

# ***Sistemas de Tempo-Real***

## **Aula 3**

### **Executivos (*kernels*) de tempo-real**

**Os estados de uma tarefa  
Arquitetura genérica de um executivo de tempo-real  
Estruturas e funções típicas do executivo**

Adaptado dos slides desenvolvidos pelo Prof. Doutor Luís Almeida  
para a disciplina “Sistemas de Tempo-Real”  
Revisto em 28.Set.2011 por Paulo Pedreiras

# Aula anterior (2)

- Modelos computacionais (**modelo de tempo-real**)
- Tarefas de tempo-real: periódicas, esporádicas e aperiódicas
- Restrições temporais do tipo **deadline**, janela, sincronismo e distância
- Implementação de tarefas e utilização de um **kernel multitasking**
- **Controlo lógico e controlo temporal**
- Tarefas **event-triggered** e **time-triggered**

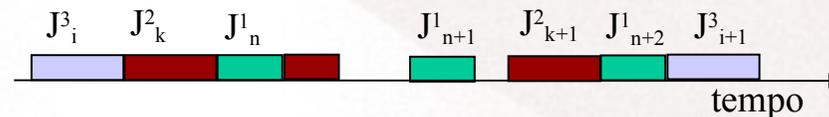
# Estados de uma tarefa

## Criação de uma tarefa

associação do código (e.g. função em linguagem “C”) a um espaço de variáveis privado (*private stack*) e a uma estrutura de gestão (*task control block – TCB*)

## Execução de uma tarefa

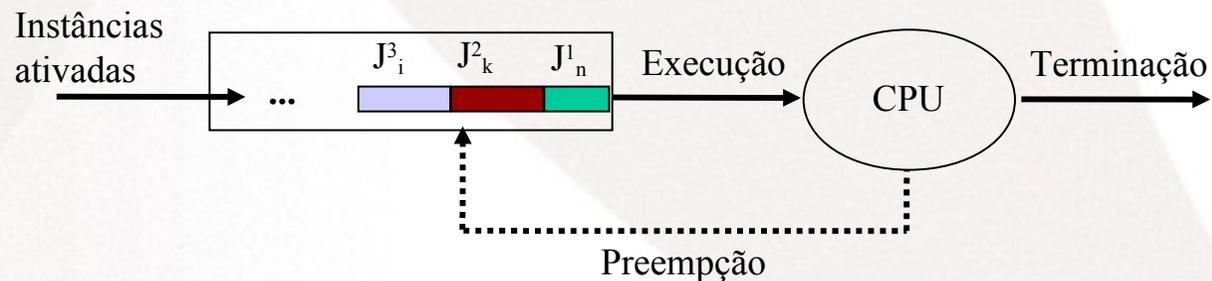
Execução concorrente do código da tarefa, usando o respetivo espaço privado de variáveis, sob controlo do *kernel*, com reativação de cada instância periodicamente ou como resposta a um evento externo.



# Estados de uma tarefa

## Execução das instâncias das tarefas

As instâncias **prontas a executar**, i.e. depois de ativadas, aguardam em **fila** pela atribuição de CPU (para execução). A fila é **ordenada** por um determinado critério de **escalonamento**, não necessariamente por ordem de chegada!

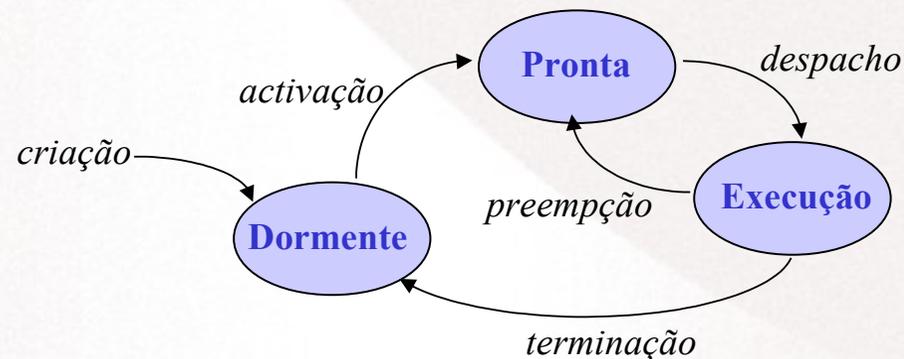


# Estados de uma tarefa

## Estados dinâmicos da tarefa

As instâncias das tarefas podem estar a **aguardar** execução (prontas) ou em **execução**. Após terminação de cada instância, a tarefa respetiva fica num estado dormente (*idle*), a aguardar a ativação da próxima instância.

