OReK - Um Executivo de Tempo Real Orientado por Objectos

Arnaldo S. R. Oliveira, Luís Almeida, Valery Sklyarov

Abstract – **This paper discusses the implementation, features and use of the OReK kernel - an object-oriented, fully preemptive real-time Kernel implemented in C++. The OReK kernel is pratically platform independent, containing only small and localized code segments that are platform dependent. Currently it can be used in PC's with an Intel x86 compatible processor family with MSDOS, because it needs to configure and access the hardware directly. Allowed timming resolutions are in the dozen of microseconds to dozens of miliseconds range. In its current state it only offers services for task management and scheduling. However, topics for future work to extend capabilities and optimize performance were already identified.**

Resumo – **Este artigo discute a implementação, as funcionalidades e a utilização do executivo OReK - um executivo de tempo real orientado por objectos, completamente preemptivo e implementado em C++. A generalidade do executivo OReK é independente da plataforma, possuindo apenas segmentos bem localizados de código específico da plataforma. Actualmente pode ser utilizado em PC's com processador compatível com a família Intel x86 e em MSDOS, uma vez que necessita de configurar e aceder directamente ao hardware. As resoluções temporais permitidas vão desde a dezena de micro-segundo até às dezenas de mili-segundo. No estado actual fornece apenas serviços para gestão e escalonamento de tarefas. No entanto, estão já identificados pontos de trabalho futuro para extensão da funcionalidade e optimização do desempenho.**

Keywords – **Real-time systems, operating systems, object-oriented programming, task scheduling**

Palavras chave – **Sistemas de tempo real, sistemas operativos, programação orientada por objectos, escalonamento de tarefas**

I. Introdução

Um executivo é uma entidade de software responsável pela execução concorrente de tarefas ou processos. As funções normalmente realizadas por um executivo são o escalonamento, o lançamento, a comutação e a terminação de tarefas. Em sistemas simples é comum integrar num único módulo executável, isto é, de forma monolítica o executivo e as tarefas que constituem o sistema. Em sistemas mais complexos que necessitem, por exemplo, de gestão de memória, de serviços de rede ou de dispositivos de entrada/saída sofisticados é usual utilizar-se um sistema operativo, do qual o executivo é uma das peças chave. Um executivo de tempo real é um executivo capaz de gerir tarefas de tempo real destinando-se portanto a sistemas de tempo real.

Um sistema de tempo real é normalmente um sistema de controlo reactivo que responde continuamente a eventos produzidos pelo ambiente em que está inserido. A resposta é feita de acordo com uma estratégia predefinida e cumprindo algumas restrições temporais. A execução do sistema é despoletada por eventos que podem ser síncronos (periódicos) ou assíncronos (aperíodicos) e provenientes de várias fontes, tais como um sensor ou um temporizador. Em qualquer dos casos, a resposta ao evento é feita através da execução de uma tarefa ou processo, onde o evento é processado e desencadeada a respectiva reacção.

Os sistemas de tempo real encontram-se nas mais variadas aplicações, tais como:

- Militares e de defesa;
- Controlo de instalações químicas e nucleares;
- Robótica;
- Transportes e controlo aéreo;
- Telecomunicações, etc.

A utilização de executivos em sistemas de tempo real para fazer a gestão de tarefas tem a vantagem de simplificar o desenvolvimento, torná-los mais robustos e de proporcionar uma abstração do hardware, tornando-os independentes da plataforma, simplificando assim a portabilidade. Um executivo de tempo real tem a responsabilidade de executar as tarefas cumprindo as restrições temporais do sistema.

O executivo OReK teve a sua origem no executivo ReTMik. No entanto, enquanto o ReTMik foi desenvolvido em C, o OReK foi escrito em C++. A escolha desta linguagem foi motivada pelo Paradigma de Orientação por Objectos (POO) suportado pelo C++, o qual, como veremos mais à frente, proporciona algumas vantagens interessantes relativamente à linguagem C.

Além desta introdução este artigo possui mais 7 secções. Na segunda secção são resumidos alguns conceitos fundamentais sobre sistemas de tempo real e úteis para a compreensão do executivo OReK. Na terceira secção é apresentada a motivação para a utilização do C++ na implementação do executivo. Na secção IV é mostrado um exemplo de utilização do executivo OReK. Na secção V é descrita a sua arquitectura. Na secção VI são revelados alguns detalhes de implementação. Na secção VII são enumerados alguns possíveis tópicos de trabalho futuro. Finalmente,

a conclusão e a bibliografia encontram-se nas secções VIII e IX, respectivamente.

II. SISTEMAS DE TEMPO REAL

Ao contrário de outros tipos de sistemas, em que o importante é apenas a sequência de operações e/ou o desempenho médio, um sistema de tempo real é:

- Um sistema computacional capaz de responder a eventos dentro de restrições temporais precisas;
- Um sistema cujo funcionamento correcto não depende apenas do valor das saídas mas também do instante em que são produzidas;
- Um sistema cuja evolução temporal deve estar sincronizada com a do ambiente em que opera.

A. Tarefas

Tal como já foi referido acima, os sistemas de tempo real são, em geral, constituídos por um conjunto de tarefas ou processos que executam concorrentemente sobre um executivo. Uma tarefa é uma sequência de instruções que, na ausência de outras actividades, é executada ininterruptamente pelo processador até ser completada. De forma simplificada, uma tarefa pode estar a executar, pronta a executar ou bloqueada. Ao conjunto das tarefas que se encontram a executar ou prontas a executar chamam-se activas.

Em termos de níveis de criticalidade, as tarefas podem ser caracterizadas da seguinte forma:

- **Ordinárias** - se não possuírem quaisquer restrições temporais;
- **Soft real-time** - se o não cumprimento de uma restrição temporal causar apenas uma degradação do desempenho do sistema;
- **Hard real-time** - se o não cumprimento de uma restrição temporal causar efeitos catastróficos no sistema controlado.

Um sistema operativo capaz de gerir tarefas de tempo real designa-se por sistema operativo de tempo real. Se além disso, também suportar tarefas *hard real-time* é chamado sistema operativo *hard real-time*.

