CMSIS-RTOS API v2
e RTOS Keil RTX 5 (continuação)

Prof. Hugo Vieira Neto

2020/2

Políticas de Escalonamento

- Escalonamento Round Robin
  - RTX_Config.h: `#define OS_ROBIN_ENABLE 1`
  - Todas as tarefas com a **mesma prioridade**
  - Tarefas executam por um período máximo de tempo (Tq), até entrarem no estado bloqueado ou cederem o controle voluntariamente (yield)
  - `#define OS_TICK_FREQ 1000` → T = 1/f = 1ms
  - `#define OS_ROBIN_TIMEOUT 5` → Tq = 5T = 5ms

Políticas de Escalonamento

- Escalonamento Round Robin Preemptivo
  - RTX_Config.h: `#define OS_ROBIN_ENABLE 1`
  - Tarefas com prioridades **diferentes**
  - Tarefas executam por um período máximo de tempo (Tq), até entrarem no estado bloqueado ou cederem o controle voluntariamente (yield)
  - Tarefas mais prioritárias que entram no estado pronto causam preempção em tarefas menos prioritárias

Políticas de Escalonamento

- Escalonamento Preemptivo
  - RTX_Config.h: `#define OS_ROBIN_ENABLE 0`
  - Tarefas com prioridades **diferentes**
  - Tarefas executam sem limite de tempo até entrarem no estado bloqueado ou cederem o controle voluntariamente (yield)
  - Tarefas mais prioritárias que entram no estado pronto causam preempção em tarefas menos prioritárias

Políticas de Escalonamento

- Escalonamento Colaborativo
  - RTX_Config.h: `#define OS_ROBIN_ENABLE 0`
  - Todas as tarefas com a **mesma prioridade**
  - Tarefas executam sem limite de tempo até entrarem no estado bloqueado ou cederem o controle voluntariamente (yield)

Escalonamento Preemptivo

- Serviços do RTOS (SVC_Handler)
- Troca de contexto ocorre para:
  - Uma tarefa bloqueada de prioridade mais alta
    - Notificação de evento pela tarefa corrente ou por ISR
    - Liberação de semáforo pela tarefa corrente ou por ISR
    - Liberação de mutex pela tarefa corrente
    - Escrita de mensagem em uma fila não-cheia
    - Leitura de mensagem de uma fila cheia
  - Uma tarefa pronta de prioridade menor ou igual
    - Redução da prioridade ou bloqueio da tarefa corrente

Objetos do RTOS

- Tarefas
- Mutexes
- Semáforos
- Filas de mensagens
- Sinalizadores de evento
- Temporizadores de software
- Pools de memória (alocação dinâmica)

Número de Objetos da Aplicação

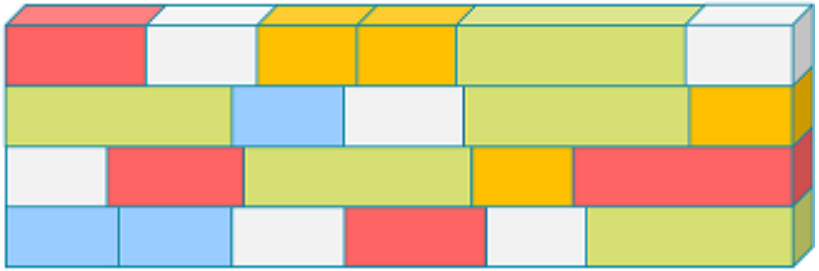
- RTX_Config.h:
  - OS_THREAD_NUM
  - OS_MUTEX_NUM
  - OS_SEMAPHORE_NUM
  - OS_MSGQUEUE_NUM
  - OS_EVFLAGS_NUM
  - OS_TIMER_NUM
  - OS_MEMPOOL_NUM

Alocação de Memória

- Objetos do RTX precisam de memória RAM
  - Criação: **`os`*`Object`*`New`**`()`
  - Deleção: **`os`*`Object`*`Delete`**`()`
- Opções de alocação:
  - Dinâmica:
    - Pool de memória global para todos os tipos de objeto
    - Pools de memória específicos para cada tipo de objeto
  - Estática

