Objetos ativos

Introdução

O Symbian OS é um sistema operacional multitarefa preemptivo. Isso quer dizer que o sistema permite o uso de várias *threads* por aplicação e a execução das *threads* pode ser interrompida a qualquer momento pelo escalonador do sistema, para dar vez a uma outra *thread*. Entretanto, escalonar uma outra *thread* possui um custo. Esse procedimento, conhecido como troca de contexto, envolve chamadas ao *kernel* do sistema e, possivelmente, gestão de memória (por exemplo, cópia dos dados do contexto de uma *thread* para ela retomar a execução). Em dispositivos como telefones celulares, que possuem recursos de memória e CPU mais restritos, esse custo pode ser grande.

Dessa forma, o Symbian OS possui uma outra alternativa para implementar aplicações multitarefa, de maneira cooperativa. Nesse modo, pode-se simular o comportamento de uma aplicação *multi-thread* utilizando-se apenas uma *thread*. Essa alternativa é representada pelos objetos ativos.

Os objetos ativos são usados para simplificar a programação de tarefas que devem ser executadas de maneira assíncrona. Eles provêm maneiras de se fazer requisições assíncronas, detectar o término de alguma tarefa e processar os resultados. São indicados para programação orientada a eventos.

A classe base para objetos ativos é `CActive`. É necessário criar uma classe derivada de `CActive`, definir o método que representará a chamada assíncrona, e implementar alguns métodos da classe base que são requeridos para o funcionamento dos objetos ativos.

O escalonador ativo (*active scheduler*) é um componente do sistema responsável por gerir a execução dos objetos ativos. Cada *thread* possui uma instância desse componente. Basicamente, essa entidade possui um laço de execução que verifica se algum evento de algum objeto ativo termina. Nesse caso, o escalonador invoca um método do objeto ativo para avisá-lo desse evento. Esse método corresponde ao `RunL()` de `CActive`, que as classes derivadas devem implementar (é requerido). O escalonador volta ao modo de espera depois que objeto ativo executa a resposta ao evento.

É importante observar que o modo de multitarefa executado pelos objetos ativos é cooperativo. Em outras palavras, o escalonador não pode interromper um objeto ativo para executar um outro objeto ativo, ou seja, uma vez que um método `RunL()` é executado, outros eventos devem esperar que o primeiro se encerre, para que possam ser executados. Isso pode ser um problema no caso de um objeto ativo executar indefinidamente, o que deve ser evitado. O sistema pode interromper a *thread* que executa os objetos ativos para dar vez a uma outra, o que corresponde à gestão de processos global do sistema. Mas isso não interfere na execução dos objetos ativos, ou seja, o objeto ativo que estava sendo executado antes da interrupção continuará o seu processamento.

Roteiro para a criação de objetos ativos

Para se criar um objeto ativo, é preciso criar uma classe derivada de `CActive` e implementar os métodos obrigatórios, que são `RunL()` e `DoCancel()`. O trecho de código seguinte define uma classe de objeto ativo.

```
class CMyActive : public CActive
{
 public:
   static CMyActive* NewL();

   CMyActive();
  ~CMyActive();

   // Representa a chamada assíncrona, ou pedido para
   // iniciar algum serviço, por exemplo
   void InvokeService();

 public:
   // declaradas em CActive

   // É executada quando a chamada assíncrona se completa (obrigatória)
```


(C) Copyright Nokia 2014. All rights reserved.

```
    void RunL();

    // Define o que fazer para cancelar uma chamada em andamento (obrigatória)
    void DoCancel();

    // Chamada para se tratar leaves que venha a ocorrer em RunL (opcional)
    TInt RunError (TInt err);

  private:
    void ConstructL();

    TRequestStatus iStatus;
}
```

Segue aqui o roteiro de criação para objetos ativos:

1. Definir funções para se criar o objeto ativo (`ConstructL`, `NewL`, construtor);

2. Registrar o objeto ativo com o escalonador ativo;

3. Definir e implementar as funções que representam as chamadas assíncronas (uma ou várias);

4. Redefinir o `RunL()`, que será executado ao término do processamento da chamada assíncrona. O nome talvez não seja dos melhores, já que esse método corresponde à resposta ao evento de terminar o processamento do serviço;

5. Redefinir o `DoCancel()`, para poder cancelar uma chamada assíncrona que ainda está em andamento

6. Definir o destrutor, que deverá chamar o método `Cancel()` da classe `CActive`, para que todas as chamadas possam ser canceladas caso o objeto seja destruído;

7. (Opcional) Tratar exceções (*leaves*) redefinindo o método `RunError()`.

Construção

Assim como *threads*, os objetos ativos podem ter prioridades de execução. Essa propriedade pode ser usada no caso de existirem vários objetos ativos esperando a sua vez de serem executados. O tipo enumerado `TPriority`, da classe `CActive`, define os valores possíveis.

O construtor da classe `CActive` requer que uma prioridade seja determinada, de modo que as classes derivadas devem satisfazer esse requisito.

```
CMyActive::CMyActive()
: CActive (CActive::EPriorityStandard)
{}
```

O método `NewL()` é responsável por construir uma instância desse objeto, ao chamar o `ConstructL()`, como é feito em várias classes do Symbian C++.