Em termos de modo de activação, as tarefas podem ser de dois tipos distintos:

- **Periódicas** (*time driven*) - a tarefa é automaticamente activada pelo núcleo em intervalos de tempo regulares;
- **Aperiódicas** (*event driven*) - a tarefa é activada assincronamente quando ocorrer um evento ou através da invocação explícita de uma primitiva de activação.

A.1 Parâmetros

Uma tarefa τ_i é caracterizada por vários parâmetros, entre os quais (figura 1):

- a_i - instante de activação;
- x_i - instante de execução;
- f_i - instante de terminação;
- d_i - *deadline* absoluta;

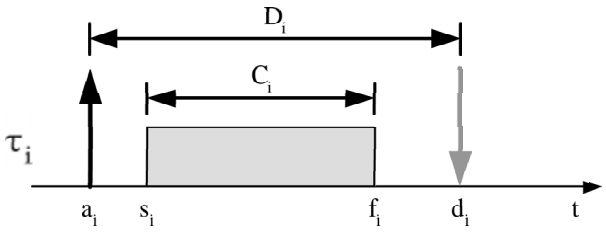

Figura 1 - Parâmetros temporais de uma tarefa.

- D_i - *deadline* relativa;
- W_i - tempo máximo de execução (*Worst Case Execution Time - WCET*);

Uma tarefa periódica, além dos parâmetros acima, tem também um período T_i associado.

Numa tarefa periódica caracterizada pelo terno $\tau_i(W_i, T_i, D_i)$ verificam-se as seguintes relações:

$$a_{i,k} = a_{i,k-1} + T_i$$

$$d_{i,k} = a_{i,k} + D_i$$

$$W_i \leq T_i$$

em que k é a k-ésima activação da tarefa.

B. Escalonamento

Um escalonamento é uma atribuição particular de tarefas ao processador. À estratégia usada para escolher uma tarefa entre as que se encontram activas e atribuir-lhe tempo de processador, chama-se algoritmo ou política de escalonamento.

O escalonamento em sistemas de tempo real pode ser efectuado de várias maneiras. Uma das mais utilizadas é o escalonamento baseado em prioridades. Nesta abordagem, a cada tarefa é atribuída uma prioridade calculada a partir das suas restrições temporais. A execução das tarefas é feita por um executivo baseado em prioridades. As prioridades são em geral atribuídas com base nos parâmetros período e *deadline*. As políticas mais utilizadas são as seguintes:

- *Rate Monotonic (RM)*;
- *Deadline Monotonic (DM)*;
- *Earliest Deadline First (EDF)*;
- *Least Slack First (LSF)*.

No algoritmo EDF seleciona-se para execução a tarefa com a *deadline* absoluta mais próxima, isto é, cada tarefa recebe uma prioridade proporcional à sua *deadline* absoluta. Este algoritmo possui as seguintes características:

- Prioridade dinâmica (d_i depende do instante de activação a_i);
- Preemptivo;
- Minimiza o número de mudanças de contexto relativamente às políticas RM e DM.

A figura 2 mostra um exemplo de 4 tarefas usando o

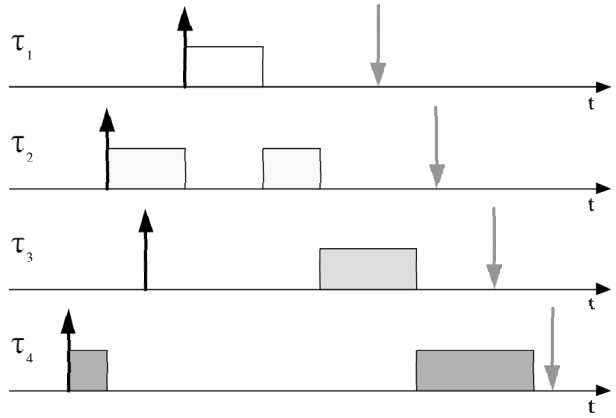

Figura 2 - Exemplo de um escalonamento de 4 tarefas usando o algoritmo EDF.

algoritmo EDF. Este é considerado óptimo entre todos os algoritmos de escalonamento e permite alcançar uma taxa de ocupação do processador de 100%.

O executivo OReK é preemptivo, suporta tarefas *hard real-time* periódicas, utiliza o algoritmo EDF para escalonamento das tarefas e permite resoluções temporais desde 10 micro-segundos até 50 milisegundos.

III. Orientação por Objectos

O Paradigma de Orientação por Objectos (POO) no desenvolvimento de software consiste na criação de aplicações constituídas por vários objectos que interagem. Cada objecto é uma estrutura de dados que integra também um conjunto de funções internas que operam sobre os campos de dados e um conjunto de funções de interface que permitem trocar informação com o exterior. A utilização do POO no desenvolvimento do executivo e na implementação das tarefas foi motivada por algumas potencialidades interessantes deste paradigma, nomeadamente, a abstracção de dados, a herança, o encapsulamento e o polimorfismo. Em conjunto, estes mecanismos permitem a construção de programas mais concisos e legíveis e, facilitam a reutilização de código pré-existente. Estas facilidades podem ser descritas sumariamente da seguinte forma:

- Abstacção de dados - uma linguagem baseada no POO permite definir novos tipos de dados (representados por classes na linguagem C++) e as operações que sobre eles podem ser executadas (métodos e operadores em C++). Esta funcionalidade, em conjunto com a capacidade de efectuar a sobrecarga das funções e dos operadores, permite utilizar intuitivamente os novos tipos de dados definidos pelo utilizador de forma análoga aos predefinidos na linguagem. Uma vez definida uma classe, podem-se declarar objectos dessa classe, isto é, criar instâncias da classe, da mesma forma que se declaram variáveis dos tipos predefinidos da linguagem.
- Herança - a partir de uma classe base podem ser derivadas uma ou mais classes. Através do mecanismo de herança é possível modificar e/ou estender numa classe derivada a funcionalidade da classe base. A classe derivada pode possuir novos atributos e métodos, bem como extender e/ou redefinir os métodos da classe base, facilitando assim a reutilização de código pré-existente. O desenvolvimento também é simplificado, uma vez que tudo o que é comum a um conjunto de classes pode ser colocado numa ou várias classe(s) base.
- Encapsulamento - no caso do C++, uma classe possui internamente variáveis e funções, também vulgarmente designadas por atributos e métodos, respectivamente. A visibilidade ou acessibilidade dos atributos e métodos pode ser individualmente especificada. Existem três níveis de acessibilidade distintos: privado, protegido e público. Por defeito os elementos são privados, isto é, são acessíveis apenas dentro da classe em que estão definidos. Os atributos e métodos protegidos são também acessíveis nas classes derivadas. Finalmente, um elemento público é visível em qualquer parte do programa onde o objecto possa ser acedido. O encapsulamento consiste geralmente em tornar privados ou protegidos os atributos da classe e públicos os seus métodos de interface. Isto significa que o acesso aos atributos é feito a partir dos métodos da classe, o que facilita a manutenção da integridade interna do objecto.
- Polimorfismo - este mecanismo permite que o método invocado para um dado objecto seja determinado durante a execução do programa em função do tipo efectivo do objecto, em vez de ser determinado estaticamente pelo compilador. No C++ o polimorfismo é implementado através através de funções virtuais e de ponteiros/referências para objectos.

No executivo OReK, uma tarefa é um objecto implementado em C++ através de uma classe derivada da *COReKTask*. Esta fornece um conjunto de serviços base para manipulação de tarefas, tais como criação, terminação, arranque, paragem, suspensão, reactivação, etc. Tal como qualquer classe, uma tarefa possui pelo menos um constructor e método(s), podendo também ter um destructor e atributos. Uma classe que implementa uma tarefa pode ser instanciada diversas vezes numa aplicação correspondendo cada uma a uma tarefa distinta que executa concorrentemente com as outras tarefas do sistema. Cada instância é tipicamente configurada durante a sua criação através de parâmetros do respectivo constructor ou método de inicialização. Uma tarefa pode também possuir vários métodos. Contudo, existe um mandatório e que define o ponto de entrada da tarefa, isto é, o método invocado pelo executivo quando é iniciada a execução da tarefa. A sua assinatura deve ser a seguinte:

```
virtual void Main();
```

A classe *COReKTask* não é instanciável porque é abstracta, uma vez que o método de entrada da tarefa

está definido como uma função virtual pura, isto é, não possui implementação.

Resumindo, na implementação do executivo OReK a herança é utilizada para colocar numa classe base astracta todas as estruturas de dados e de controlo comuns a todas as tarefas. O encapsulamento permite esconder essas estruturas. O polimorfismo simplifica a implementação do método de entrada da tarefa.

A. Tipos de Variáveis

Do ponto de vista de duração, num programa existem em geral três tipos de variáveis, correspondendo cada um a diferentes zonas ou segmentos de dados:

- Automáticas - declaradas no corpo de funções ou métodos. As variáveis automáticas são colocadas na pilha, criadas aquando da sua declaração e destruídas automaticamente quando o bloco onde foram declaradas termina.
- Estáticas - estas variáveis podem ser declaradas dentro ou fora das funções ou métodos, são colocadas no segmento de dados estáticos, criadas e inicializadas quando o programa arranca e destruídas automaticamente quando o programa termina.
- Dinâmicas - as variáveis deste tipo são acedidas através de ponteiros para uma zona de dados específica designada por *heap*. As variáveis dinâmicas são criadas e destruídas explicitamente durante a execução do programa usando funções para alocação e libertação de memória.

As linguagens baseadas no POO possuem ainda outro tipo de variável já mencionado acima, os atributos.

As variáveis automáticas são específicas de cada invocação de cada tarefa, isto é, não são preservadas entre diferentes activações da mesma tarefa. As variáveis estáticas são preservadas durante toda a execução do programa e partilhadas por todas as tarefas. Os atributos da tarefa são específicos de cada tarefa e preservados entre diferentes activações da mesma tarefa. Isto permite declarar facilmente variáveis que são específicas de cada tarefa e preservadas entre activações sem a necessidade de passar explicitamente parâmetros à função que implementa a tarefa.

IV. EXEMPLO

A figura 3 ilustra um exemplo muito simples de um sistema de tempo real implementado com o executivo OReK. O sistema possui duas tarefas semelhantes (diferentes instâncias da classe `CDemoTask`). Cada uma inverte uma saída de dados da porta paralela do PC a uma frequência especificada pelo utilizador. A definição da tarefa `CDemoTask` encontra-se na primeira metade da figura 3 e possui dois atributos, um constructor e o método de entrada. O primeiro atributo é uma máscara usada na inversão do bit sobre o qual a tarefa opera. O segundo atributo é uma cópia do valor actual das saídas de dados do porto paralelo, sendo partilhado por todas as tarefas do tipo `CDemoTask`. No constructor, com base no parâmetro fornecido pelo utilizador e, que identifica o bit sobre o qual a tarefa actua, é calculada a máscara referida acima. No método de entrada é lido o valor actual de todos os bits de dados da porta, invertido o bit pretendido e actualizado o estado da porta.

O programa principal (função `main`) começa por instanciar dois objectos (`task1, task2`). Seguidamente inicializa o executivo (função `COReKKernel::Initialize(...)`). Os argumentos desta função correspondem aos seguintes parâmetros de configuração:

- Número máximo de tarefas - 8
- Resolução temporal - 0.1 mseg

Em caso de erro de inicialização o programa termina. Caso contrário são criadas as tarefas, isto é, especificados os seus parâmetros temporais através do método `Create(...)` o qual também regista a tarefa no executivo. A existência de uma função `Create` simplifica a escrita do construtor da tarefa, uma vez que os parâmetros temporarais das tarefas são passados directamente ao núcleo. Além disso, facilita também a sinalização de situações de erro.

Seguidamente, é dada ordem ao executivo para arrancar as tarefas. A partir deste momento as tarefas são executadas concorrentemente com o programa principal. Finalmente, quando tiverem sido completados 60000 ciclos de execução (*ticks*) o programa principal chama a função de terminação do executivo terminando também de seguida.

