheduling List for multiple parallel scheduler
consumers
template<typename T,
          unsigned int Size,
          typename R = typename T::Criterion,
          typename El = List_Elements::Doubly_Linked_Scheduling<T, R> >
class Multiple_Scheduling_List: private Ordered_List<T, R, El>
{
private:
    typedef Ordered_List<T, R, El> Base;

public:
    typedef T Object_Type;
    typedef R Rank_Type;
    typedef El Element;
    typedef typename Base::Iterator Iterator;

public:
    Multiple_Scheduling_List() {
        assert(Size > 0);

        for(unsigned int i = 0; i < Size; i++)
            _chosen[i] = 0;
    }
    ...

    Element * volatile & chosen(unsigned int consumer_id) { return
_chosen[consumer_id]; }

    void insert(Element * e, unsigned int consumer_id) {
        if(_chosen[consumer_id])
            Base::insert(e);
        else
            _chosen[consumer_id] = e;
    }

    Element * remove(Element * e, unsigned int consumer_id) {
        if(e == _chosen[consumer_id])
            _chosen[consumer_id] = Base::remove_head();
        else
            e = Base::remove(e);

        return e;
    }

    Element * choose(unsigned int consumer_id) {
        if(!empty()) {
```

```
            Base::insert(_chosen[consumer_id]);
            _chosen[consumer_id] = Base::remove_head();
        }

        return _chosen[consumer_id];
    }

    Element * choose_another(unsigned int consumer_id) {
        if(!empty() && head()->rank() != R::IDLE) {
            Element * tmp = _chosen[consumer_id];
            _chosen[consumer_id] = Base::remove_head();
            Base::insert(tmp);
        }

        return _chosen[consumer_id];
    }

    Element * choose(Element * e, unsigned int consumer_id) {
        if(e != _chosen[consumer_id]) {
            Base::insert(_chosen[consumer_id]);
            _chosen[consumer_id] = Base::remove(e);
        }

        return _chosen[consumer_id];
    }

private:
    Element * volatile _chosen[Size];
};
```

A nova *Multiple_Scheduling_List* é semelhante à *Scheduling_List*, mas define um array de _chosen ao invés de um único elemento. Também adicionamos em cada método o id do consumidor atual (em nosso caso o id da CPU que requisitou o método). Assim, em cada ação podemos manipular o elemento associado ao ID do objeto consumidor atual. A fila *ready* continua igual e única para todas os consumidores.

No contexto do escalonamento das Threads, resolvemos nossos problemas com esta abordagem. Por padrão, uma requisição à Thread::running() retornará o objeto rodando na CPU atual. A fila *ready* continua global e única para todas as CPUs. Quando uma CPU desejar escalonar uma nova Thread para rodar, a invocação de *choose()* irá liberar a Thread desta CPU, recolocá-la na fila *ready* do escalonador (deixando-a disponível para outras CPUs) e retirar uma Thread da fila *ready*, associando ao *_chosen* da CPU atual e evitando que outra CPU a escalone.

O problema da **ordem de inicialização** também é resolvido com esta abordagem, já que a primeira Thread criada será a primeira escolhida pelo escalonador, mas no contexto de *cada CPU*. Assim, a única ordem que interessa é a ordem de inicialização de Threads para uma mesma CPU. Isso só acontece na CPU 0, que já possui a ordem correta (primeiro Main, depos Idle).

Nova função *idle*

A função Idle foi modificada para garantir que o sistema só seja encerrado quando de fato todas as threads não-idles tiverem sido encerradas.

Para isso, optamos por designar somente a CPU 0 para de fato encerrar o sistema. Enquanto houver mais threads do que CPUs, sabemos que há pelo menos uma Thread não idle no sistema, assim, precisamos continuar com o sistema operacional ligado, esperando que a thread seja reescalonada para terminar sua execução.

Caso haja um número igual ou menor de threads do que CPUs podemos encerrar o sistema. Caso a CPU 0 atinja este ponto, quer dizer que Main Thread terminou sua execução, então designamos a ela o trabalho de encerrar o sistema operacional.

```
int Thread::idle()
{
    while(true) {
       db<Thread>(TRC) << "Thread::idle(CPU=" << Machine::cpu_id()
           << ",n_cpus=" << Machine::n_cpus()
           << ",running=" << running() << ")" << endl;

       if (_thread_count > Machine::n_cpus() // We still have non-idles
to execute
           || Machine::cpu_id() != 0)        // Just CPU 0 can finish
OS
       {
           CPU::int_enable();
           CPU::halt();
       } else {
           // If we're here, we are CPU 0 and there are only Idle
Threads on System...
           db<Thread>(WRN) << "The main 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;
}
```

Atomicidade da Heap

O acesso simultâneo dos métodos *alloc* e *free* da Heap foi bloqueado utilizando um Spin lock e desligamento de interrupções:

heap.h

```
class Heap: private Grouping_List<char> {
...
private:
    Spin _lock;

    void take() {
        _lock.acquire();
        CPU::int_disable();
    }

    void release() {
        _lock.release();
        CPU::int_enable();
    }

public:
     void * alloc(unsigned int bytes) {
        take();
        …
        release();
        return addr;
    }

    void free(void * ptr, unsigned int bytes) {
        take();
        ...
        release();
    }
```