Pool de Memória Global

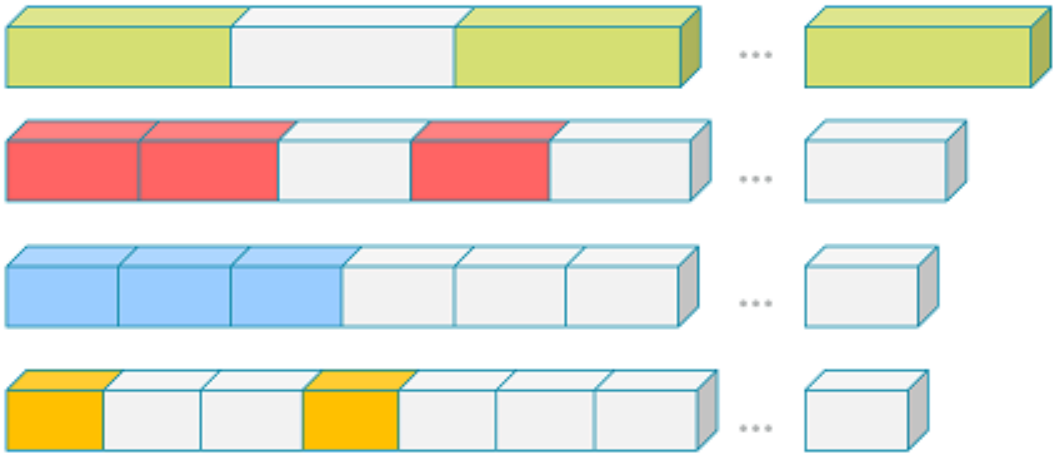
- Pool único para todos os tipos de objeto (default)
  - Sucessivas alocações e liberações de objetos de diferentes tamanhos causam fragmentação da memória e seu eventual esgotamento



- RTX_Config.h:
  - Sistema: `OS_DYNAMIC_MEM_SIZE` (default: 4096)

Pools de Memória Específicos

- Pools individuais para cada tipo de objeto (tamanhos fixos)
  - Não há fragmentação da memória
  - Alocação e liberação determinísticas

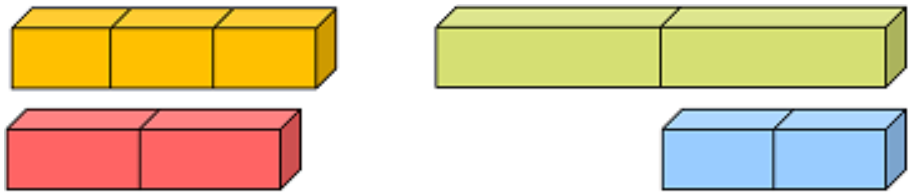


Pools de Memória Específicos

- RTX_Config.h:
  - Tarefas: `OS_THREAD_OBJ_MEM`
  - Mutexes: `OS_MUTEX_OBJ_MEM`
  - Semáforos: `OS_SEMAPHORE_OBJ_MEM`
  - Filas de mensagens: `OS_MSGQUEUE_OBJ_MEM`
  - Sinalizadores de eventos: `OS_EVFLAGS_OBJ_MEM`
  - Temporizadores: `OS_TIMER_OBJ_MEM`
  - Pools de memória: `OS_MEMPOOL_OBJ_MEM`

Alocação Estática

- Ocorre em tempo de compilação, evitando a possibilidade de esgotamento de memória do sistema em tempo de execução – requisito típico em sistemas críticos em segurança (*safety critical*)
  - Configuração realizada **manualmente**



Uso da Pilha pelas Tarefas

- Cada tarefa possui sua própria pilha local para armazenamento do contexto, variáveis locais e endereços de retorno de funções aninhadas
- Contexto de tarefa:
  - Cortex-M sem FPU (M0/M3): 64 bytes
  - Cortex-M com FPU (M4F/M7F): 200 bytes
- O alinhamento deve ser de 8 bytes (AAPCS)
- RTX_Config.h:
  - Thread Stack Size: `OS_STACK_SIZE`

Privilégio de Acesso das Tarefas

- Privilegiado:
  - Acesso a todos os recursos e instruções
- Não privilegiado:
  - Sem acesso ao SysTick, NVIC ou SCB
  - Possível acesso limitado a memória e periféricos
  - Sem acesso à instrução CPS
  - Acesso limitado às instruções MSR e MRS
- RTX_Config.h:
  - Processor Mode: `OS_PRIVILEGE_MODE`

Keil RTX 5

- É um RTOS que possui **muitos** recursos
- Estudaremos as suas funcionalidades **básicas**
  - Detalhamento técnico: documentação oficial
- As suas demais funcionalidades poderão ser estudadas com base na documentação oficial

Informações do Kernel

```
osStatus_t osKernelGetInfo(osVersion_t
*version, char *id_buf, uint32_t id_size)
```

