UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO - P3

Parallel Philosopher's Dinner

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Locking dos Componentes do SO

Locking correto da Heap

No método *enter* da Heap, primeiro adquiria-se o lock e depois desabilitava-se as interrupções. Entretanto, isto pode levar a um *deadlock* na seguinte situação: a CPU adquire o lock e imediatamente após isso, antes das interrupções serem desligadas, uma interrupção acontece. Digamos que o *handler* desta interrupção também chame um *alloc* ou *free* da Heap. Neste caso, a CPU irá requisitar o spin lock e ficará bloqueada por si mesma, ocasionando um *deadlock*. Por este motivo, alteramos o método enter para primeiro desabilitar as interrupções e só então adquirir o spin lock.

Também adicionamos uma condição para verificar se método enter() foi chamado já com as interrupções desabilitadas. Neste caso, quem chamou a função já está gerenciando um bloco crítico e neste caso não reabilitamos a interrupção no método leave().

Isto é particularmente interessante de ser feito na Heap, já que o desenvolvedor pode esquecer que a cada invocação de *new* um *int_enable()* implícito será invocado.

heap.h

```
private:
    bool _lock_enable_interruption;

    void enter() {
        // Save whether interruptions are already disabled or not
        // If interruptions are already disabled, we do not enable it
on leave() since
        // the caller is already managing interruptions
        _lock_enable_interruption = CPU::int_enabled();

        CPU::int_disable();
        _lock.acquire();
    }

    void leave() {
        _lock.release();

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

Locking da Thread

Para garantir o lock correto dos métodos de Thread, a mesma estratégia do locking da heap foi adotado: o uso de um spin lock.

Desta forma, modificamos os métodos *lock*() e *unlock*() de Thread, que antes só desabilitavam e reabilitavam as interrupções, para também usar um *spin lock*, bloqueando através de um busy waiting CPUs que por ventura disputem a seção crítica demarcada por esses métodos.

thread.h

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

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

Ao utilizar estas instruções, porém, um problema acontece: *deadlocks* ocorrem sempre que um **Thread::dispatch()** é chamado.

Isto acontece pois o método dispatch é chamado dentro de um bloco *lock()* e a função CPU::switch_context é invocada, esta que de fato muda o contexto de execução de uma CPU de uma Thread para outra. Mas quando o contexto de execução for alterado, o spin lock ainda não foi liberado (a Thread anterior ainda o possui). Como o spin lock usa o id da Thread (através de This_Thread::id()) quando a nova Thread tentar adquirir o spin lock, o owner ainda será a Thread anterior, que adquiriu o spin lock e nunca o liberou. Desta forma, um *deadlock* acontece e a aplicação trava.

Realizamos as seguintes modificações para tratar esta situação:

thread.cc

```
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;

        // ... debugs

        // Release the spin lock here before switching context, otherwise
       // a deadlock may occur (owner values are different for different
threads, see This_Thread::id())
       // We do not need to enable interruptions here since the 'iret' -
interruption return - assembly
       // instruction (inside CPU::switch_context) already does the job by
changing FLAGS (IF - interrupt flag)
       _lock.release();

        CPU::switch_context(&prev->_context, next->_context);
    } else
        // _lock.release() must be called just once! Otherwise _level
will be
        // incorrect (spin is released just when _level is 0)
        unlock();
}
```

Adicionamos um *_lock.release()* imediatamente antes de *switch_context*, liberando o spin lock da *previous* Thread. Não precisamos religar as interrupções aqui já que o *switch_context* faz este trabalho através da instrução assembly *iret*.

Também adicionamos o *unlock* do final do método na cláusula *else*, que é executada quando a *previous* Thread é igual a *next*. Fazemos isso para garantir que o *_lock.release()* é invocado apenas uma vez, caso contrário um valor errado será setado em _level do spin lock, ocasionando em comportamentos indesejados, inclusive deadlocks.

Execução paralela do Jantar dos Filósofos

Scheduler Multicore

A classe Scheduler foi alterada para utilizar a lista *Multihead_Scheduling_List* disponibilizada. Para tanto, o critério de escalonamento também precisou ser modificado para implementar alguns métodos e atributos requeridos por esta lista:

scheduler.h

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

namespace Scheduling_Criteria
{
    …

    // Multihead for Global Scheduling
    class Multihead
    {
    public:
        static const unsigned int HEADS = Traits<Build>::CPUS;
        static unsigned int current_head() {
            return Machine::cpu_id();
        }
    };

    // Round-Robin
    class RR: public Priority, public Multihead
    {
    public:
    ...
```