Neste exemplo existe um problema. Se uma das terefas for suspensa na instrução

```
m_portValue ^= m_xorValue;
```

depois de ler o valor inicial de `m_portValue` mas antes de escrever o valor final na variável e a outra tarefa for executada completamente, quando a primeira tarefa for retomada vai partir de um valor incorrecto para a variável `m_portValue`. Este problema existe porque o atributo é comum às duas tarefas, sendo portanto um recurso partilhado. Por outras palavras, esta instrução constitui uma região crítica das tarefas. Duas tarefas não podem entrar simultaneamente numa região crítica onde seja feito o acesso a um recurso partilhado.

V. ARQUITECTURA

A figura 4 ilustra a constituição típica de um sistema de tempo real baseado no executivo OReK. O executivo cria uma camada de abstração que esconde alguns detalhes do hardware, em particular os relativos à gestão dos temporizadores e interrupções.

O sistema é composto por *N+1* tarefas, uma de entrada e *N* de tempo real. Cada tarefa possui código e dados. O código é partilhado por todas as instâncias da mesma classe de tarefas, enquanto os dados (atributos e variáveis automáticas) são específicos de cada tarefa. Uma aplicação baseada no executivo OReK é monolítica, isto é, consiste num único módulo exe-

```
#include <dos.h>
#include "OReK.h"

#define LPT1_PORT_ADDR     0x0378

class CDemoTask : public COReKTask
{
public:
  CDemoTask(uchar bit)
  {
    m_xorValue = 1 << bit;
  }

public:
  virtual void Main()
  {
    m_portValue ^= m_xorValue;
    outportb(LPT1_PORT_ADDR, m_portValue);
  }

protected:
  uchar m_xorValue;
  static uchar m_portValue;
};

uchar CDemoTask::m_portValue = 0x00;

int main(int argc, char* argv[])
{
  CDemoTask task1(0);
  CDemoTask task2(1);

  cerr << "Initializing OReK... ";
  if (COReKKernel::Initialize(8, 100) != OREK_INFO_SUCCESS)
  {
    cerr << "Failed\n";
    return 1;
  }
  cerr << "OK\n";

  cerr << "Creating tasks... ";
  if ((task1.Create(1, 1, 2) != OREK_INFO_SUCCESS) ||
      (task2.Create(4, 2, 20) != OREK_INFO_SUCCESS))
  {
    cerr << "Failed\n";
    COReKKernel::ShutDown();
    return 2;
  }
  cerr << "OK\n";

  COReKKernel::StartAllTasks();
  while (COReKKernel::GetTickCount() < 60000u);
  return COReKKernel::ShutDown();
}
```

Figura 3 - Exemplo de um sistema implementado com o executivo OReK.

cutável contendo quer a implementação das tarefas, quer o executivo.

O contexto de uma tarefa é composto por:

- Conteúdo do registo *Program Counter - PC* (par de registos CS:IP na arquitectura Intel x86);
- Conteúdo do registo de estado, se aplicável (*FLAGS* na arquitectura Intel x86);
- Conteúdo de todos os registos internos do processador utilizáveis pelas tarefas (registos de uso geral, registos de coprocessadores, etc.);
- Pilha e respectivo ponteiro.

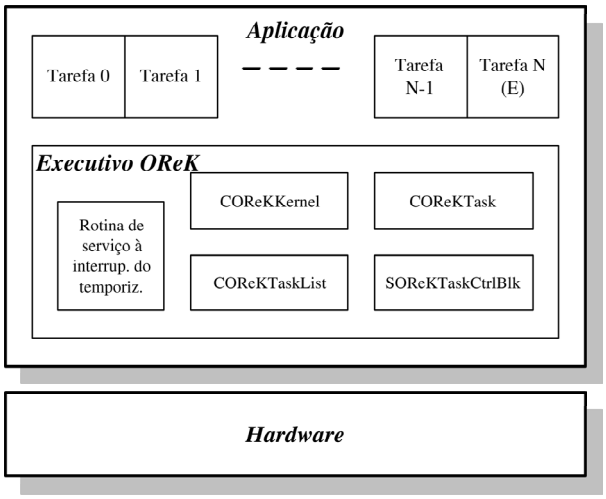

Figura 4 - Arquitectura de uma aplicação baseada no executivo OReK.

O temporizador de hardware gera interruções periodicamente. Sempre que ocorre uma interrupção é executado o algoritmo de escalonamento das tarefas e eventualmente feita a comutação de tarefas. Tudo isto é feito na rotina de serviço à interrupção do temporizador. No início desta rotina é necessário salvaguardar o contexto da tarefa actual, seguidamente determinar a próxima tarefa a executar e no final da rotina restaurar o contexto da tarefa que vai ser executada a seguir. No caso de ser executada uma operação que faça com que a tarefa activa transite para um estado não activo é também gerada uma interrupção de forma a que seja invocada a respectiva rotina de serviço onde é feito o escalonamento e a comutação de tarefas. O conteúdo do PC e do registo de estado é automaticamente salvaguardado quando ocorre uma interrupção. Os restantes registos têm que ser explicitamente salvaguardados e restaurados pelo executivo. A troca da pilha activa é feita através da alteração do valor de registo usado como ponteiro da pilha de forma a apontar para o topo da pilha da próxima tarefa que vai ser executada.

O executivo OReK é preemptivo, isto é, permite que a execução de uma tarefa seja interrompida em qualquer instante sem que para tal a própria tarefa tenha que se auto-suspender explicitamente. Independentemente do ponto onde a tarefa se encontre, o executivo é capaz de interromper a sua execução, comutar para outra(s) tarefa(s) e mais tarde retomar a execução da primeira tarefa a partir do ponto onde foi interrompida.

A. Núcleo

O componente principal do executivo OReK é o seu núcleo. Internamente, o núcleo serve as interrupções do temporizador, executa o escalonamento e efectua a comutação das tarefas. À aplicação disponibiliza os seguintes serviços:

- Inicialização e terminação do núcleo;
- Criação e destruição de tarefas;
- Arranque e paragem de tarefas;

- Adormecimento, suspensão e reactivação de tarefas;
- Informação temporal e de estado das tarefas;
- Gestão de grupos de tarefas.