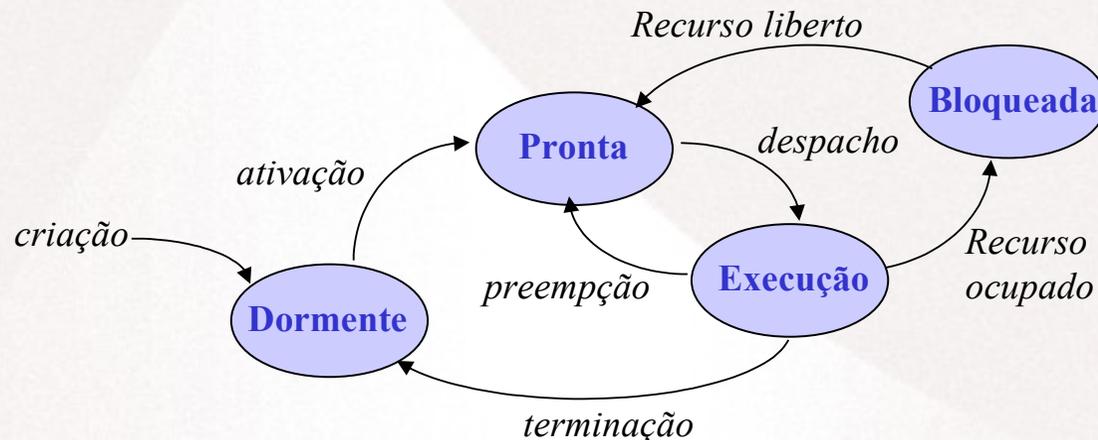
Assim, considera-se que uma tarefa pode estar **pronta**, em **execução** ou **dormente**.



# Estados de uma tarefa

## Outros estados: Bloqueio

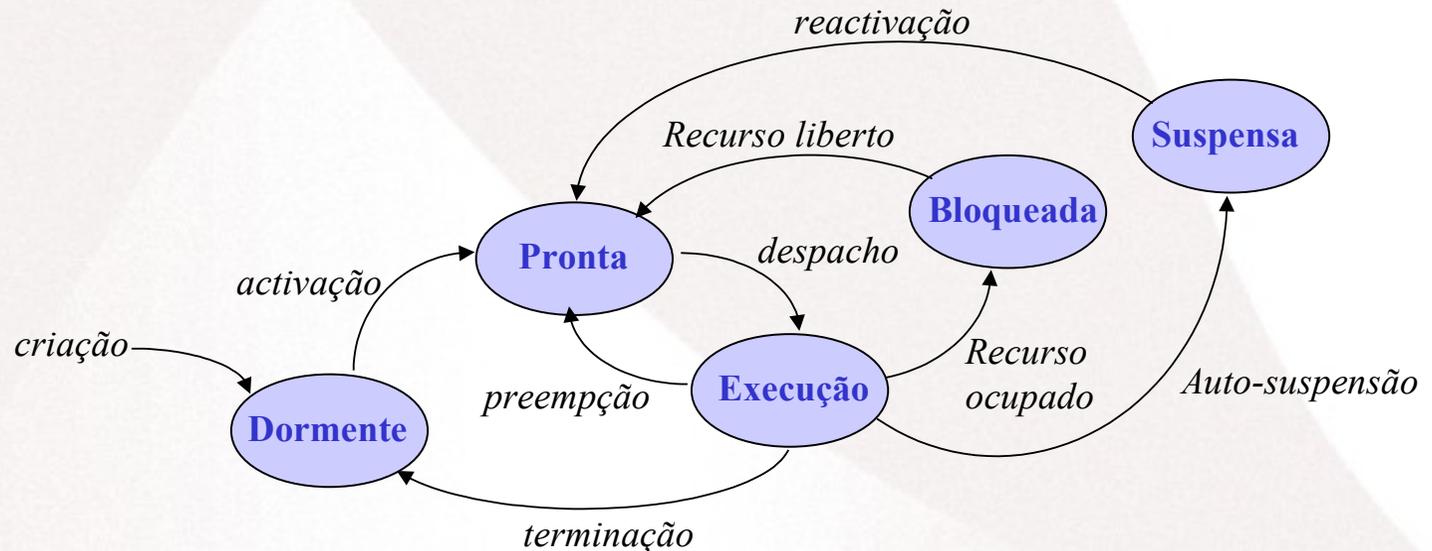
Quando uma tarefa em execução tenta aceder a um recurso partilhado (e.g. uma porta de comunicação, um *buffer* em memória) que está ocupado em modo exclusivo por uma tarefa pronta, a primeira diz-se que fica **bloqueada**. Quando o recurso é libertado, a tarefa bloqueada fica novamente pronta para execução.