Philosophers_dinner.cc sem delay

Para a eliminação do delay em philosophers_dinner foi realizado um loop e dentro deste foi realizado uma operação aritmética para que o compilador não descarte este bloco de código.

```
….
int delay;

int philosopher(int n, int l, int c)
{
...
            for ( int a = 1 ; a <= 10000 ; a++ )
               for ( int b = 1 ; b <= 2000 ; b++ )
                  delay= b/a;

      chopstick[first]->p(); // get first chopstick
      chopstick[second]->p();   // get second chopstick

    ...

      for ( int a = 1 ; a <= 10000 ; a++ )
               for ( int b = 1 ; b <= 2000 ; b++ )
```

```
                delay= b/a;

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

Término correto da execução

Correção do fluxo de Barreiras da Inicialização

Ao rodar a aplicação com múltiplos cores e analisando o log de saída, notou-se que apenas a Idle Thread correspondente à CPU 0 rodava corretamente. As demais Idle Threads simplesmente não executavam a função Idle.

Após análise cuidadosa, notou-se que apenas a CPU 0 atingia a linha final do método Init_First, que é o comando *first->_context->load()*, responsável por carregar o contexto primeira thread da CPU. As demais CPUs travavam em algum momento anterior a esta e, apesar de serem reescalonadas e preemptadas, não executavam nada ao rodar.

O problema aqui é o mal uso de barreiras. Em especial, a última barreira do Init_First. Enquanto a CPU 0 passa por ela, as demais CPUs continuam bloqueadas ali.

Analisando o fluxo de inicialização do sistema, notou-se o mal uso de barreiras em Init_System: enquanto as CPUs diferentes de 0 passavam por 2 barreiras, a CPU 0 passava somente por 1. Como a barreira deve ser atingida por todas as CPUs para que todas sejam liberadas, as CPUs diferentes de 0 estavam sempre uma “barreira atrás” da CPU 0. Desta forma, a última barreira nunca será transposta pelas CPUs atrasadas.

O problema foi resolvido igualando o número de barreiras atingidas pelas CPUs em Init_System():

init_system.cc

```
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();
            Machine::smp_barrier();
            return;
        }

        // Initialize the processor [...]
        // Initialize System's heap [...]
        // Initialize the machine [...]

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

        // Initialize system abstractions [...]

        // Randomize the Random Numbers Generator's seed [...]

        Machine::smp_barrier(); // signalizes "end of Init_System" to
other CPUs
    }
```

Evitando Reescalonamentos Durante a Inicialização

Imagine que a CPU 0 tenha criado a Thread Main. O escalonador inseriu esta thread na cabeça da fila relacionada à CPU 0, fora da fila ready. Então a CPU 0 cria a Thread Idle e o escalonador insere-a na fila ready, que tem 1 membro por enquanto (a thread main está na cabeça da CPU 0, setada como *chosen*).

Então, a CPU 1 cria sua Thread Idle e insere-a no escalonador, que a aloca como *chosen* da CPU 1. Se neste momento, ainda durante a execução de Init_First(), o timer do escalonador for acionado para a CPU 1, o escalonador verá que a Idle da CPU 0 está na fila *ready* e irá trocar a Idle da CPU 1 pela Idle da CPU 0.

Acontece que, ao trocar uma Thread por outra, o *dispatch* salva o contexto de execução anterior pelo próximo, de forma que quando a Thread que está saindo de execução volte a ser escalonada, seu contexto possa ser restaurado. Mas em nosso exemplo, o contexto da Thread anterior é a própria inicialização do sistema: algum ponto do meio do Init_First() que estava em execução quando a interrupção do *timer* do escalonador foi acionado. Desta maneira, quando esta Thread for novamente escalonada, por qualquer CPU e em qualquer momento, um espaço de memória que possivelmente contém lixo será executado. Ou, na melhor das hipóteses, a inicialização voltará ao ponto que parou, mas em um momento totalmente inoportuno.

Para sanar este problema, utilizamos um artifício para evitar que Threads sejam reescalonadas durante o processo de inicialização:

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));

            new (SYSTEM) Thread(Thread::Configuration(Thread::READY,
Thread::IDLE), &Thread::idle);
        } else
            first = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING,
Thread::IDLE), &Thread::idle);

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

        This_Thread::not_booting();

        Machine::smp_barrier();

        // Signalize CPU as ready
        CPU::finc(Thread::_cpus_ready);

        db<Init, Thread>(INF) << CPU::int_enabled() << " first context load
[cpus_ready=" << Thread::_cpus_ready << "]" << endl;
        first->_context->load();
}
```

thread.cc

```
volatile unsigned int Thread::_cpus_ready = 0;

void Thread::time_slicer(const IC::Interrupt_Id & i)
{
    if (Thread::_cpus_ready != Machine::n_cpus()) {
        db<Scheduler<Thread>>(TRC) << "Thread::time_slicer() => Threads
still booting... stopping reschedule" << endl;
    } else {
        lock();
        db<Scheduler<Thread>>(TRC) << "Thread::time_slicer()" << endl;
        reschedule();
    }
}
```

Utilizamos a variável *Thread::_cpus_ready* para contar o número de CPUs que chegaram ao final do *Init_First*. No *handler* do *timer*, fazemos uma checagem e só reescalonamos se este número for igual do número de CPUs da máquina.

Porém, ainda pode ser que o *time_slicer* seja acionado imediatamente antes do *first->_context->load()* final do *Init_First*() e voltamos a ter o problema anterior.

Para evitar este último caso, adicionamos uma cláusula em *Thread::dispatch()*, que evita que uma Thread Idle seja trocada por outra, em qualquer momento do ciclo de vida do SO. Optamos por esta abordagem já que nossas Idles são idênticas e a troca de uma Thread Idle por outra nunca será benéfica.

thread.cc

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

    if (prev->priority() == IDLE && next->priority() == IDLE) {
        db<Thread>(TRC) << "Thread::dispatch IDLE BY IDLE(prev=" <<
prev << ",next=" << next << ")" << endl;
        next = _scheduler.choose(prev);
    }

   if(prev != next) {
       ...
}
```