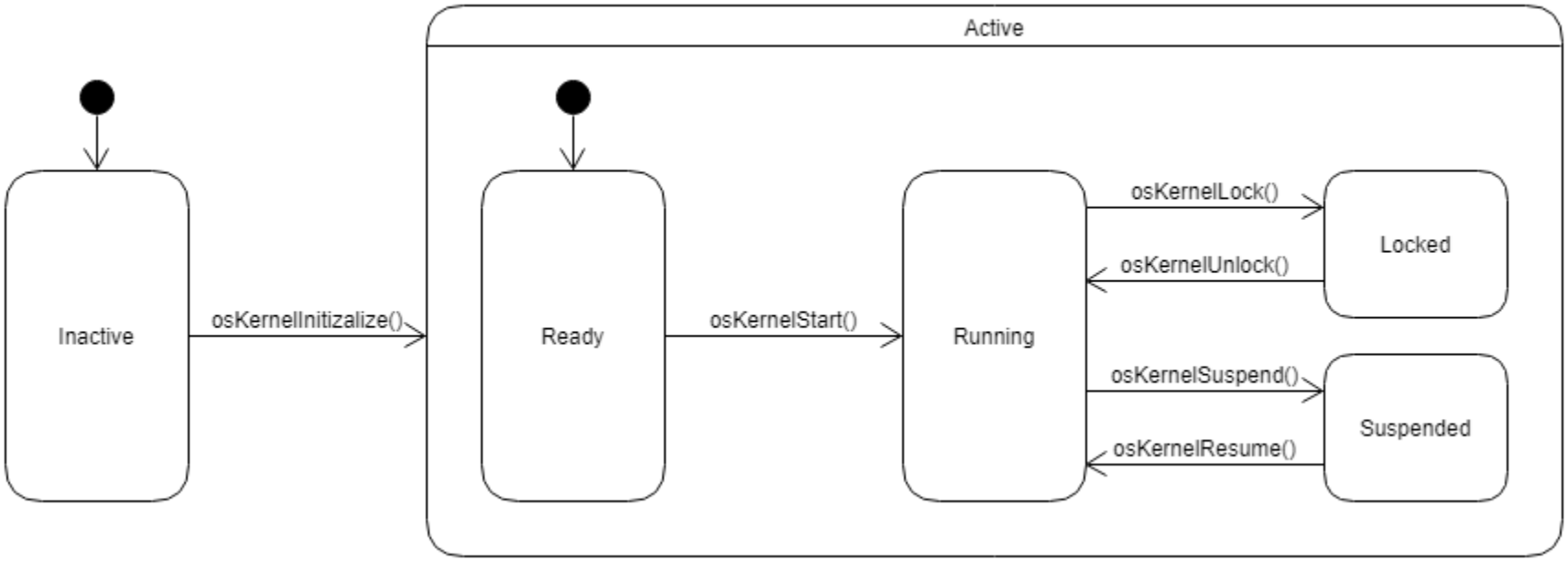
- Obtém informações sobre o kernel do RTOS (versão da implementação e versão da API)*

```
osKernelState_t osKernelGetState(void)
```

- Obtém o estado atual do kernel do RTOS (inativo, pronto, executando, bloqueado, suspenso, erro)*

- Observação: nem todas as funções do RTOS podem ser chamadas a partir de ISRs*

Estados do Kernel



Controle do Kernel

`osStatus_t` **`osKernelInitialize`**`(void)`

- Inicializa o kernel do RTOS
- Antes da sua chamada só é permitido chamar as funções **`osKernelGetInfo`** e **`osKernelGetState`**

`osStatus_t` **`osKernelStart`**`(void)`

- Inicia o escalonamento de tarefas
- Antes da sua chamada só é permitido chamar as funções **`osKernelGetInfo`**, **`osKernelGetState`** e de criação de objetos (**`os`*`Object`*`New`**)

Informações do Kernel

`uint32_t` **`osKernelGetTickFreq`**`(void)`

- Obtém a frequência dos tiques do sistema em Hz*

`uint32_t` **`osKernelGetTickCount`**`(void)`

- Obtém a contagem atual dos tiques do sistema*
- Uso típico com **`osDelayUntil`**

- Observação: podem ser chamadas a partir de ISRs*

Estrutura Básica de Código
(Alocação Dinâmica de Objetos)

```
#include "device.h"    // CMSIS-Core
#include "cmsis_os2.h" // CMSIS-RTOS

// declarações de variáveis globais
// declarações de tarefas e funções

void main(void){
  ... // inicializações de hardware
  osKernelInitialize(); // inicialização do kernel
  ... // criações de objetos (tarefas, mutexes, etc)
  osKernelStart(); // início da execução das tarefas
  while(1); // execução nunca deve chegar aqui!
} // main
```

Informações das Tarefas

```
uint32_t osThreadEnumerate(osThreadId_t
*thread_array, uint32_t array_items)
```