# Estados de uma tarefa

## Auto-suspensão (sleep)

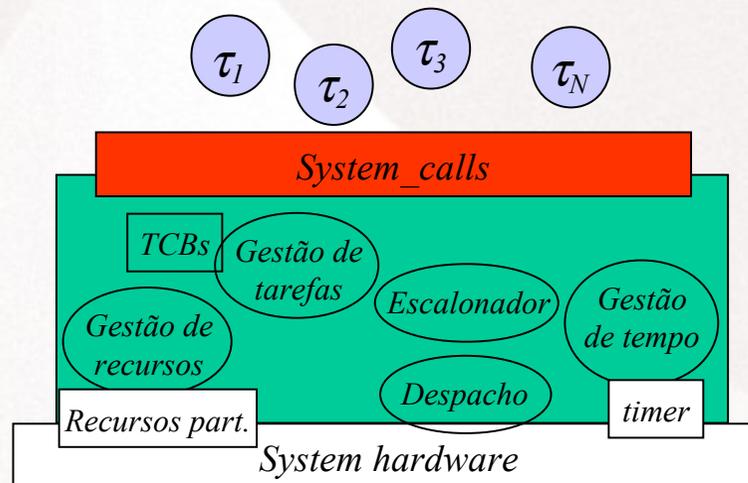
Em certas aplicações pode ser útil uma tarefa em execução auto-suspender-se e retomar a execução mais tarde, ainda durante a mesma instância. Diz-se que a tarefa fica **suspensa**.



# Arquitetura de um kernel tempo-real

## Serviços básicos

- Gestão de tarefas (criação , destruição, ativação inicial, estado)
- Gestão do tempo (ativações, policiamento, medição de intervalos)
- Escalonamento de tarefas (escolha da tarefa a executar)
- Despacho de tarefas (colocação em execução)
- Gestão de recursos partilhados (*mutexes*, semáforos, monitores)



# Estruturas de gestão

## TCB (task control block)

Esta estrutura é fundamental num *kernel* e serve para caracterizar a tarefa, bem como para gerir a respetiva execução.

Alguns campos usuais:

- Identificador
- Ponteiro para o código a ser executado
- Ponteiro para o *stack* privado (salvaguarda do contexto)
- Atributos de ativação (periódica (primeira act), esporádica, aperiódica)
- Atributos de criticalidade (*hard*, *soft*, não *real-time*)
- Outros atributos (*deadline*, prioridade)
- Estado dinâmico de execução e outras variáveis para controlo de ativação (*timers* de SW, *deadline* absoluta)

