UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO III

Idle Thread

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Idle Thread

A Idle Thread deve ser a Thread de mais baixa prioridade em todo o sistema. Para gerenciar este aspecto, adicionamos o valor IDLE ao enum de prioridade de threads. Este valor será usado apenas para a idle thread. Como a Queue utilizada no processo de escalonamento gerencia prioridade, sabemos que a Idle Thread sempre será inserida no final da fila *ready*, atrás de qualquer outra thread.

```
class Thread {
    ...
    // Thread Priority
    typedef unsigned int Priority;
    enum {
        HIGH = 0,
        NORMAL = 15,
        LOW = 31,
        IDLE = 63
    };
```

Para verificar que nenhuma outra Thread tenha sido criada com uma prioridade mais baixa do que a IDLE, utilizamos o seguinte assert no construtor de **Thread::Configuration**:

```
Configuration(const State & s = READY, const Priority & p = NORMAL,
unsigned int ss = STACK_SIZE)
        : state(s), priority(p), stack_size(ss)
{
    assert(p <= IDLE);
}
```

Criação da Idle Thread

O momento de criação da Thread Idle é um ponto crucial para este exercício. Caso a Idle Thread seja criada antes da Main Thread (Thread principal do programa de aplicação do usuário), pode ser que o alarme do time_slicer seja chamado e a idle thread comece a executar indefinidamente, antes da Main Thread ser sequer inicializada. Para evitar esse problema, fizemos duas modificações: criamos a Idle Thread logo após a criação da Main Thread e desativamos o alarme do escalonador logo após criá-lo, só voltando a ativá-lo após a criação da Main e da Idle Thread.

O método Thread::init(), passa a ser:

```
void Thread::init()
{
    ...
    if(preemptive) {
        _timer = new (kmalloc(sizeof(Scheduler_Timer)))
Scheduler_Timer(QUANTUM, time_slicer);
        _timer->disable();
    }
}
```

A modificação aqui foi desativar o timer logo após criá-lo, desta forma o escalonador não será acionado enquanto não voltarmos a reativar este timer.
A criação da Main e da Idle Thread acontece em **Init_First**:

```
Init_First() {

        db<Init>(TRC) << "Init_First()" << endl;

        if(!Traits<System>::multithread) {
            CPU::int_enable();
            return;
        }

        db<Init>(INF) << "Initializing the first thread: " << endl;
        Thread::_running = new (kmalloc(sizeof(Thread)))
Thread(Thread::Configuration(Thread::RUNNING, Thread::NORMAL),
reinterpret_cast<int (*)()>(__epos_app_entry));
        db<Init>(INF) << "done!" << endl;

        // 1:
        db<Init>(INF) << "Initializing the idle thread: " << endl;
        Thread::init_idle();
        db<Init>(INF) << "done!" << endl;

        This_Thread::not_booting();

        // 2:
        if (Thread::preemptive)
            Thread::_timer->enable();

        Thread::running()->_context->load();
    }
```

As adições neste método foram a chamada ao método **Thread::init_idle()**, responsável por criar a Idle Thread (1), e a reativação do timer de escalonamento da Thread (2). Desta forma, garantimos que o escalonador não será chamado durante a fase de inicialização de threads e que a inicialização da Main Thread precede a Idle. A ordem de precedência é

importante para o caso em que o escalonamento não leve em consideração prioridade, por exemplo.

```
void Thread::init_idle()
{
    db<Init, Thread>(TRC) << "Thread::init_idle()" << endl;
    new (kmalloc(sizeof(Thread))) Thread(Configuration(READY, IDLE),
idle);
}
```

É importante notar que a Idle Thread foi criada com estado inicial READY, o que a põe no final da fila *ready* do escalonador e com prioridade IDLE, a mais baixa prioridade que criamos especificamente para a Idle Thread.
O ponteiro para o objeto criado não é armazenado. A Idle Thread não pode ser manipulada por nenhum objeto, a não ser o escalonador, que irá colocá-la em execução quando nenhuma outra Thread estiver pronta para executar.
O bloco de código que será executado pela Thread é a função *idle()*, que foi modificada e será discutida mais adiante neste relatório. Essa função era deliberadamente chamada por outros métodos de manipulação de Thread quando a fila *ready* estivesse vazia. Não precisamos mais, porém, desta checagem a desta chamada de função literal, já que sempre haverá pelo menos uma Thread na fila *ready*: a Idle Thread.
Desta forma, todas as checagens de fila ready vazia e chamadas literal à função *idle* foram retiradas do sistema:

```
void Thread::yield()
{
      lock();

      db<Thread>(TRC) << "Thread::yield(running=" << _running << ")" <<
endl;

      //if(!_ready.empty()) {
      Thread * prev = _running;
      prev->_state = READY;
      _ready.insert(&prev->_link);

      _running = _ready.remove()->object();
      _running->_state = RUNNING;

      dispatch(prev, _running);
      //} else
      //idle();

      unlock();
}
```

```
void Thread::wait(Queue * waiting_queue)
{
      lock();

      db<Thread>(TRC) << "Thread::wait(running=" << _running << ")" <<
endl;

      //if(!_ready.empty()) {
      Thread * prev = _running;
      prev->_state = WAITING;
      waiting_queue->insert(&prev->_link);

      _running = _ready.remove()->object();
      _running->_state = RUNNING;

      dispatch(prev, _running);
      //} else
      //idle();

      unlock();
}
```

```
void Thread::suspend()
{
    ...
    if((_running == this)
        //&& !_ready.empty()
    ) {
        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(this, _running);
    } //else
        //idle(); // implicit unlock()

    unlock();
}
```