- Enumera as tarefas ativas

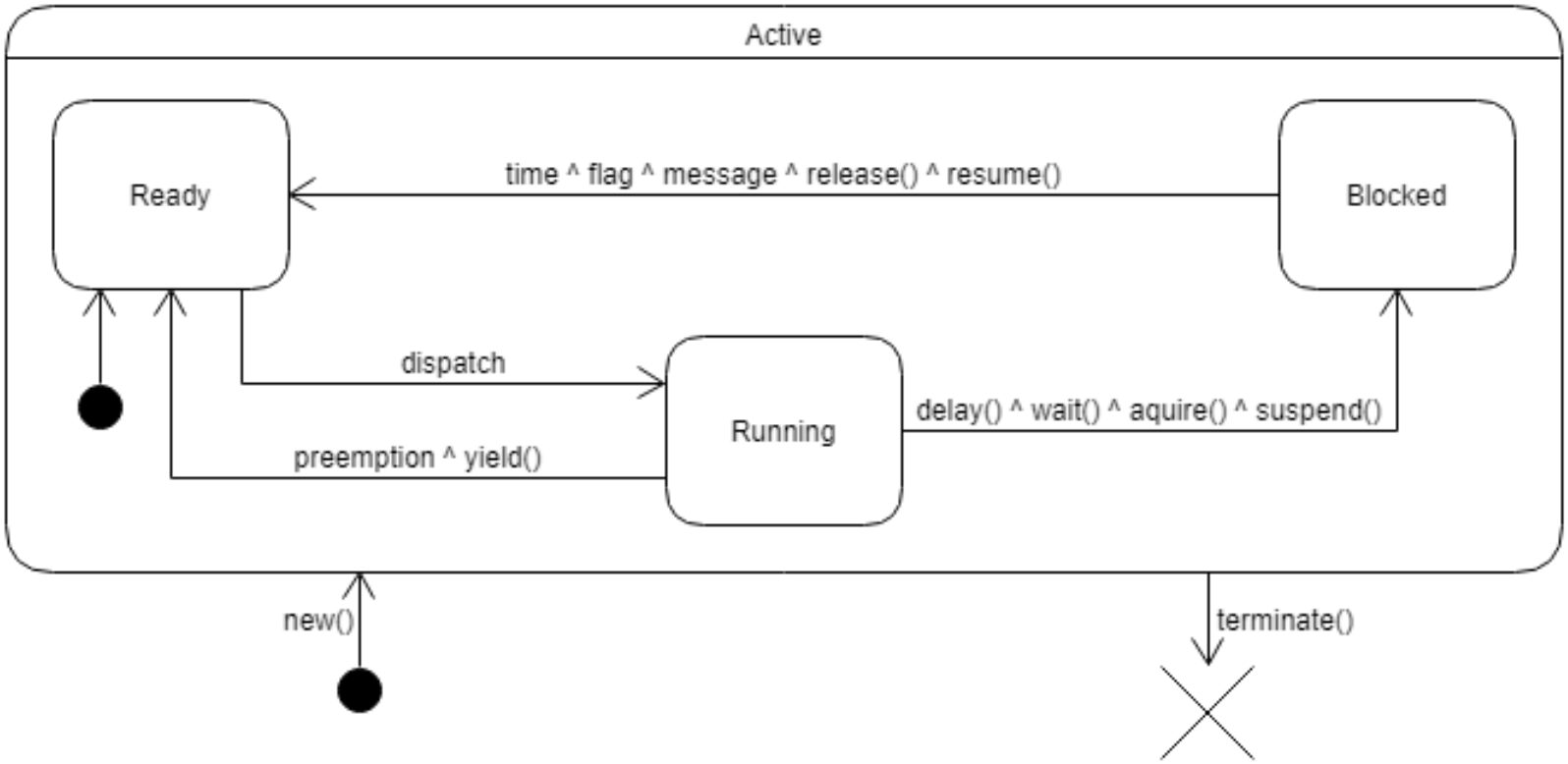
```
uint32_t osThreadGetCount(void)
```

- Obtém o número de tarefas ativas

```
osThreadState_t osThreadGetState
(osThreadId_t thread_id)
```

- Obtém o estado corrente da tarefa especificada

Estados das Tarefas



Gerenciamento de Tarefas

`osThreadId_t` **`osThreadNew`**`(osThreadFunc_t func, void *arg, const osThreadAttr_t *attr)`

- Adiciona uma tarefa às tarefas ativas

`osThreadId_t` **`osThreadGetId`**`(void)`

- Obtém o identificador da tarefa corrente*

`osStatus_t` **`osThreadYield`**`(void)`

- Passa o controle à próxima tarefa pronta

`osStatus_t` **`osThreadTerminate`**`(osThreadId_t thread_id)`

- Remove a tarefa especificada das tarefas ativas

Estrutura Básica de uma Tarefa

```
void thread(void *arg){
  ... // inicializações da tarefa
  while(1){
    ... // atividades da tarefa
  } // while
} // thread
```