# Estruturas de gestão

## TCB do RTKPIC

```
typedef struct {
    unsigned char id;           /* task id - 0..14 */
    void (*func_ptr)(void);    /* task first instruction address */
    unsigned char state;       /* task state */
    unsigned int period;        /* task period in ticks */
    unsigned int deadline;     /* task deadline relative to activation */
    unsigned long nx_activ;    /* task next activation in absolute ticks */
    unsigned long nx_deadline; /* task next deadline in absolute ticks */
    unsigned char priority;    /* task priority */
} TASK;

/* Number of tasks */
#define NTASKS      14          /* main + 13 user tasks */

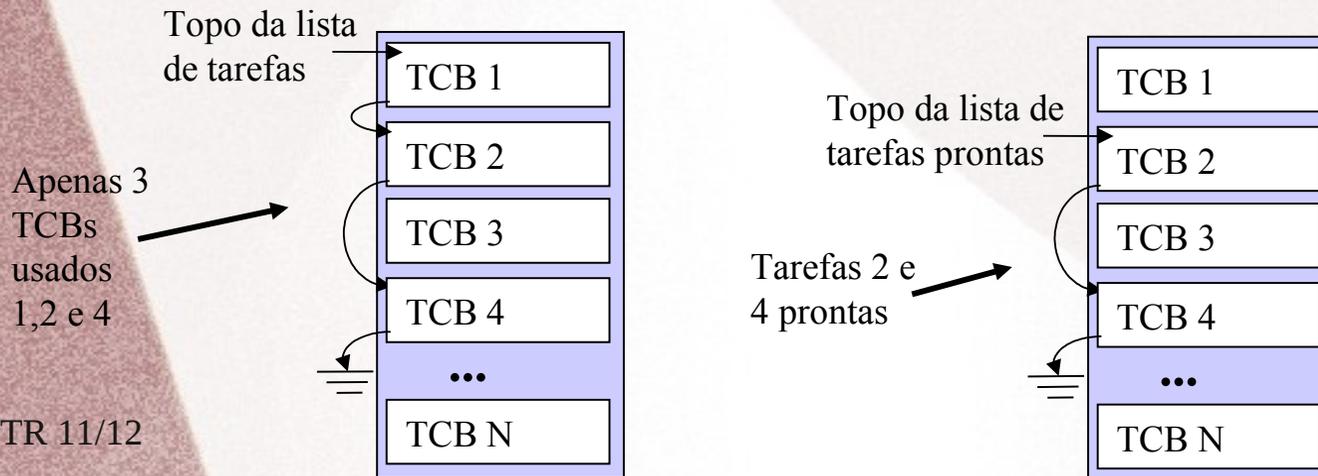
/* Task Control Table */
TASK tcb[NTASKS]
```

# Estruturas de gestão

## Estrutura de TCBs

Normalmente, os TCBs estão definidos num *array* estático mas estruturados segundo uma lista ligada para facilitar buscas sobre o conjunto das tarefas (poderá haver mais TCBs que tarefas!)

Qualquer lista ordenada (e.g. das tarefas prontas – *ready queue*) pode ser facilmente realizada sobre a estrutura de TCBs através de um ponteiro (pode ser um índice) para o próximo TCB na lista.



# Estruturas de gestão

## Estrutura de TCBs no RTKPIC18

O RTKPIC18 foi concebido para aplicações com pequeno número de tarefas. Neste caso optou-se por não realizar as filas de tarefas com listas. Sempre que é necessário fazer uma busca, e.g. Para efetuar a ativação de tarefas, percorre-se todo o *array* !

Percorre  
todo o  
array

Testa se  
deve  
activar

***/\* Secção do "tick handler" \*/***

```
for (temporary_i = 1; temporary_i < n_task; temporary_i++)  
{
```

```
    temporary_task_i = tcb + temporary_i;  
    if (temporary_task_i->nx_activ == system_clock)  
    {  
        /* new activation */
```

```
        if ((temporary_task_i->state == READY) || (temporary_task_i->state == RUN))  
            deadline_miss |= (0x01 << temporary_i); /* deadline missing */
```

Altera o  
estado e  
invoca o  
escalador

```
        temporary_task_i->state = READY;  
        temporary_task_i->nx_activ += temporary_task_i->period;  
        temporary_task_i->nx_deadline += temporary_task_i->deadline;
```

```
        if ((preempt_sys == PREEMPT) || (run_task_id == 0))  
            Call_scheduler = 1; /* New task READY: Call scheduler */
```

```
        if (sch_alg == EDF)  
            Call_EDF = 1; /* Call EDF priority set */
```

```
    }
```

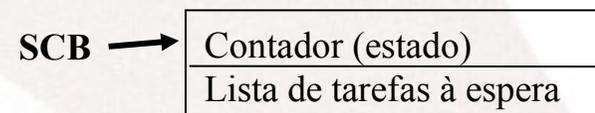
Testa se  
ocorreu  
violação de  
deadline

# Estruturas de gestão

## Acesso a recursos partilhados

Os recursos partilhados com acesso exclusivo (**regiões críticas**) têm, tal com o CPU, de ser geridos de forma apropriada de modo a serem acedidos por uma tarefa de cada vez. Uma forma comum de efetuar este controlo é recorrer a **flags atómicas** (*mutexes*), **monitores** (execução não preemptiva) ou **semáforos**.

No caso de se utilizarem **semáforos**, deverá haver uma estrutura por semáforo que indique o respetivo estado bem como a lista de tarefas a aguardar acesso – **SCB (semaphore control block)**



No **RTKPIC** apenas existe apenas controlo da **preempção**, a qual pode ser inibida.

