Relatório IV de Sistemas Operacionais II - INE5424
Implementação Alternativa para *Delay*

Bruno Izaias Bonotto - 16104059

Fabíola Maria Kretzer - 16100725

João Vicente Souto - 16105151

1. Introdução

Em diversas aplicações, alguns recursos são necessários para garantir a correta execução do programa. Um deles é utilizado para permitir um processo ou *thread* aguardar certo período de tempo sem nenhuma ação, comumente chamado de *delay*. Outro, geralmente chamado de alarme, permite a execução de uma função (denominada *handler*) ao fim do *delay*. Este por sua vez, é um recurso disponibilizado pelo sistema operacional e também muito utilizado por ele no tratamento de certas interrupções de *hardware*.

A versão didática do Sistema Operacional Paralelo Embarcado - *Embedded Parallel Operating System* (EPOS) conta com uma implementação simples dos recursos de *delay* e *alarme*. Toda função *handler* executa “atomicamente”, uma vez que o sistema operacional desabilita interrupções de *hardware* durante a execução. Porém, como a função *handler* pode ser fornecida por uma aplicação do usuário, caso a mesma tenha inconsistências (como um *loop* infinito) é possível que o sistema operacional acabe inoperável. Outro problema advindo de uma ingênua implementação desses recursos é a ocupação desnecessária do processador causada pelo *busy waiting* no método de *delay*.

Com objetivo de aplicar os conhecimentos adquiridos ao longo das disciplinas, este trabalho busca solucionar os problemas relacionados aos métodos *handler* e *delay* do EPOS. Assim, garantir um uso mais proveitoso do processador e a consistência do sistema operacional durante a execução de *handlers* fornecidos pela aplicação de usuários.

2. Interrupções de Tempo

As interrupções de tempo no EPOS (versão ia32) são emitidas por um oscilador, *Programmable Interval Timer (PIT)*, que possui três contadores de 16 bits, comumentemente chamados de *channels*. Cada *channel* possui uma frequência diferente e permite que o desenvolvedor do sistema operacional os configure conforme sua necessidade. Ao zerar um contador, o oscilador emite uma interrupção que é capturada pela CPU, onde seu identificador é encaminhado, através de código de máquina, a função *PC_IC::dispatch* do EPOS. Por fim, *PC_Timer::int_handler* é chamado através de um vetor de funções *handlers* e executa o *handler* de algum dos timers existentes.

O sistema operacional EPOS conta com *Scheduler_Timer, Alarm_Timer e User_Timer* para lidar com as interrupções do *channel* 0, *channel* 1 e *channel* 2, respectivamente. O foco deste trabalho está em estudar a implementação da função *handler* passado ao *Alarm_Timer* e resolver os problemas de reentrância e espera ocupada existentes.

3. Solução Utilizada

O primeiro problema solucionado neste trabalho foi em relação ao método *handler*. Para garantir que o sistema operacional não permanecesse executando a função do usuário sem critério de parada, esta implementação permite que o *handler* do usuário seja executado com as interrupções de *hardware* ativas. Assim, o sistema operacional pode ser interrompido durante a execução de um *handler* e realizar outras atividades.

a. Detalhes de implementação (alarm.cc):

```
...
void Alarm::handler(const IC::Interrupt_Id & i)
{
    /* Variável utilizada para armazenar
    o próximo tick a ser decrementado */
```

```
static Tick next_tick;

/* Variável utilizada para armazenar o
ponteiro do próximo handler a ser executado */
static Handler * next_handler;

/* Garante que o código executado a seguir não
seja interrompido (desabilita interrupções) */
lock();

/* Variável utilizada para contar o tempo global */
_elapsed++;

…

/* Verificação do valor de tick, só
decrementa se for maior que zero */
if(next_tick)
    next_tick--;

/* Prossegue apenas se o valor do tick é zero */
if(!next_tick) {

    /* Cria-se um handler temporário para permitir sua
    utilização posterior mesmo alterando o next_handler */
    Handler * current_handler = next_handler;

    /* Se não existe nenhum alarme atualiza o valor do
    next_handler para zero (não existe handler neste caso) */
    if(_request.empty())
        next_handler = 0;
    else {

        /* Caso exista um próximo alarme, pega-se
        ele na fila de alarmes (ordenada por tick) */
        Queue::Element * e = _request.remove();
        Alarm * alarm = e->object();

        /* Atualiza-se os valores do próximo tick e do
        ponteiro da próxima função handler a ser executada */
        next_tick = alarm->_ticks;
        next_handler = alarm->_handler;
```

```
                /* Decrementa-se o número de vezes em que o alarme já foi
                acionado apenas quando ainda é maior ou igual a zero */
                if(alarm->_times != -1)
                    alarm->_times--;

                /* Se o número de vezes que o alarme foi acionado for maior que zero então
                  atualiza o rank (garante ordenamento) e insere o alarme na lista novamente
*/
                if(alarm->_times) {
                    e->rank(alarm->_ticks);
                    _request.insert(e);
                }
            }

            /* Habilita as interrupções novamente */
            unlock();

            /* Se houver um handler válido então executa-o */
            if(current_handler)
            {

                …

                (*current_handler)();
            }
        } else

            /* Habilita as interrupções novamente */
            unlock();
    }
    …
```

O segundo problema solucionado diz respeito ao método *delay*. A primeira abordagem consistiu em atribuir ao próprio alarme a responsabilidade de acordar a *thread* que solicitou o tempo de espera. Porém constatou-se que esta implementação causava *deadlock* na aplicação dos filósofos. Isso ocorreu pois o alarme poderia liberar a *thread* antes mesmo dela ser suspensa.