- É possível passar um único argumento por referência à tarefa [**`osThreadNew`**`()`]
  - Útil para definir comportamentos diferentes para diferentes instâncias de uma mesma tarefa

Funções Genéricas de Espera

`osStatus_t` **`osDelay`**`(uint32_t ticks)`

- Espera por um período de tempo (relativo) especificado em tiques do sistema

`osStatus_t` **`osDelayUntil`**`(uint32_t ticks)`

- Espera até um instante de tempo (absoluto) especificado em tiques do sistema
- Uso típico com **`osKernelGetTickCount`**`()`

- Observação: funções de espera **não** podem ser chamadas a partir de rotinas de atendimento de interrupção (ISR)

Tempo de Espera

- Um tempo de espera (timeout) é passado como argumento a várias das funções do RTX para aguardar pela disponibilidade de um recurso (bloqueio)
- O valor desse argumento constitui o limite **superior** do tempo de espera e depende de quanto tempo passou desde a ocorrência do último tique

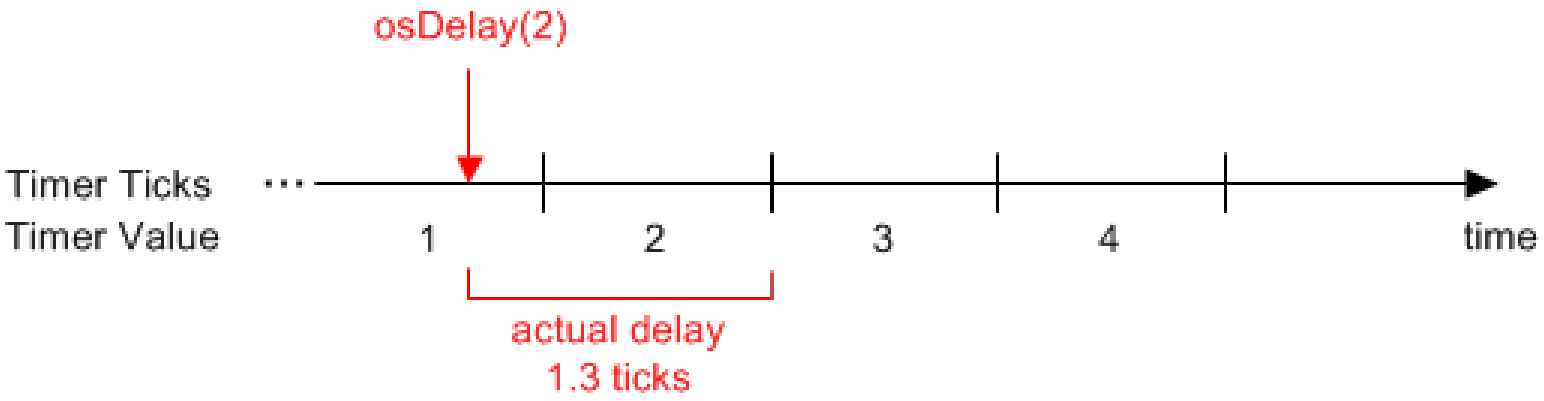
Tempo de Espera

- Tempo de espera = `0`:
  - A função retorna imediatamente, mesmo quando não há recurso disponível

- Tempo de espera = `1`:
  - O sistema aguarda a ocorrência do próximo tique, que pode corresponder a um período de tempo bem curto

Tempo de Espera

- Tempo de espera = `n`:
  - O tempo de espera **real** varia entre `n-1` e `n` tiques



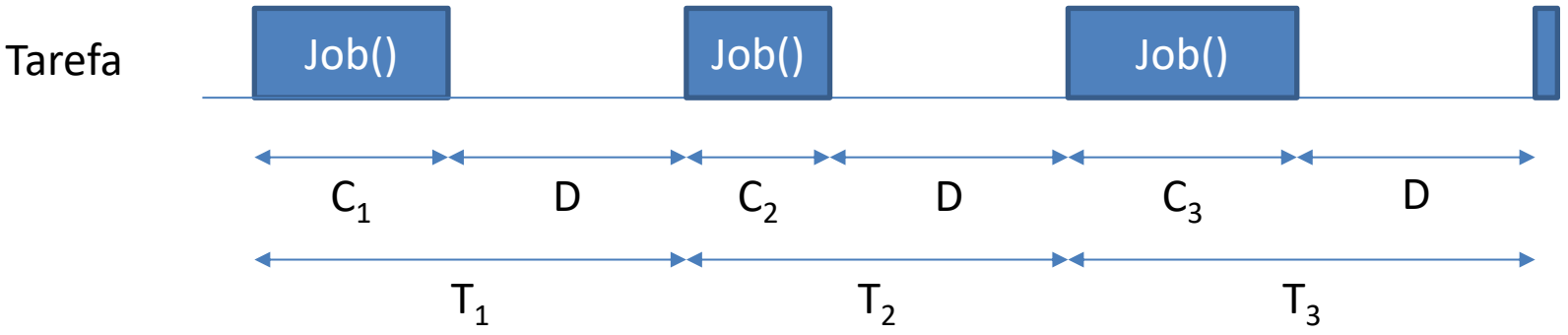
- Tempo de espera = `osWaitForever`:
  - O sistema aguarda indefinidamente (para sempre)