B. Tarefas

As tarefas são objectos que encapsulam código e dados. O processamento é feito por tarefas. Uma tarefa pode estar num dos seguintes estados:

- *Stopped* (Parada)
- *Idle* (Inactiva)
- *Ready* (Pronta)
- *Running* (A executar)
- *Sleeping* (Adormecida)
- *Suspended* (Suspensa)
- *Blocked* (Bloqueada)

Qualquer tarefa que se encontre num dos estados *Idle*, *Ready*, *Running*, *Sleeping*, *Suspended* ou *Blocked* diz-se iniciada (*Started*), caso contrário diz-se parada (*Stopped*). Uma tarefa que se encontre num dos estados *Ready* ou *Running* diz-se activa (*Active*). A figura 5 ilustra os vários estados em que uma tarefa se pode encontrar, bem como as transições permitidas.

Uma tarefa quando é criada é colocada no estado *Stopped*, permanecendo nesse estado até ser explicitamente iniciada. Depois de iniciada é colocada no estado *Idle* até ser alcançado o instante da primeira activação. Nessa altura transita para o estado *Ready*. Uma tarefa no estado *Ready* está pronta a executar. A passagem para o estado *Running* é feita pelo algoritmo de escalonamento através da atribuição de tempo de processador à tarefa. Uma tarefa activa pode ser adormecida (passagem para o estado *Sleeping*) ou suspensa (passagem para o estado *Suspended*). O retorno ao estado *Ready* é automático no caso de uma tarefa adormecida e feito quando expirar o respectivo tempo. Uma tarefa suspensa deve ser reactivada explicitamente. Uma tarefa depois de terminar é colocada no estado *Idle* até à próxima activação. Uma tarefa activa é colocada no estado *Blocked* sempre que estiver à espera de um evento ou da libertação de um recurso partilhado. Este estado ainda não se encontra implementado no executivo OReK. Em qualquer estado pode ser dada ordem de paragem a uma tarefa, transitando neste caso para o estado *Stopped*. Uma tarefa depois de parada pode ser destruída, deixando de existir no sistema.

VI. IMPLEMENTAÇÃO

A linguagem escolhida para implementar o executivo foi o C++ por ser:

- Eficiente;
- Orientada por objectos;
- Popular e bem suportada por ferramentas para diversas plataformas.

A figura 4 ilustra também as componentes principais do executivo OReK, nomeadamente as classes `COReKKernel`, `COReKTask` e `COReKTaskList`, a estrutura `SOReKTaskCtrlBlk` e a rotina de serviço à interrupção do temporizador.

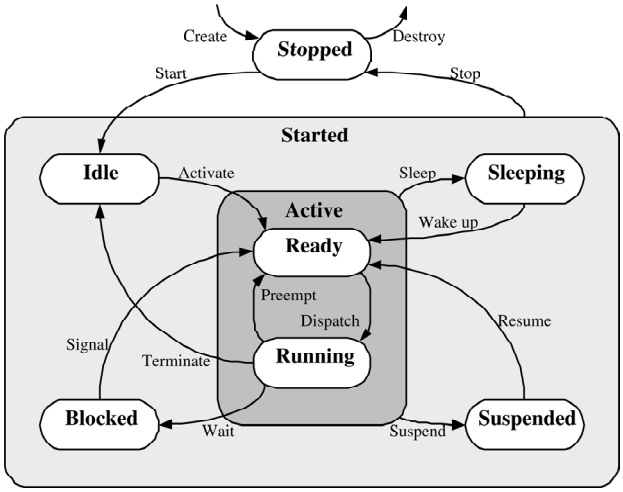

Figura 5 - Estados das tarefas e possíveis transições no executivo OReK.

A. Classes e Estruturas

Tal como já foi referido, a implementação do executivo consiste nas classes `COReKKernel`, `COReKTask` e `COReKTaskList` e na estrutura `SOReKTaskCtrlBlk`. As subsecções seguintes descrevem sucintamente cada um destes elementos.

A.1 A Classe `COReKKernel`

A classe `COReKKernel` implementa o núcleo do executivo. A figura 6 mostra o interface (métodos públicos) desta classe. Estes implementam os serviços disponibilizados pelo núcleo e enumerados acima. Todos os métodos e atributos desta classe são estáticos e a classe não é instanciável porque não faz sentido existir mais do que um núcleo no sistema.

A.2 A Classe `COReKTask`

A classe `COReKTask` fornece uma definição abstracta de uma tarefa. Também simplifica a manipulação de tarefas, guardando internamente o identificador da tarefa devolvido pelo núcleo aquando da criação da mesma. Sempre que necessário utiliza esse identificador para invocar a função do núcleo correspondente. A figura 7 mostra o interface desta classe e que consiste num construtor, num destrutor e nos métodos `Create(...)`, `Destroy(...)`, `Start(...)`, `Stop(...)`, `Sleep(...)`, `Suspend(...)`, `Resume(...)` e `GetState(...)`. Além destes métodos, define ainda a assinatura do método de entrada da tarefa (`virtual void Main() = 0`). Este método não está implementado nesta classe e deve ser definido em todas as classes que implementam tarefas concretas as quais têm que ser derivadas da classe `COReKTask`.

```
class COReKKernel
{
// Methods
public:
    static int Initialize(uint numMaxTasks, uint timeResolution);
    static int ShutDown();
    static uint GetTickCount();

    static int CreateTask(COReKTask* pTask,
                          uint period, uint deadline, uint firstActivation,
                          uint stackSize, uint groupId, handle* phTask);
    static int DestroyTask(handle hTask);

    static int StartTask(handle hTask);
    static int StopTask(handle hTask);

    static int StartTaskGroup(uint groupId);
    static int StopTaskGroup(uint groupId);

    static void StartAllTasks();
    static void StopAllTasks();

    static int SleepTask(handle hTask, uint nTicks);
    static int SuspendTask(handle hTask);
    static int ResumeTask(handle hTask);

    static int GetTaskState(handle hTask, TOReKTaskState* pTaskState);
    static int GetTaskGroupState(uint groupId, TOReKTaskState* pTaskState);
};
```