# Funções de gestão

## Gestão do tempo

A gestão do tempo num *kernel* é fundamental e serve para:

- **Ativar** as tarefas periódicas
- **Verificar** cumprimento de restrições temporais
- **Medir** intervalos de tempo (inclusive para **auto-suspensão**)

É efetuada com recurso a um *timer* do sistema. O timer pode ser programado como:

- **Periodic tick**: gerar interrupções periódicas (ticks). O respetivo *handler* faz a gestão do tempo. Os atributos temporais são múltiplos inteiros do *tick*.
- **Single-shot/One-shot/tickless**: o timer é programado para gerar interrupções apenas nos precisos instantes em que existe ativação de uma ou mais tarefas.

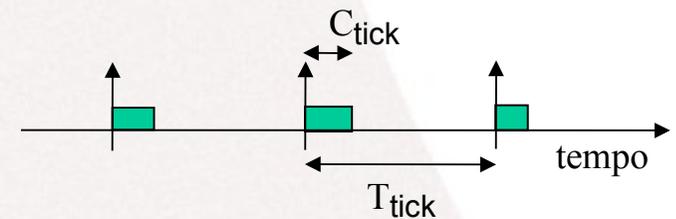
# Funções de gestão

## Sistemas baseados em tick

A duração do tick estabelece a resolução temporal do sistema.  
Quanto menor o *tick*, melhor a resolução !

E.g. tick de 10ms => períodos das tarefas  $T_1=20\text{ms}$ ,  $T_2=1290\text{ms}$ ,  $T_3=25\text{ms}$

A gestão temporal (atendimento do *tick*) representa *overhead* ( $C_{\text{tick}}/T_{\text{tick}}$ )  
Quanto maior o *tick*, menor o *overhead* !!



Compromisso: ***tick* = MDC ( $T_i$ ,  $i=1..N$ )**

E.g.  $T_1=20\text{ms}$ ,  $T_2=1290\text{ms}$ ,  $T_3=25\text{ms}$  =>  $\text{MDC}(20,1290,25)=5\text{ms}$   
mas ***tick* > min\_tick** imposto pela velocidade do CPU !

# Funções de gestão

## Medição de intervalos de tempo

Em sistemas com *ticks*, o *kernel* mantém uma variável que conta o n° de *ticks* desde a respetiva ativação

- e.g. no RTKPIC “unsigned long system\_clock”, lida pela macro `get_sys_time()`
- com *tick*=10ms, esta variável faz *wrap around* após 1,6 anos

Para maior precisão é necessário ler diretamente o *timer*. Para maior longevidade é necessário usar outro contador suplementar.

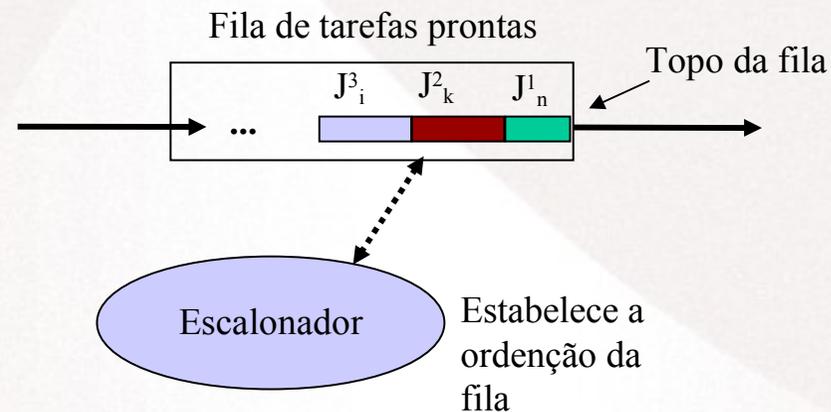
Em CPUs Pentium, com um relógio de 1GHz, o TSC faz *wrap around* após 486 anos !!!