Tarefa Periódica com osDelay

```
void thread(void *arg){
  Init(); // inicializações da tarefa
  while(1){
    Job(); // atividades a serem realizadas (serviço)
    osDelay(100); // atraso entre serviços = 100
  } // while
} // thread
```

- O uso de **`osDelay`**`()` resulta em períodos de ativação diferentes, conforme o tempo de computação do serviço (função genérica `Job`)

Diagrama de Gantt (osDelay)



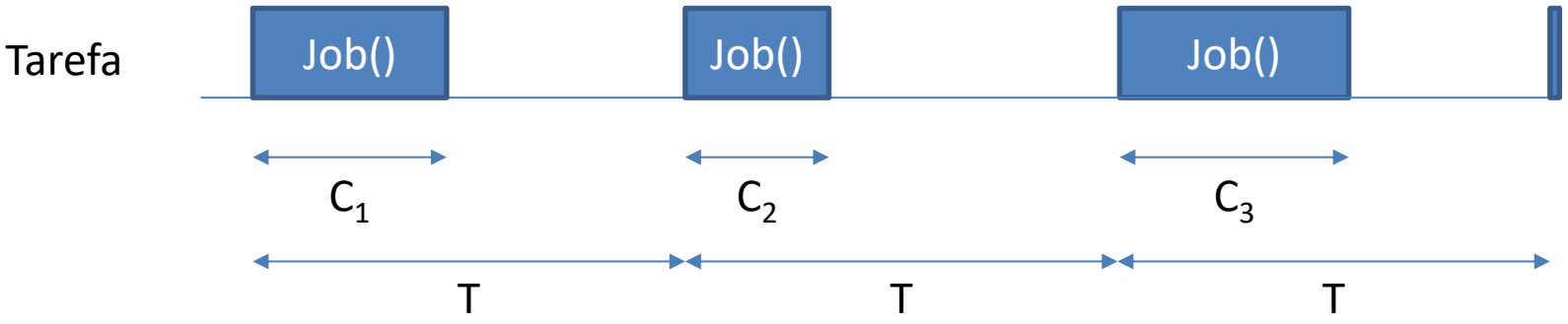
- Tempo de computação (C_i): duração de cada serviço da tarefa
- Tempo de atraso (D): duração do bloqueio entre serviços
- Período de ativação (T_i): intervalo entre cada início de serviço
- Jitter (J): variação máxima do período de ativação (máx ΔT)

Tarefa Periódica com osDelayUntil

```
void thread(void *arg){
  Init(); // inicializações da tarefa
  uint32_t tick;
  while(1){
    tick = osKernelGetTickCount();
    Job();// atividades a serem realizadas (serviço)
    osDelayUntil(tick + 100); // período de ativação = 100
  } // while
} // thread
```

