UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO IV

Timing

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Modificação do Alarm::handler() e reentrância

O método Alarm::handler() é chamado a cada interrupção de relógio causada no sistema. Quando o contador de Timer chega a um determinado número de pulsos de clock, uma interrupção é lançada e o tratador desta interrupção é o método Alarm::handler(). Quando esta interrupção ocorre, o fluxo de código é deslocado da thread que está atualmente em execução para o tratador de interrupção em questão. Este método, portanto, é responsável por tratar e executar possíveis alarmes esperando para serem executados.

Este *handler* porém, possui algumas limitações decorrentes de sua implementação. A primeira é que ele não é reentrante, ou seja, uma interrupção deve ser totalmente tratada para outra ser lançada. Isto, graças ao escopo *lock* / *unlock* do método. Outra questão limitante que pode levar a comportamentos inesperados é se o usuário setar alarmes cujas funções sejam loops infinitos ou que demorem muito para terminar, neste caso, o sistema iria esperar que a função terminasse, não importe o tempo que ela demore para encerrar.

Resolvemos estes problemas removendo o escopo *lock* / *unlock* do método e modificando suas lógica de implementação para livrar-nos das variáveis estáticas *next_tick* e *next_handler*, permitindo a reentrância desta interrupção.

```
void Alarm::handler(const IC::Interrupt_Id & i)
{
    //static Tick next_tick;
    //static Handler * next_handler;

    //lock();

    _elapsed++;

    if(Traits<Alarm>::visible) {
        Display display;
        int lin, col;
        display.position(&lin, &col);
        display.position(0, 79);
        display.putc(_elapsed);
        display.position(lin, col);
    }

    if (!_request.empty()) {
        _request.head()->promote(); // 1
        // 2:
        while (_request.head()->rank() <= 0) {
            Queue::Element * e = _request.remove();
```

```
            Alarm * alarm = e->object();
            // 3:
            if (alarm->_times != -1)
                alarm->_times--;
            // 4:
            if (alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }

            db<Alarm>(TRC) << "Alarm::handler(h=" <<
reinterpret_cast<void *>(alarm->_handler) << ")" << endl;
            (*alarm->_handler)(); // 5
        }
    }

    //unlock();
}
```

(1) - **Promote** decrementa o rank da cabeça da lista *_request*. Esta é uma fila ordenada e relativa e o rank de seus elementos (que são alarmes) são na verdade o número de *ticks* até que o alarme possa executar. Esta operação, portanto, decrementa o tick de cada alarme.

(2) Se a cabeça da lista possuir um rank igual (ou menor) a 0, então está na hora de disparar este alarme. Utilizamos um *while* aqui para, no caso de dois alarmes na fila com o mesmo rank dispararem ao mesmo tempo, podemos executar ambos nesta mesma interrupção. Caso contrário (se tivéssemos utilizado um *if*, por exemplo) os alarmes subsequentes dispariam somente no próxima interrupção de alarme, quando seus *ranks* estivessem negativos.

(3) Diminuímos o número de vezes que o alarme ainda deve executar

(4) Caso o alarme ainda deva executar, reinserimos o elemento na fila, com seu rank igual ao número original de *ticks* até que possa ser executado.

(5) Por fim, executamos o *handler* associado ao alarme.

Remoção do Busy Waiting em delay()

A implementação atual de Delay() do EPOS utiliza um busy waiting para pausar a Thread em execução até que o tempo de delay estipulado transcorra.

Para retirar este busy waiting, optamos por colocar a Thread em um modo de espera e utilizar um alarme que será disparado após o tempo de delay estipulado e que irá acordar

esta thread. Para utilizarmos tal técnica, no entanto, devemos ser capazes de estipular parâmetros ao método *handler* executado pelo alarme.

Na implementação atual do EPOS, porém, o *handler* é uma *void function* sem parâmetros. Decidimos por alterar o tipo deste método para uma classe abstrata Handler, com o método virtual puro *operator()()*. Desta forma, através de polimorfismo, podemos associar diferentes Handlers a alarmes.

```
class Handler {
public:
    Handler() {}
    virtual ~Handler() {}

    virtual void operator()() = 0;

protected:
    typedef void (Function)();
};

class Function_Handler : public Handler {
public:
    Function_Handler(Function * f) : _f(f) {}

    void operator()() {
        _f();
    }

protected:
    Function * _f;
};
```

Assim, conseguimos criar um Handler especialmente projetado para acordar uma Thread que está esperando por um delay, utilizando um semáforo e um alarme, desta maneira:

```
void Alarm::delay(const Microsecond & time)
{
    db<Alarm>(TRC) << "Alarm::delay(time=" << time << ")" << endl;

    //Tick t = _elapsed + ticks(time);

    //while(_elapsed < t);

    Semaphore s(0);
    Delay_Handler handler(&s);
    Alarm alarm(time, &handler);
    s.p();
}
```

Ao entrar em *delay*, um semáforo será criado e inicializado com valor 0. Um alarme será configurado para disparar após o tempo de delay estipulado. O tratador (*handler*) deste alarme será um Handler especial que irá liberar este semáforo (*Delay_Handler*). No final do método, a Thread atual entrará em modo WAIT ao chamar *p()* do semáforo que foi inicializado com 0 e só voltará a executar quando for liberada pelo *v()* do *Delay_Handler*():

```
class Delay_Handler : public Handler
{
public:
    Delay_Handler(Semaphore * s) : _s(s) {}

    void operator()() {
        _s->v();
    }

protected:
    Semaphore * _s;
};
```