Uma segunda abordagem consistiu em passar ao alarme um semáforo fechado, garantindo que mesmo que o tempo dele terminasse antes da *thread* alocar o semáforo, ela ainda poderia continuar. Porém, existe ainda a necessidade de conhecer o alarme atual, o que não era possível devido ao fato dos *handlers* serem funções estáticas e sem parâmetros. Todas as variações implementadas a partir dessa ideia também envolviam atribuir ao alarme alguma funcionalidade e, portanto, falharam.

Ao esgotar das ideias, sentiu-se a necessidade de conhecer uma solução já implementada (que foi obtida em: https://github.com/eloisamec/epos) e compreender sua abordagem. Como foi necessária ajuda apenas neste ponto, a solução da função *handler* foi mantida. A implementação alternativa encontrada utiliza uma interface de *functor* que, ao implementar *overloading* sobre o *operator()* permite aos objetos de determinadas classes serem tratados como funções. A principal vantagem no uso de um *functor* é o encapsulamento de um escopo específico, tais como variáveis locais, algo impossível para implementações de funções comuns sem o uso de variáveis globais ou estáticas. O uso desse recurso auxiliou na resolução do problema encontrado na segunda abordagem, permitindo que o *functor Semaphore_Handler* liberasse o semáforo em outro momento sem torná-lo estático ou escondido dentro do alarme. Porém, a desvantagem da interface *Handler* (*functor*) consiste no uso de herança e métodos virtuais para permitir a criação de *handlers* genéricos, introduzindo um *overhead* em toda chamada do método *operator(),* sendo necessária a buscar na *vtable* da função correta a ser executada.

b. Detalhes de implementação (handler.h):

```
/* Interface Functor para abstração dos handlers */
class Handler {
public:
        /* Tipo de função utilizada por handlers */
        typedef void (Function)();

        Handler() {}
```

```
        virtual ~Handler() {}

        /* Função virtual para obrigar quem herda desta classe em implementá-la */
        virtual void operator()() = 0;
};
```

c. Detalhes de implementação (Semaphore.h):

```
/* Handler que encapsula um semáforo */
class Semaphore_Handler : public Handler
{
public:
  Semaphore_Handler(Semaphore * h) : _handler(h) {};
  ~Semaphore_Handler() {};

  /* Libera o semáforo ao ser executado na função Alarm::handler */
  void operator()() { _handler->v(); }

private:
  Semaphore * _handler;
};
```

c. Detalhes de implementação (alarm.cc):

```
void Alarm::delay(const Microsecond & time)
{
        ...

        /* Semáforo inicializado fechado dentro do contexto da função delay() */
        Semaphore semaphore(0);

        /* Handler encapsula o semáforo */
        Semaphore_Handler handler(&semaphore);

        /* Cria-se um alarme com o tempo de espera e o handler para ser
        executado apenas uma vez */
        Alarm alarm(time, &handler, 1);

        /* Thread espera o alarme executar o handler e liberar o semáforo */
        semaphore.p();
}
```

4. Conclusão

Inicialmente, a implementação da função *handler* existente no EPOS não era reentrante, permitindo que o sistema operacional fosse comprometido (o usuário poderia fornecer uma função inconsistente, tal como um *loop* infinito). A abordagem utilizada neste trabalho, como alternativa à originalmente utilizada, resolveu o problema de reentrância ao habilitar as interrupções antes de executar qualquer *handler*, tomando cuidado em manter uma variável local para não corromper as variáveis estáticas utilizadas na execução da função *handler()*.

Em seguida, procurando solucionar o problema de espera ocupada na função *delay()* e enfrentar dificuldades relacionadas à linguagem C++, buscou-se uma solução alternativa como guia de ideias. Tal implementação utiliza o conceito de *functors* para criar uma abstração maior para os *handlers* do sistema operacional. Assim, com a utilização desse recurso foi possível implementar umas das ideias propostas anteriormente.

Apesar das vantagens apresentadas, a solução deste trabalho também não é perfeita. A herança utilizada nos *Functors* introduz um custo adicional à toda chamada de função e, portanto, torna o sistema operacional mais lento. Ainda assim, buscou-se fazer uso de diversos conhecimentos adquiridos ao longo do curso.