Execução da Idle Thread e Término da Última Thread

A função estática *idle()* passa a ser executada pela Idle Thread. Ela foi modificada da seguinte maneira:

```
int Thread::idle()
{
    while(_thread_count > 1) {
        db<Thread>(TRC) << "Thread::idle(thread_count=" <<
```

```
_thread_count << ")" << endl;

        db<Thread>(INF) << "There are no runnable threads at the
moment!" << endl;
        db<Thread>(INF) << "Halting the CPU ..." << endl;

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

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

    return 0;
}
```

A idle Thread começa executando um loop que colocará a CPU em modo *halt*. Antes habilitam-se as interrupções, para acordar a CPU no caso da ocorrência de alguma. Durante o estado halt, se alguma interrupção ocorrer e outra Thread for colocada na fila *ready*, o escalonador imediatamente iniciará a execução da nova Thread, já que ela terá uma prioridade maior à da Idle.
O loop da Idle Thread deverá perdurar enquanto Threads que estão ativas no sistema, mas esperando por algum evento, possam ser escalonadas. No entanto, se a Thread Idle for a única ativa, então o sistema pode ser completamente desligado, pois tem-se a certeza de que nenhuma outra Thread será criada. Para fazer esta checagem, precisa-se levar em conta toda a Thread possivelmente ativa, incluindo aquelas suspensas (na fila *_suspended*), esperando por um sincronizador (na fila de um semáforo, por exemplo) e aquelas na fila *_joining* de outra Thread. Como não temos acesso a cada uma dessas filas, mas precisamos de uma forma para contar o número de threads atualmente ativas no SO, criamos a variável *_thread_cont*.
A variável ***_thread_count***, é um contator do número de Threads ativas no sistema e é definida como `(private) static unsigned int _thread_count` na classe Thread e inicializada com `0`. Seu valor é incrementado ao construir uma Thread e decrementado ao destruir uma Thread. Desta forma, temos o número de Threads atualmente ativas no sistema. Se este valor for igual a 1, sabe-se que a Idle Thread é a única ativa e que o sistema pode ser desligado. Nesse caso, a Idle Thread sairá do loop do halt e o SO será encerrado.
O *_thread_count* é incrementado no *constructor_epilog* de cada Thread e decrementado no método exit() de cada Thread. Porém, caso uma Thread seja deletada antes de terminar, esta nunca cairá no exit() e o contador não será decrementado. Para resolver esse

problema, também adicionamos um decremento no destrutor da Thread, caso ela não esteja sendo encerrada:

```
Thread::~Thread()
{
    ...
     if (_state != FINISHING)
         _thread_count--;
    ...
}
```

O status FINISHING é setado dentro do método exit(), então sabemos que a Thread não terminou sua execução completamente caso, no momento de destrução, seu estado não seja FINISHING.

Novo método exit()

A nova implementação do método exit() não precisa mais checar se ainda existem Threads ativas no sistema. Essa verificação passa a ser feita na Idle Thread, conforme discutido acima.

```
void Thread::exit(int status)
{
    lock();

    db<Thread>(TRC) << "Thread::exit(status=" << status << ")
[running=" << running() << "]" << endl;

    *reinterpret_cast<int *>(_running->_stack) = status;
    _running->wakeup_joiners(); // implicit unlock();

    lock();

    //while(_ready.empty() && !_suspended.empty())
        //idle(); // implicit unlock();

    //lock();

    //if(!_ready.empty()) {
        Thread * prev = _running;
        prev->_state = FINISHING;
        _thread_count--;

        _running = _ready.remove()->object();
        _running->_state = RUNNING;

        dispatch(prev, _running);
```

```
    //} else {
        //db<Thread>(WRN) << "The last thread in the system has
exited!" << endl;
        //if(reboot) {
            //db<Thread>(WRN) << "Rebooting the machine ..." << endl;
            //Machine::reboot();
        //} else {
            //db<Thread>(WRN) << "Halting the CPU ..." << endl;
            //CPU::halt();
        //}
    //}

    unlock();
}
```