Universidade Federal de Santa Catarina

Centro Tecnológico

Departamento de Informática e Estatística

INE5424 - Sistemas Operacionais II

Professor: Antônio Augusto Medeiros Fröhlich

Estagiário de Docência: Mateus Krepsky Ludwich

Grupo: 01

Alunos: Alisson Granemann Abreu matrícula: 11100854

Maria Eloísa Costa matrícula: 10200630

Osvaldo Edmundo Schwerz da Rocha matrícula: 10103132

E4: Timing

To do

You are requested to modify the implementation of Alarm to eliminate the busy-waiting in method delay() and also to remodel the interrupt handling routine so that user's handler functions no longer have a chance to disrupt the system timing.

Informações úteis do enunciado

O enunciado do exercício deixa clara a existência de um busy-waiting no método delay(), enfatizada no código em vermelho (código original do EPOS).

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

      Tick t = _elapsed + ticks(time);

      while(_elapsed < t);
}
```

Outro fator, indicado como o principal no enunciado, é das funções de usuário poderem perturbar o tempo do sistema. No caso, o tratador de exceções permite que funções de usuário sejam executadas dentro do seu próprio contexto.

Solução adotada

De acordo com discuções em sala e dicas dadas, consideramos 3 pontos de alterações necessárias para que o exercício fosse completado. Inicialmente decidimos estender a noção de *Handler* para todas as classes que derivavam o *Synchronizer_Common* para que pudessem tirar proveito da noção de temporização. Com isto, o tratamento do método *delay* torna-se mais fácil, nosso segundo ponto, fazendo com que a chamada do método de liberação de uma classe *synchronizer* seja feita assim que o tempo desejado tenha finalizado.

O terceiro ponto, e de maior complexidade, a nosso ver, estava no método handler do próprio *Alarm*. Este deveria ser reentrante, visto que é compartilhado por diversas threads do sistema ao mesmo tempo e isso implica em uma única cópia residindo em memória para atender a todos os programas/threads que estejam sendo executados.

Implementação

Método Alarm::delay

Para que pudéssemos possibilitar a escolha entre busy-waiting e idle-waiting, criamos uma variável global em *Traits*.

```
template<> struct Traits<Alarm>: public Traits<void>
{
    static const bool visible = hysterically_debugged;
       static const bool idle_waiting = true;
};
```

E alteramos o método *delay* em *Alarm* para que houvesse uso da funcionalidade idle-waiting. A idle-waiting funciona de forma que a temporização seja feita pela própria classe *Alarm*, utilizando um semáforo e a chamada do seu método *v* caso *time* seja menor que *tick*. O mesmo poderia ser feito com um mutex, por exemplo.

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

      if(idle_waiting) {
            Semaphore semaphore(0);
            Semaphore_Handler handler(&semaphore);
            Alarm alarm(time, &handler, 1);
            semaphore.p();
      } else {
            Tick t = _elapsed + ticks(time);
            while(_elapsed < t);
      }
}
```

Em alarm.h

```
protected:
static const bool idle_waiting = Traits<Alarm>::idle_waiting;
```

Método Alarm::handler

Toda vez que o método *Alarm::handler()* for invocado indica que um novo tick de relógio ocorreu no sistema e alguns pontos precisam ser vistos neste momento:

1 - Como a posição na fila é relativa, como trataremos quando tivermos mais de uma thread chegando a zero ao mesmo tempo?

2 - Existe alguma forma desse valor vir a ficar negativo?

3 - Não é interessante termos o mesmo *handler* para todos os *alarms.* Cada *alarm* terá seu próprio *handler* e cada um precisa ser chamado quando o *alarm* acordar.

4 - Como garantir reentrância na função?

A questão 3 foi feita com auxílio de uma alteração na aplicação *alarm_test.cc*.

```
Function_Handler handler_a(&func_a);
Alarm alarm_a(2000000, &handler_a, iterations);
Function_Handler handler_b(&func_b);
Alarm alarm_b(1000000, &handler_b, iterations);
```

Para pensarmos no ponto 4, tivemos a dica do professor quanto a código reentrante. No caso, para que isso ocorra, não podemos mais ter variáveis e/ou dados estáticos, visto que várias threads irão acessar esta mesma parte do código e fazer alterações nas variáveis. Esta alteração nos garante maior segurança na execução do código.

Removendo o envolvimento das variáveis estáticas do código, nos sobrou isto (acreditamos que tenha sido esta a dica do professor durante a aula):

```
void Alarm::handler(const IC::Interrupt_Id & i)
{
    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()){
        Queue::Element * e = _request.remove();
        Alarm * alarm = e->object();
        if(alarm->_times != -1)
            alarm->_times--;
        if(alarm->_times) {
            e->rank(alarm->_ticks);
            _request.insert(e);
        }
    }
    unlock();
}
```

Porém notamos que com essa alteração o sistema não funcionava mais. Analisando melhor o código antigo, vimos que reativando a parte de código abaixo, resolveríamos parte da questão 3 levantada: Se tiverem outros handlers para acordar, devemos chamá-los.

```
if(next_handler) {
      db<Alarm>(TRC) << "Alarm::handler(h=" << reinterpret_cast<void
*>(next_handler) << ")" << endl;
      (*next_handler)();
}
```