Figura 6 - Interface da classe `COReKKernel`.

```
class COReKTask
{
// Constructors / Destructors
public:
    COReKTask();
    virtual ~COReKTask();

// Methods
public:
    int Create(uint period, uint deadline,
               uint firstActivation,
               uint stackSize,
               uint groupId = 0);
    int Destroy();
    int Start();
    int Stop();

    int Sleep(uint nTicks);
    int Suspend();
    int Resume();

    int GetState(TOReKTaskState* pTaskState);

    virtual void Main() = 0;
};
```

Figura 7 - Interface da classe `COReKTask`.

```
typedef struct SOReKTaskCtrlBlk
{
    COReKTask* m_pTask;
    COReKTaskList* m_pTaskList;
    SOReKTaskCtrlBlk* m_pPrevious;
    SOReKTaskCtrlBlk* m_pNext;

    uint m_stackSize;
    uint* m_pStack;
    uint m_groupId;
    uint m_period;
    uint m_deadline;
    uint m_firstActivation;

    int m_activation;
    int m_ponctuality;

#ifdef __MSDOS__
    uint _SP;
    uint _BP;
#endif

}TOReKTaskCtrlBlk;
```

Figura 8 - Definição da estrutura `SOReKTaskCtrlBlk`.

A.3 A Estrutura `SOReKTaskCtrlBlk`

A estrutura `SOReKTaskCtrlBlk` define o bloco de controlo da tarefa (TCB - *Task Control Block*). Esta estrutura possui vários campos onde são armazenados os parâmetros de configuração e guardado o estado da respectiva tarefa. A sua definição possui duas partes (ver figura 8): a primeira, independente da plataforma e a segunda dependente da plataforma.

O campo `m_pTask` é um ponteiro para um objecto ou instância de uma classe derivada da `COReKTask` onde é implementada a funcionalidade da tarefa propriamente dita. É através deste ponteiro que o executivo invoca o método de entrada da tarefa. Como veremos de seguida, uma tarefa está numa das listas de tarefas internas do núcleo. Cada lista possui um estado associado. O campo `m_pTaskList` é um ponteiro para a lista

na qual a tarefa se encontra num dado momento. Os campos `m_pPrevious` e `m_pNext` são ponteiros utilizados pela classe `COReKTaskList` na implementação das listas biligadas de TCB's.

À primeira vista pode parecer que uma solução mais eficiente para implementar as listas de tarefas poderia ser baseada numa fila (*FIFO - First In, First Out*) circular. Contudo, como o escalonamento das tarefas é dinâmico, podendo em qualquer instante ser colocada uma tarefa no estado *Ready* com uma *deadline* arbitrária, na realidade não existem filas de tarefas mas sim listas ordenadas. Para optimizar simultaneamente o acesso arbitrário a qualquer bloco assim como a inserção ou remoção de blocos, as listas de tarefas são biligadas e implementadas sobre uma tabela (*array*) de blocos de tamanho predefinido alocada estaticamente. Sobre a mesma tabela são implementadas várias listas correspondendo cada uma a um dos estados em que uma tarefa se pode encontrar. Além dessas, existe também uma lista de blocos livres. A utilização de uma tabela tem também a vantagem de simplificar a validação do identificador *handle* da tarefa, o qual corresponde ao índice to TCB dentro da tabela. A utilização de uma tabela estática de tamanho predefinido tem também a vantagem de eliminar o tempo de alocação/libertação de memória necessário numa solução dinâmica, obviamente à custa de uma diminuição da flexibilidade. No entanto, é importante notar que é sempre possível realizar a realocação da tabela de TCB's, embora essa seja uma operação que obriga a suspender temporariamente a execução das tarefas.

O campo `m_stackSize` é o tamanho da pilha associada à tarefa medido em palavras nativas da arquitecura base (*word* de 16 bits para a plataforma Intel x86/MSDOS). O valor deste campo é estabelecido durante a criação da tarefa não podendo ser alterado posteriormente. O campo `m_pStack` é um ponteiro para o primeiro endereço de memória reservado para a pilha.

O campo `m_groupId` identifica o grupo a que a tarefa pertence.

Os campos `m_period`, `m_deadline`, `m_firstActivation` armazenam os parâmetros temporais da tarefa, respectivamente o período, o tempo limite de execução e o instante da primeira activação relativamente ao arranque da tarefa.

Os campos `m_activation` e `m_ponctuality` são dois contadores descrescentes usados no escalonamento das tarefas. O primeiro possui o número de unidades temporais que faltam para a (re)activação da tarefa, sendo usado em todos os estados do macroestado *Started* que possuam transições para o estado *Ready*. O contador `m_ponctuality` armazena o número de unidades de tempo que a tarefa possui para terminar antes de ser atingida a sua *deadline*. Este contador é utilizado em todos os estados do macroestado *Started* à excepção do estado *Idle*.

Ao contrário dos campos anteriores, que são completamente independentes da plataforma, os campos `_SP` e `_BP` são específicos da plataforma MSDOS/Intel x86 e servem para salvaguardar durante a preempção da tarefa o valor dos registos `SP` e `BP` do processador, respectivamente.

A.4 A Classe `COReKTaskList`

A classe `COReKTaskList` implementa uma lista biligada de estruturas do tipo `SOReKTaskCTRLBlk` sobre uma tabela predefinida e de tamanho fixo. Disponibiliza os serviços tradicionais para inserção e remoção de elementos da lista além de outras funcionalidades úteis para o escalonamento de tarefas. Para optimizar o desempenho, esta classe utiliza as capacidades *inline* do C++.

B. Independência da Plataforma

A generalidade do núcleo é independente da plataforma e está escrito em C++ portável. No entanto, existem pequenas partes que são específicas da plataforma, tais como as funções de programação do temporizador, as partes onde se acede aos registos do processador e alguns segmentos da rotina de serviço à interrupção. Assim, apesar de no estado actual o executivo OReK ser suportado apenas na plataforma Intel x86/MSDOS, houve o cuidado de isolar todos os blocos de código que sejam dependentes da plataforma. Desta forma é relativamente simples portar o executivo para outras plataformas.