# Funções de gestão

## Escalonador

Escolhe qual a **próxima tarefa** a executar de entre as **tarefas prontas**

Deve usar um **critério determinístico** para permitir calcular o atraso máximo (pior caso) que uma tarefa pode sofrer na fila

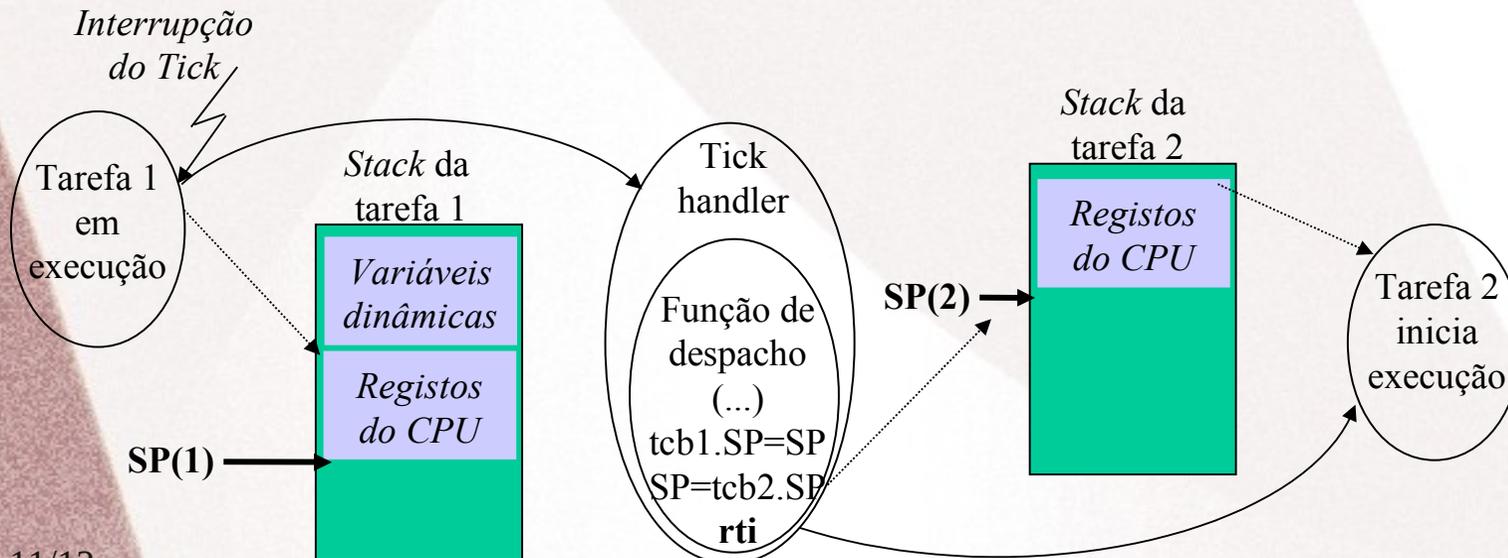


# Funções de gestão

## Despacho

Coloca uma tarefa seleccionada em execução.

Se o sistema admitir preempção pode ser necessário interromper a tarefa que está em execução. Nestes sistemas o mecanismo de despacho é baseado em manuseamento do *stack*.



# ReTMiK – Real Time Micro-mouse Kernel

<http://sweet.ua.pt/~lda/retmik/retmik.html>

- Baseado em *ticks*.
- Corre em arquiteturas x86 (há uma versão para PC)
- Código das tarefas é acíclico
- Escalonador inserido no código do *kernel* (necessário recompilar para alterar)
- Sincronização entre tarefas por inibição de preempção (monitor)
- IPC por variáveis globais
- Gera aplicação monolítica (código de aplicação + *kernel*)

```
main()
{
  init_system();
  /* para cada tarefa */
  t= atributos;
  create_task (t);
  start_all();
  while(1)
  {
    /* background */
  }
}
```

```
task_n()
{
  /* código da tarefa
  SEM ciclo infinito */
}
```

# OReK – Object Oriented Real Time Kernel

<http://www.ieeta.pt/~arnaldo/projects/OReK/OReK.htm>

- Baseado no ReTMiK:
  - *ticks*,
  - arquiteturas x86
  - Código das tarefas acíclico
  - Escalonador inserido no kernel
  - IPC por variáveis globais
- Orientado a objetos (C++)
- Operações sobre grupos
- Gera aplicação monolítica (código de aplicação+*kernel*)

```
main()
{
  /* criação dos objectos tarefa */
  AppliTask task0;
  CORKKernel::Initialize();
  /* para cada tarefa */
  task0.Create (atributos);
  CORKKernel::StartAllTasks();
  while(1 ou x tempo)
  {
    /* background */
  }
  CORKKernel::ShutDown();
}
```

```
class AppliTask : public
                    CORKTask
{
public:
  AppliTask ()
    { /* inicialização */
    }
public:
  virtual void Main ()
    { /* código da tarefa
      SEM ciclo infinito */
    }
protected:
  /* variáveis do objecto */
}
```