Porém não temos o próximo handler a chamar (variável next_handler removida por ser estática). Mas ainda temos como chamá-lo como era feito anteriormente;

```
next_handler = alarm->_handler;
```

Reativando o código no local original, encontramos mais um erro. O alarm trava após a chamada do delay.

```
void Alarm::handler(const IC::Interrupt_Id & i)
{
    ...

    Alarm * alarm = 0;
    if(alarm) {
            db<Alarm>(TRC)       <<       "Alarm::handler(h="       <<
reinterpret_cast<void *>(alarm->_handler) << ")" << endl;
            (*alarm->_handler)();
        }
        if(!_request.empty()){
   ...
}
```

Porém o next_handler só recebia algum dado após o else (código original). No caso, quando a fila _request já não estava mais vazia e existiam outros alarms vivos no sistema. Colocamos a verificação após toda a verificação do nosso então if (antigo else).

```
    ...
            if(alarm->_times) {
                e->rank(alarm->_ticks);
                _request.insert(e);
            }
            if(alarm) {
                db<Alarm>(TRC)  <<  "Alarm::handler(h="  <<
reinterpret_cast<void *>(alarm->_handler) << ")" << endl;
                (*alarm->_handler)();
            }
    }

    unlock();
}
```

E encontramos mais um erro. O programa passa a imprimir todas as informações deliberadamente na tela, com os prints totalmente misturados e finaliza o programa. Provavelmente por este if só estar liberando o fim do programa prematuramente sem nenhum tratamento dos *alarms* na fila _request. O que nos leva ao primeiro ponto levantado: Como a posição na fila é relativa, como trataremos quando tivermos mais de uma thread chegando a zero ao mesmo tempo? E o segundo ponto: Existe alguma forma desse valor vir a ficar negativo? Será que temos que fazer mais algum tratamento nesta fila?

Todos os *alarms* precisam tomar conhecimento do novo tick que ocorreu, diminuindo o rank do primeiro elemento da fila _request. Isso é feito através da chamada do método *promote*. Porém agora é necessário tratar os próximos *alarms* que podem vir a estar negativos (em vermelho):

```
...
if(!_request.empty()){
    _request.head()->promote();
```

```
            if(_request.head()->rank() <= 0) {
                  _request.insert(&(_request.remove()->object()->_link));
            }
            Queue::Element * e = _request.remove();
            alarm = e->object();
      ...
```

Acreditamos que ainda haja algum erro no funcionamento do programa, visto que a impressão ainda é feita quase que automaticamente após o início do programa, porém não conseguimos imaginar qual outra questão esquecemos de abordar neste caso, visto que é a primeira vez que trabalhamos com algo relacionado a temporização de um Sistema Operacional. O método *handle* ficou da seguinte forma:

```
      void Alarm::handler(const IC::Interrupt_Id & i)
      {
            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);
          }

            Alarm * alarm = 0;

            if(!_request.empty()){
                  _request.head()->promote();
                  if(_request.head()->rank() <= 0) {
                        _request.insert(&(_request.remove()->object()-
>_link));
```

```
                    }
                    Queue::Element * e = _request.remove();
                    alarm = e->object();
                    if(alarm->_times != -1)
                           alarm->_times--;
                    if(alarm->_times) {
                           e->rank(alarm->_ticks);
                           _request.insert(e);
                    }
                    if(alarm) {
                           db<Alarm>(TRC)     <<     "Alarm::handler(h="     <<
reinterpret_cast<void *>(alarm->_handler) << ")" << endl;
                           (*alarm->_handler)();
                    }
             }

             unlock();

      }
```

Classe *Handler*

No modelo atual do EPOS, o *Handler* era apenas um *define*. Para tanto, fizemos com que a *Handler* passe a ser uma interface com um único método. A partir dela, criamos uma classe para as funções de usuário, semáforo, mutex, condition e thread nos respectivos headers, sendo em *handler.h* que tratamos as funções de usuário e onde resolvemos o problema do delay.

```
class Thread_Handler : public Handler {
public:
    Thread_Handler(Thread * h) : _handler(h) {};
    ~Thread_Handler() {};

    void operator()() { _handler->resume(); }

private:
    Thread * _handler;
};
```

```
class Semaphore_Handler : public Handler
{
public:
    Semaphore_Handler(Semaphore * h) : _handler(h) {};
    ~Semaphore_Handler() {};

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

private:
    Semaphore * _handler;
};

class Mutex_Handler: public Handler
{
public:
    Mutex_Handler(Mutex * h) : _handler(h) {};
    ~Mutex_Handler() {};

    void operator()() { _handler->unlock(); }

private:
    Mutex * _handler;
};

class Condition_Handler: public Handler
{
public:
    Condition_Handler(Condition * h) : _handler(h) {};
    ~Condition_Handler() {};

    void operator()() { _handler->signal(); }

private:
    Condition * _handler;
};
```

```
class Handler {

public:
	typedef void (Function)();

	Handler() {}
    virtual ~Handler() {}

	virtual void operator()() = 0;
};

class Function_Handler : public Handler {

public:
	Function_Handler(Function * handler) : _handler(handler) {};
	~Function_Handler() {};

	void operator()() {
		_handler();
	}
private:
	Function* _handler;
};
```