Um aspecto específico da arquitectura é a pilha da tarefa. De forma a poder ser feita a comutação de tarefas, sempre que ocorra uma interrupção do temporizador, o contexto da tarefa suspensa deve ser armazenado parte na pilha e parte no TCB correspondente.

C. Escalonamento das Tarefas

Tal como já foi referido acima, devido ao seu bom desempenho e simplicidade, o algoritmo de escalonamento adoptado foi o *Earliest Deadline First* - EDF. A sua implementação revelou-se bastante simples. O núcleo possui internamente várias listas de tarefas, uma por cada estado em que uma tarefa se pode encontrar, mais uma lista de TCB's livres. As listas *Idle*, *Ready* e *Sleeping* são listas ordenadas, isto é, os TCB's são inseridos por ordem decrescente de um dos seus contadores de controlo de forma a optimizar a sua manipulação. Na lista *Idle* e *Sleeping* o contador usado como chave de inserção é o `m_activation`, enquanto na lista *Ready* é o `m_ponctuality`. As restantes listas não são ordenadas, sendo os TCB's colocados em geral no final da lista e retirados de forma arbitrária.

O contador `m_activation` dos TCB's pertencentes às listas *Idle* ou *Sleeping* é decrementado todas as unidades de tempo. Quando atingir o valor 0, significa que a respectiva tarefa deve transitar para o estado *Ready*.

Como nas listas *Idle*, *Ready* e *Sleeping*, os TCB's estão ordenados por ordem decrescente do contador de controlo, os elementos são sempre retirados do início.

O contador `m_ponctuality` dos TCB's pertencentes a uma das listas *Ready*, *Sleeping*, *Suspended* ou *Blocked*

também é decrementado todas as unidades de tempo. Quando atingir o valor 0, significa que foi alcançado o tempo limite de execução e que se a tarefa ainda não tiver terminado, ocorreu uma situação de *deadline miss*.

As tarefas são retiradas da lista *Blocked* quando ocorre um evento ou o recurso partilhado fica disponível. Os TCB's são retirados das listas *Stopped* e *Suspended* quando são explicitamente iniciadas ou retomadas, respectivamente.

O escalonamento baseado em listas ordenadas é bastante simples, uma vez que se baseia unicamente no decremento de contadores e na inserção/remoção de elementos de listas.

Inicialmente todas as listas estão vazias excepto a de TCB's livres. Na figura 9 é ilustrada a situação das listas correspondente aos seguintes estados das tarefas (assumindo um sistema com o máximo de 6 tarefas):

- Tarefa 0 e 1 - *Ready*;
- Tarefa 2 - *Running*;
- Tarefa 3 - *Idle*;
- Tarefa 4 - *Stopped*;
- Tarefa 5 - *Free*;

D. Utilização

Finalmente, para concluir esta descrição sobre o executivo OReK falta apenas descrever a sua utilização. O executivo OReK é disponibilizado na forma de ficheiros objecto (**.obj*) ou numa biblioteca (*OReK.lib*), o que significa que deve ser associado com a aplicação durante a geração do módulo executável. Tal como já foi referido acima, a construção de um sistema consiste na criação de um módulo executável monolítico que integra quer a implementação das tarefas, quer o executivo de tempo real. Do ponto de vista do utilizador os ficheiros que importa considerar são os seguintes:

- *OReK.h* - ficheiro geral de interface que deve ser incluído nos ficheiros fonte da aplicação onde são invocadas funções do executivo;
- *OReK.lib* - ficheiro de implementação que deve ser associado com a aplicação.

VII. Trabalho Futuro

Nas seguintes subsecções vão ser descritas algumas ideias e possíveis direcções de trabalho futuro, as quais incidem essencialmente sobre os seguintes tópicos:

- Suporte para diferentes tipos de tarefas;
- Diversificação dos algoritmos de escalonamento e análise de escalonabilidade;
- Implementação de mecanismos para sincronização e comunicação entre tarefas;
- Alargamento da gama de plataformas suportadas;
- Análise e optimização do desempenho do executivo;
- Concepção e implementação de hardware dedicado para escalonamento de tarefas.

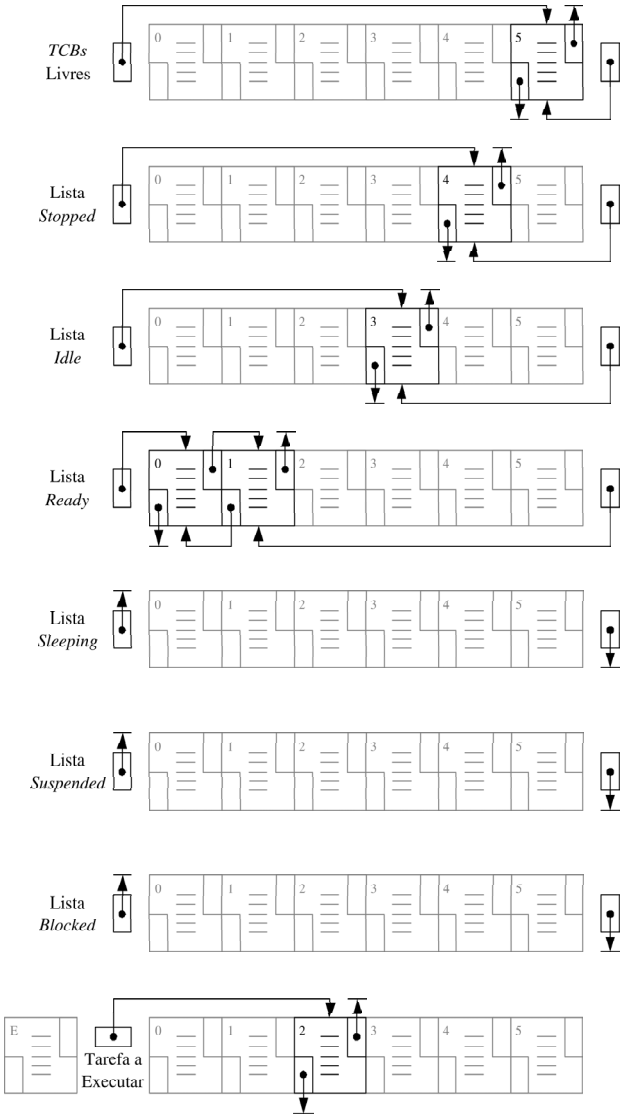

Figura 9 - Estados das listas de TCB's em pleno functionamento.