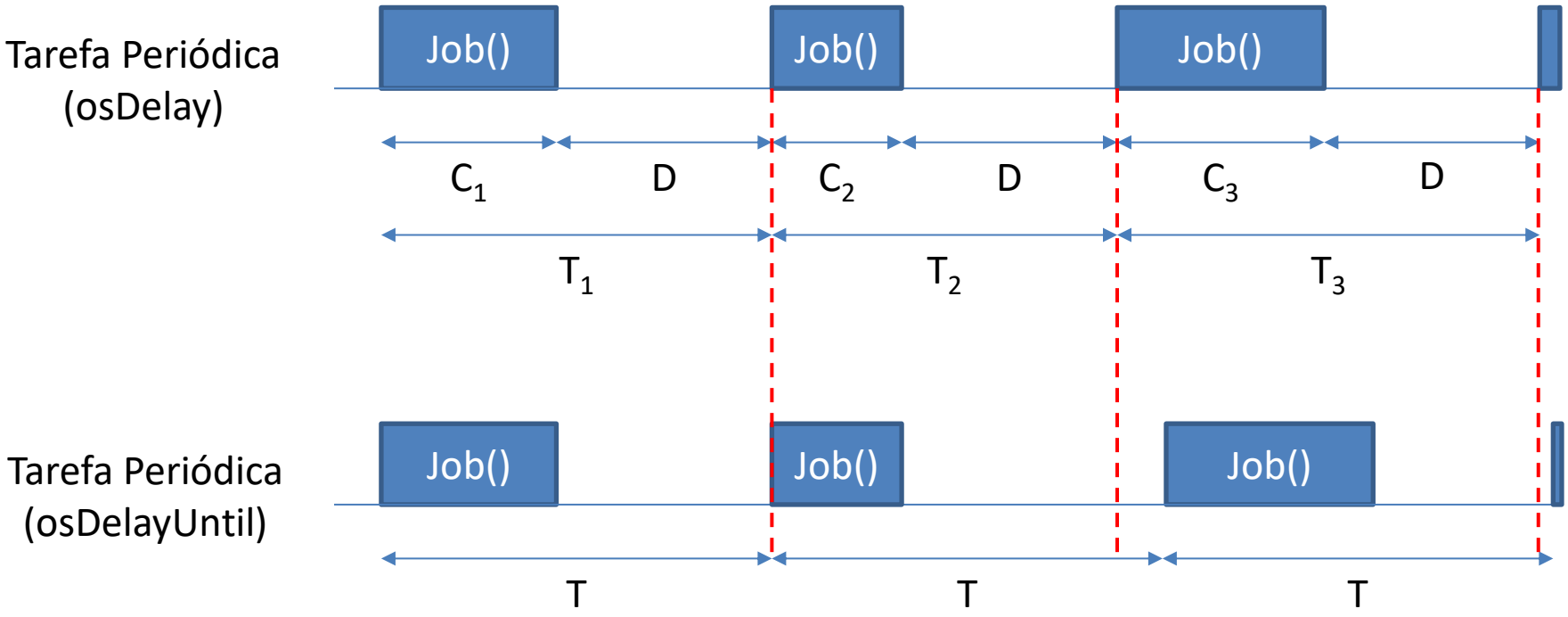
- O uso de **`osDelayUntil`**`()` resulta em período de ativação independente do tempo de computação do serviço (função genérica `Job`)

Diagrama de Gantt (osDelayUntil)



- Tempo de computação (C_i): duração de cada serviço da tarefa
- Período de ativação (T): intervalo entre cada início de serviço
- O jitter nesse caso é minimizado

Comparação dos Diagramas de Gantt



Exemplo de Uso (Tarefas)

```
#include "system_tm4c1294.h" // CMSIS-Core
#include "driverleds.h" // device drivers
#include "cmsis_os2.h" // CMSIS-RTOS

osThreadId_t thread1_id, thread2_id;

void thread1(void *arg){
  uint8_t state = 0;

  while(1){
    state ^= LED1;
    LEDWrite(LED1, state);
    osDelay(100);
  } // while
} // thread1
```

Exemplo de Uso (Tarefas)

```
void thread2(void *arg){
  uint8_t state = 0;
  uint32_t tick;

  while(1){
    tick = osKernelGetTickCount();

    state ^= LED2;
    LEDWrite(LED2, state);

    osDelayUntil(tick + 100);
  } // while
} // thread2
```

Exemplo de Uso (Tarefas)

```
void main(void){
  SystemInit();
  LEDInit(LED1 | LED2);

  osKernelInitialize(); // Initialize CMSIS-RTOS

  thread1_id = osThreadNew(thread1, NULL, NULL);
  thread2_id = osThreadNew(thread2, NULL, NULL);

  if(osKernelGetState() == osKernelReady)
    osKernelStart();

  while(1);
} // main
```

Diagrama de Objetos (Arquitetura)

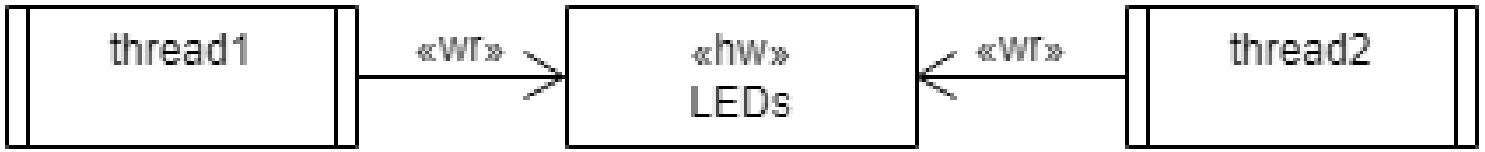


Diagrama de Atividades (Tarefa 1)

