Relatório III de Sistemas Operacionais II - INE5424
Implementação para *Thread Idle*

Bruno Izaias Bonotto - 16104059

Fabíola Maria Kretzer - 16100725

João Vicente Souto - 16105151

1. Introdução

Alguns sistemas operacionais, como o Linux por exemplo, utilizam um processo especial (comumente chamado de *idle*) para lidar com os casos onde não exista nenhum processo apto a ser escolado. Nestes casos, visando a economia de energia, o processo *idle* é escalonado e sua função consiste em colocar a CPU em um estado de baixo consumo de energia. O sistema operacional mantém o baixo consumo enquanto não houver outro processo além do *idle* pronto para executar.

A versão didática do Sistema Operacional Paralelo Embarcado (*Embedded Parallel Operating System* - EPOS) é *single task* e *multi thread*, e *idle* é apenas uma função da *thread*. Dessa maneira, fica à cargo das *threads* a verificação constante da existência de outra *thread* a ser escalonada. Assim, durante a execução do método *idle*, caso não haja sequer uma *thread* que possa executar o sistema é finalizado e, caso contrário, a *thread* atual permanece presa em um loop enquanto não houver ninguém apto a executar (colocando a CPU em estado de baixo consumo e acordando-a com interrupções) ou chama o escalonador caso seja possível.

Com finalidade de estimular e aplicar os conhecimentos adquiridos ao longo das disciplinas, este trabalho demanda um estudo sobre a inicialização de um sistema operacional, funcionamento de *threads* e otimizações de códigos críticos executados constantemente. Ainda, busca uma implementação da *thread idle*, garantindo certos critérios de concorrência e, dessa maneira, manter a correta inicialização do sistema operacional.

2. Solução Adotada

Foi realizada a criação da *thread idle* na inicialização do sistema operacional, logo após a criação da *thread main*. Foi tomada essa decisão pois antes que o fluxo de execução chame __epos_app_entry e inicie o código da função *main()*, *idle* precisa estar na fila de *_ready*, garantindo que esta não esteja vazia, pelo menos enquanto a *thread idle* não estiver em execução. Com isso, é possível remover todas as verificações que checam a existência uma *thread* para executar, uma vez que *idle* sempre será escalonável.

Diferentemente da *main* que inicia já em execução, *Idle* inicializa no estado pronto (*READY*) e é inserida na fila *_ready* durante a execução de seu construtor. Além disso, foi criado uma nova prioridade para *idle* (VERYLOW), uma vez que ela só deve executar quando de fato não houver outra *thread* pronta. Assim sendo, *idle* não deve concorrer pelo uso de qualquer recurso, já que seu objetivo é simplesmente manter a CPU em baixo consumo enquanto não houver outra *thread* apta a ser escalonada.

A função delegada à *thread idle* permite monitorar a quantidade de *threads* no sistema, de modo que, sempre que houver uma *thread* que possa executar ela será escalonada. Caso nenhuma *thread* esteja pronta, *idle* colocará a CPU em *halt*. Assim, quando não houver nenhuma *thread* além da *idle*, esta se responsabiliza por chamar a função *reboot()* ou colocar a CPU em *halt()* por tempo indeterminado, resultando no desligamento do sistema.

Por fim, o controle da quantidade de *threads*, ocorre por meio da variável estática e volátil (*_threads_amount*). Sempre que um objeto do tipo thread é criado, o construtor incrementa o contador e, de forma semelhante, o decremento ocorre na função *exit()* ou ao deletar uma *thread* que ainda não esteja em estado *FINISHING*.

3. Implementação

a. init_first.cc

```
class Init_First {
```

```
    Init_First() {
        ...
        Thread * _idle = new (kmalloc(sizeof(Thread))) Thread(Thread::Configuration
                                    (Thread::READY, Thread::VERYLOW), &Thread::idle);
        ...
    }
};
```

b. thread.h

```
class Thread {
    …
    enum {
        …
        VERYLOW = 47
    }
    …
    private:
        static int idle();
    …
    private:
        static unsigned int volatile _threads_amount;
    …
};

inline Thread::Thread(...)
{
    …
    lock();
    _threads_amount++;
    unlock();
    …
}
```

c. thread.cc

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

inline Thread::~Thread(...) {
    …
    if (_state != FINISHING)
        _threads_amount--;
    …
}

void Thread::suspend() {
  …
  if(_running != this)
      _ready.remove(this);

   _state = SUSPENDED;
   _suspended.insert(&_link);

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

      dispatch(this, _running);
  }
  ...
}

void Thread::yield() {
  ...
  Thread * prev = _running;
  prev->_state = READY;
  _ready.insert(&prev->_link);

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

```
    dispatch(prev, _running);
    ...
}

void Thread::exit(int status) {
    ...
    _running = _ready.remove()->object();
    _running->_state = RUNNING;

    _threads_amount--;

    dispatch(prev, _running);
    ...
}

void Thread::sleep(Queue * q) {
    ...
    Thread * prev = running();
    prev->_state = WAITING;
    prev->_waiting = q;
    q->insert(&prev->_link);

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

    dispatch(prev, _running);
    ...
}

int Thread::idle() {
    lock();

    while (_threads_amount > 1) {
        if (_ready.empty()) {
            unlock();
```

```
            CPU::halt();
        } else {
            yield();
        }

        lock();
    }

    db<Thread>(WRN) << "The last thread in the system has exited!\n";

    if(reboot) {
        db<Thread>(WRN) << "Rebooting the machine ...\n";
        Machine::reboot();
    } else {
        db<Thread>(WRN) << "Halting the CPU ...\n";
        CPU::halt();
    }

    return 0;
}
```

4. Conclusão

Inicialmente, por não ter uma *thread* responsável pela verificação da existência de outras no sistema, diversas verificações eram necessárias para manter a consistência do sistema EPOS. Como uma alternativa à essa implementação, neste trabalho foi criado uma *thread* (*idle*) cuja responsabilidade é justamente monitorar o número de *threads* e, dessa maneira tomar determinadas medidas. Como a verificação do estado do sistema operacional ficou a cargo de uma única *thread*, as demais não necessitam fazer certas verificações, o que permite reduzir o escopo de vários métodos da classe.

Tomando os devidos cuidados com a prioridade da *thread idle* e o estado em que ela é disparada no sistema, foram executados os testes e aplicações disponíveis no sistema. Não havendo erros, assume-se que a implementação do

grupo, assim como a anteriormente implementada na versão do sistema operacional, estão de acordo com os objetivos propostos. No entanto, para melhor avaliar tais implementações, seriam necessários testes mais específicos e que levassem o sistema à seu limite, ou testar casos particulares que pudessem levar o sistema à uma possível situação de erro.