# RTKPIC – Real-Time Kernel for PIC18

Baseado no ReTMiK

- Baseado em ticks
- Corre em processadores PIC18FXXX
- Código das tarefas é cíclico
- Escalonador inserido no código do kernel (necessário recompilar para alterar)
- Controlo de preempção
- IPC por variáveis globais
- Gera aplicação monolítica (código de aplicação + *kernel*)

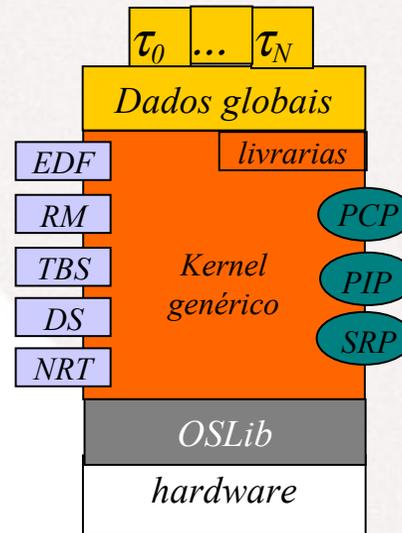
```
main()  
{  
    create_system(... );  
    /* para cada tarefa */  
    create_task (...);  
  
    config_system();  
    release_system( );  
    while(1)  
    {  
        /* background */  
    }  
}
```

```
task_n()  
{  
    task_init();  
  
    while(1) {  
        /* código da  
        tarefa*/  
    }  
}
```

# SHaRK – Soft and Hard Real Time Kernel

<http://shark.sssup.it/>

- Objetivo principal de obter flexibilidade na modificação das políticas de escalonamento quer para CPU quer para recursos partilhados
- POSIX (parcial/ compatível)
- Arquiteturas x86 >= i386
- Código das tarefas cíclico
- Variados métodos de IPC
- Conceito de *Task Model* (HRT, SRT, NRT, per, aper) e de *Scheduling Module*
- Policiamento, controlo de admissão
- Gera aplicação monolítica



```
InitFile
(declaração dos módulos)

tarefa __init__
(inicializações e chama main())
```

```
int main () {
/* outras inicializações */
/* definir tarefas de acordo com
o Task Model */
    task_create ();
    task_activate ();
/* pode terminar ou esperar para
fechar o sistema */
    while (keyb_getchar() != ESC);
    sys_end ();
}
```

```
void * TaskBody (void *arg) {
/* inicialização */
while (cond) {
    /* código da tarefa */
    (...)
    task_endcycle ();
/* termina com "cond" */
return my_val;
}
```

# RTAI – Real Time Application Interface for Linux

<http://www.rtai.org/>

- Objetivo de permitir usar o SO Linux em aplicações de tempo-real
- Usa o conceito de módulo carregado dinamicamente (acesso ao *kernel space*)
- POSIX (parcial/ compatível)
- Código das tarefas cíclico
- Variados métodos de IPC, quer entre tarefas TR quer com o *user-space* (fifo, shm, mailbox\)

```
/* módulo RT_test */  
  
void TaskBody (void *arg){  
    /* inicialização */  
    while (1) {  
        /* código da tarefa */  
        rtf_put(FIFO,...);  
        rt_task_wait_period( );  
    }  
  
int init_module(void){  
    rt_set_periodic_mode( );  
    rt_task_init( );  
    rtf_create(FIFO, ...);  
    start_rt_timer( );  
    rt_make_periodic( );  
}  
  
void cleanup_module(void){  
    stop_rt_timer( );  
    rt_task_delete( );  
    rtf_destroy(FIFO);  
}
```

```
/* comandos na shell */  
  
insmod rtai  
(...)  
insmod rt_test  
linux_test_proc  
rmmod rt_test  
(...)  
rmmod rtai
```

```
/* processo linux_test_proc */  
  
int main ( ){  
    fifo=open("/dev/rtf0",...);  
    while (cond){  
        read(fifo, ...);  
        printf( ... );  
    }  
}
```

# Resumo da Aula 3

- Os **estados de execução** de uma tarefa
  - diagrama de transição de estados
- A **arquitectura genérica** de um kernel de tempo-real
- Os **componentes básicos** de um kernel de tempo-real, estruturas de dados e funções
- Exemplos: ReTMiK, OReK, RTKPIC18, SHaRK e RTAI