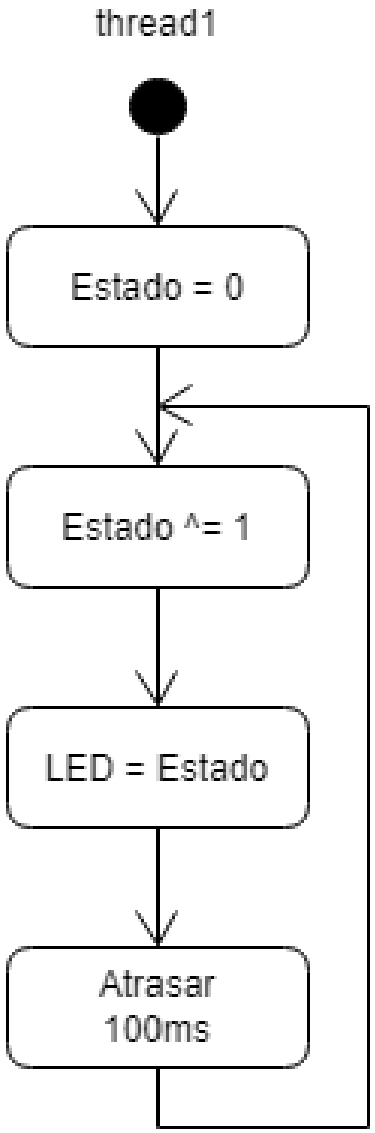
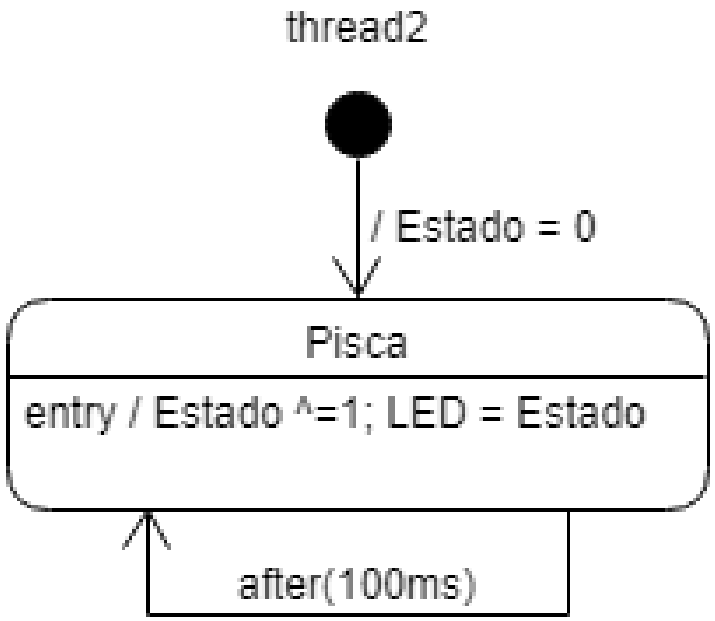


Diagrama de Estados (Tarefa 2)



Exercício 2

1. Abra o projeto “tarefas” a partir da área de trabalho “EK-TM4C1294_RTOS_IAR8”.
2. Localize o arquivo `RTX_Config.h` na lista de dependências do arquivo `rtx_lib.c` e analise-o. Quais são as configurações para:
   - `OS_TICK_FREQ`?
   - `OS_THREAD_NUM`?
   - `OS_THREAD_DEF_STACK_NUM`?

Exercício 2

3. Altere a configuração `OS_TICK_FREQ` para 500.
   – Qual é o efeito dessa alteração na execução do programa do projeto “tarefas”?
4. Retorne a configuração `OS_TICK_FREQ` para o seu valor original.
   – Quais são os períodos de ativação das tarefas `thread1` e `thread2`?

Exercício 2

5. Modifique o código-fonte no arquivo `tarefas.c` para que duas instâncias de uma *mesma* tarefa sejam usadas para acionar o LED D1 e o LED D2 do kit de desenvolvimento.
   – Para que isso seja possível, será necessário passar a informação sobre qual LED deve ser acionado como parâmetro na criação de cada uma das duas instâncias da tarefa.

Exercício 2

6. Modifique novamente o código no arquivo `tarefas.c` para que seja possível informar às instâncias da tarefa o LED a ser acionado e também o seu período de ativação.
   - Para que isso seja possível, será necessário passar um ponteiro para uma struct como parâmetro na criação de cada instância da tarefa. Essa struct deverá conter dois elementos que identifiquem o LED a ser acionado e o tempo de acionamento.

Exercício 2

7. Crie quatro instâncias da nova tarefa, uma para acionar cada LED do kit, com períodos diferentes: LED D1 = 200ms, LED D2 = 300ms, LED D3 = 500ms, LED D4 = 700ms.

Para pensar...

- O sistema de tarefas para acionamento individual dos LEDs, *como encontra-se implementado*, tem um problema em potencial que somente não se manifesta porque as tarefas são bastante curtas e possuem períodos de ativação iguais.
- Qual é esse problema? Como garantir que ele seja evitado mesmo se as tarefas forem mais longas e tiverem períodos de ativação diferentes?