A. Suporte para Diferentes Tipos de Tarefas

No estado actual o executivo OReK apenas suporta tarefas críticas periódicas de tempo real (*hard real-time periodic*). Seria interessante estender o núcleo com mecanismos que possibilitassem a coexistência de diferentes tipos de tarefas no mesmo sistema, tais como:

- Tarefas críticas periódicas de tempo real (*hard real-time periodic*);
- Tarefas críticas aperiódicas de tempo real (*hard real-time aperiodic*);
- Tarefas não críticas periódicas (*soft real-time periodic*);
- Tarefas não críticas aperiódicas (*soft real-time aperiodic*);
- Tarefas ordinárias (*non real-time*).

Por sua vez, as tarefas ordinárias podem ser decompostas em várias níveis de prioridade, tais como *elevada*, *normal*, *baixa*, etc. A tabela I mostra os atributos que caracterizam cada um destes tipos de tarefas.

Tipo de Tarefa	Período	Tempo limite
Hard real-time periódica	X	X
Hard real-time aperiódica		X
Soft real-time periódica	X	
Soft real-time aperiódica		
Ordinária		

Tabela I
ATRIBUTOS DOS VÁRIOS TIPOS DE TAREFAS.

B. *Diversificação dos Algoritmos de Escalonamento e Análise de Escalonabilidade*

A política de escalonamento EDF, à semelhança da RM e da DM, não tem em consideração o tempo máximo de execução da tarefa na realização do escalonamento. O algoritmo LSF, com base no conhecimento do tempo máximo de execução das tarefas, permite fazer uma planificação mais precisa, verificar a escalonabilidade de um conjunto de tarefas e antecipar dinamicamente a ocorrência de violações temporais.

C. *Implementação de Mecanismos para Sincronização e Comunicação entre Tarefas*

Tal como já foi ilustrado no exemplo acima, o executivo OReK ainda não oferece qualquer solução que garanta uma execução correcta de tarefas concorrentes com acesso a recursos partilhados. Assim, um possível ponto de trabalho futuro é a concepção e implementação de mecanismos de comunicação e sincronizaçao entre tarefas tais como:

- Memória partilhada;
- Mensagens;
- Semáforos;
- Eventos.

D. *Alargamento da Gama de Plataformas Suportadas*

Actualmente, o executivo é suportado apenas na plataforma Intel x86/MSDOS. Devido às limitações impostas por uma plataforma de 16 bits, seria interessante portar o executivo para uma plataforma de 32 bits. Contudo, a necessidade de aceder directamente ao hardware impõe algumas restrições.

E. *Análise e Optimização do Desempenho do Executivo*

A plataforma suportada (Intel x86/MSDOS) não fornece mecanismos que permitam analisar com precisão o desempenho do executivo OReK. Um aspecto interessante será comparar o desempenho do núcleo implementado usando a linguagem C *versus* a linguagem C++. As variantes de interesse seriam as seguintes:

- Núcleo implementado em C / Tarefas implementadas como funções ordinárias em C;
- Núcleo implementado em C / Tarefas implementadas como classes em C++;
- Núcleo implementado em C++ / Tarefas implementadas como classes em C++.

A utilização do C++ como linguagem de implementação do núcleo tem também a vantagem de flexibilizar a implementação das tarefas, uma vez que é bastante simples disponiblizar uma API híbrida C/C++. O compilador usado actualmente é o Borland C++ 3.0. O emprego de um compilador mais recente capaz de produzir código mais eficiente poderá também contribuir para um melhor desempenho do executivo.

F. *Concepção e Implementação de Hardware Dedicado para Escalonamento de Tarefas*

A existência de dipositivos lógicos programáveis de elevada capacidade (e.g. FPGA) capazes de operar a frequências de relógio elevadas juntamente com o conceito de arquitecturas reconfiguráveis motiva a construção de coprocessadores de hardware dedicados ao escalonamento, sincronização e comunicação de tarefas. Este conceito pode ainda ser levado mais longe através da integração de um processador de uso geral com um coprocessador de escalonamento numa única FPGA. A própria arquitectura do processador pode inclusivamente ser optimizada para sistemas de tempo real multi-tarefa através da implementação em hardware de suporte para execução paralela de tarefas. Todas estas optimizações permitem reduzir substancialmente a sobrecarga computacional associada ao escalonamento e comutação de tarefas existente nos sistemas de tempo real convencionais baseados em processadores de uso geral. A reconfiguração pode inclusivamente ser utilizada para implementar dinamicamente diferentes algoritmos de escalonamento.

VIII. CONCLUSÃO

O executivo OReK descrito neste artigo fornece um conjunto de serviços básicos que simplificam a implementação de sistemas de tempo real constituídos por várias tarefas concorrentes. Dado que os serviços disponibilizados consistem apenas na gestão e escalonamento de tarefas, a gama de sistemas em que este executivo pode ser utilizado é bastante restricta. No entanto, é importante referir que este não é um trabalho concluído. Os pontos de trabalho futuro apresentados na secção anterior irão constituir uma boa base de investigação e desenvolvimento com o objectivo de alargar a gama de sistemas alvo, bem como melhorar consideravelmente o seu desempenho. A página

`http://www.ieeta.pt/~arnaldo`

possui informação actualizada sobre o executivo OReK, incluindo publicações e, num futuro próximo, código fonte e compilado.

IX. BIBLIOGRAFIA

1. Buttazo, G. (1997). Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers.
2. Liu, J.W.S. (2000). Real-Time Systems. Prentice Hall.
3. Krishna, C.M. and K. Shin (1997). Real-Time Systems. McGraw-Hill.
4. Stroustrup, B. (1995). The C++ Programming Language (second edition). Addison-Wesley Publishing Company.