Registro no escalonador

O escalonador de objetos ativos é representado pela classe `CActiveScheduler`. Para se registrar os objetos com o escalonador, é necessário chamar o método estático `Add()` dessa classe.

```
CMyActive::CMyActive()
: CActive (CActive::EPriorityStandard)
{
    CActiveScheduler::Add (this);
}
```

Essa chamada poderia ser realizada também no `ConstructL()` ou no `NewL()`. O objeto ativo será removido do escalonador quando for destruído, de modo que não é preciso fazer isso explicitamente.

(C) Copyright Nokia 2014. All rights reserved.

Chamada assíncrona

O método que representa a chamada assíncrona é responsável por iniciar a requisição relacionada com a chamada. O roteiro para se realizar a chamada é como este:

1. Antes de realizar a chamada, é importante verificar se já existe alguma pendente, porque na prática cada objeto ativo pode ter apenas uma requisição em andamento. Caso se tente submeter uma outra, algumas situações podem acontecer dependendo a implementação. Exemplos: gerar um erro do tipo *panic*, recusa em submeter outra requisição e cancelar a pendente e submeter uma nova;

2. Submeter a requisição, passando a variável `iStatus` para a função assíncrona. A função assíncrona irá usar essa variável para informar o status da requisição;

3. No caso da requisição ser bem-sucedida, o método `SetActive()` deve ser chamado para indicar que existe uma requisição pendente.

Aqui está um exemplo:

```
_LIT (KErrorStr, "Requisição já realizada");

void CMyCActive::InvokeService ()
{
   // checando por requisições em andamento
   __ASSERT_ALWAYS (!IsActive(), User::Panic (KErrorStr, KErrInUse));

   // chamada assíncrona
   RequestServiceProcessing (iStatus);

   // informando que a requisição está pendente
   SetActive();
}
```

Resposta ao término do serviço

Quando o processamento relacionado com a requisição terminar, o escalonador irá chamar o método `RunL()` do objeto ativo. O status atual da requisição pode ser conferido através do campo `iStatus`, aquele mesmo que foi passado para a função assíncrona no momento da chamada. O objeto ativo pode, se for o caso, submeter outra requisição a ele mesmo ou a outro objeto.

```
void CMyActive::RunL()
{
   if (iStatus == KErrNone)
   {
     // processamento normal do evento
   }
   else
   {
     // tratamento de algum erro ocorrido no processo
   }

}
```

É importante lembrar que o método `RunL()` não pode ser interrompido pelo escalonador ativo. Assim, caso essa função trave, os outros objetos ativos ficarão impedidos de responderem a eventos.

Cancelamento de requisições

Os objetos ativos precisam implementar o método `DoCancel()`, para interromper requisições pendentes. O sistema chama esse método através do `Cancel()` de `CActive`.

(C) Copyright Nokia 2014. All rights reserved.

Esse método não pode gerar abandonos (*leaves*), já que é potencialmente chamado no destrutor dos objetos ativos. Não é preciso verificar se existe alguma requisição pendente para poder continuar o procedimento de cancelamento, porque isso já é realizado pelos métodos que chamam `DoCancel()`.

Um ponto importante a se destacar é que quando um objeto ativo é cancelado, o método `RunL()` não é executado. Dessa forma, o método `DoCancel()` deve ser usado para se fazer liberação de recursos que ainda possam estar sendo usados, no caso de um cancelamento.

Tratamento de erros

Os abandonos (*leaves*) que ocorrerem em `RunL()` podem ser tratadas pelo método `RunError()`. O parâmetro representa o código de erro da exceção.

O método deverá retornar `KErrorNone` caso o erro tenha sido tratado. Se outro valor for retornado, o escalonador ativo será responsável por tratar o erro.

Possíveis erros

Algumas vezes o erro (*panic*) "`E32USER-CBASE 46`" pode ocorrer quando se usa objetos ativos. Esse erro também é conhecido como *stray signal panic*.

Essa exceção é gerada pelo escalonador ativo quando uma requisição termina mas não é possível identificar o objeto ativo responsável por tratá-la. Algumas possíveis causas para isso são:

- O objeto ativo não foi registrado no escalonador através do `CActiveScheduler::Add()`. Se isso não ocorrer, o escalonador não saberá que o objeto ativo existe;
- O método `SetActive()` não foi chamado após a chamada assíncrona. Se isso não ocorrer, o objeto aparecerá sempre como inativo para o escalonador;
- A requisição foi completada mais de uma vez, seja por ter feito a requisição mais de uma vez ao objeto ativo (antes da primeira ter acabado), ou por algum outro erro de programação;
- Cancelar uma requisição que não está associada com algum objeto ativo. Assim, quando o escalonador receber o evento de confirmação de cancelamento, ele não conseguirá identificar um objeto ativo associado com a requisição, e irá gerar o erro.

Outro erro que pode acontecer é o "`E32USER-CBASE 42`", que indica uma tentativa de se fazer uma nova requisição pelo serviço quando ainda existe uma requisição pendente. Em outras palavras, o erro é gerado ao se invocar o método `SetActive()` quando ainda existe uma requisição anterior não satisfeita (ou seja, ainda está ativo).

(C) Copyright Nokia 2014. All rights reserved.