Sistemas embarcados

Técnicas de projeto de software de sistemas embarcados

Projeto com laço infinito

O projeto de software com laço infinito é baseado no conceito *background / foreground.*

Um laço infinito executa determinadas tarefas através da chamada de módulos (implementados no formato de **funções**) para realizar as operações desejadas.

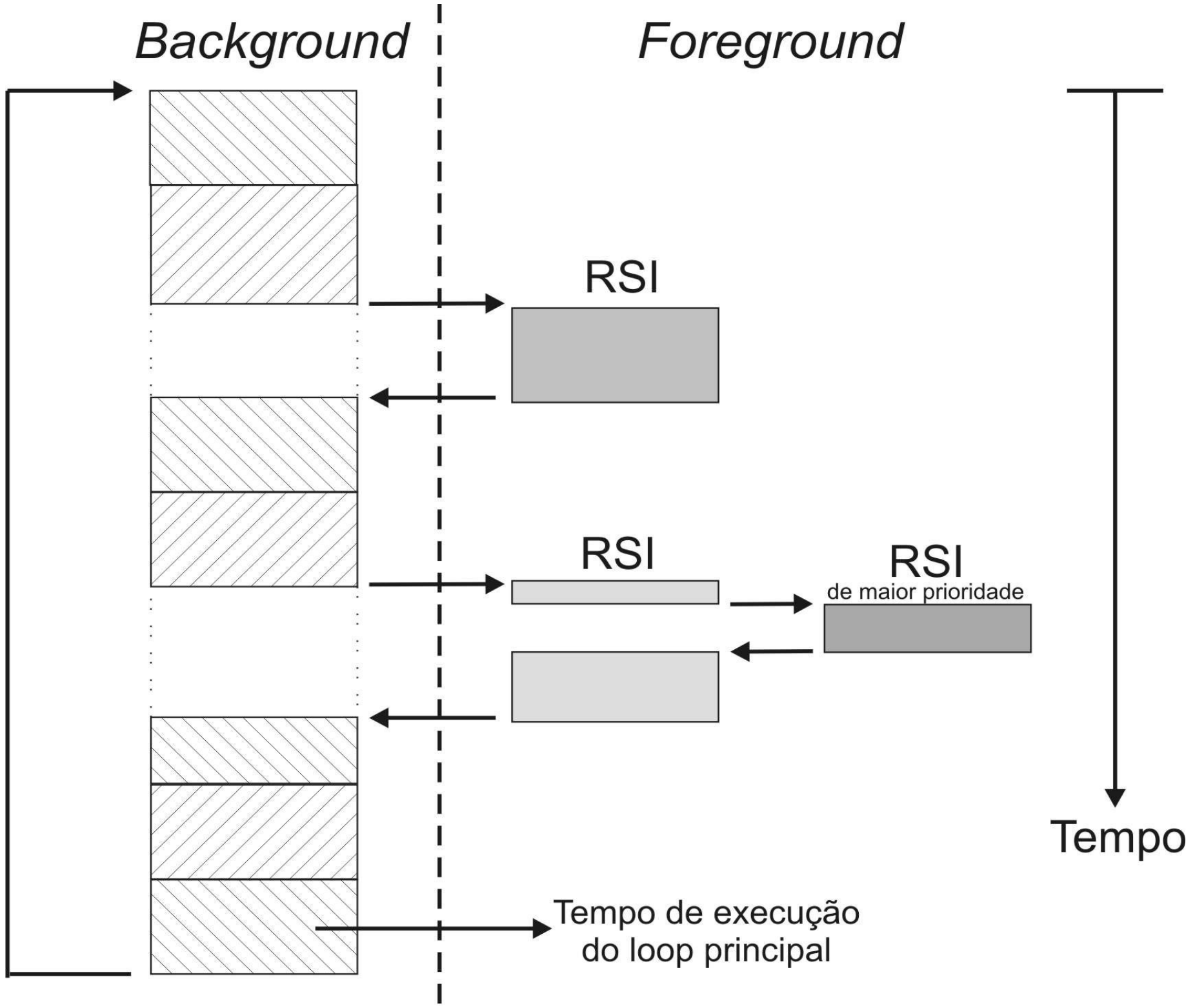
Este laço infinito é conhecido por *background*.

Projeto com laço infinito

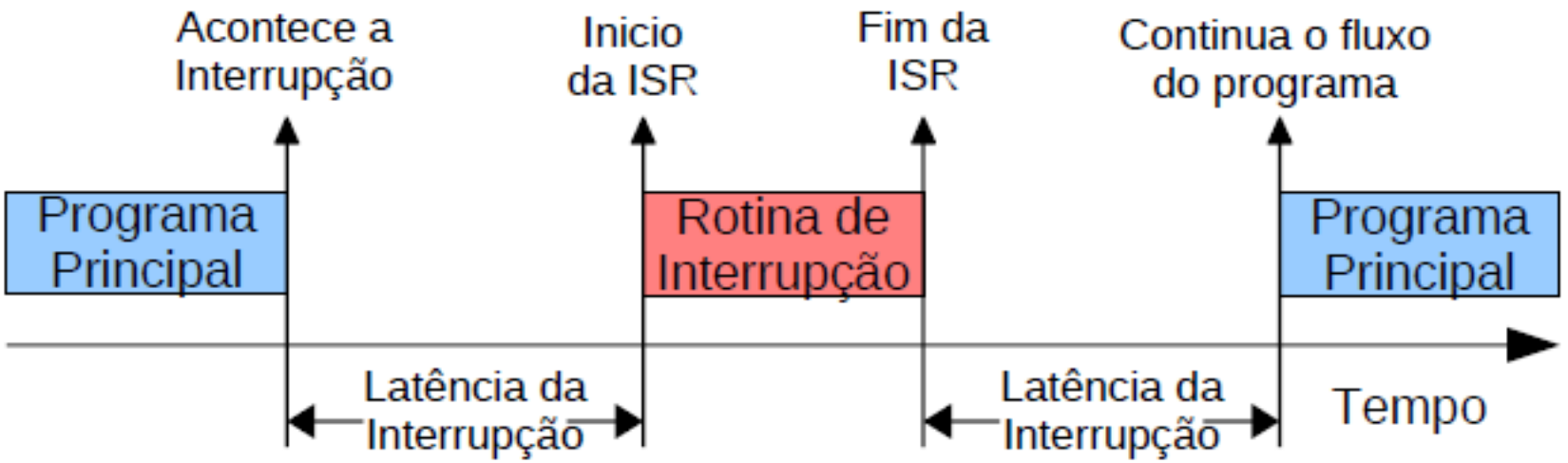
Rotinas de serviço de interrupção (RSI) são utilizadas para o tratamento de eventos assíncronos.

Quando o sistema encontra-se nestas rotinas, define-se que está em *foreground*.

As posições de *foreground* e *background* são também conhecidas por “Nível de Interrupções” e “Nível de Tarefas”, respectivamente.


Background
Foreground
RSI
RSI
RSI
de maior prioridade
Tempo
Tempo de execução
do loop principal

Projeto com laço infinito



Projeto com laço infinito

```
for ( ; ; )
{
    if(display == 1)
    {
      display = 0;
      atualiza_display();
    }

    if(teclado == 1)
    {
      teclado = 0;
      le_teclado();
    }

    if(serial == 1)
    {
      serial = 0;
      le_serial();
    }
}
```

Nesta técnica, geralmente, são utilizadas variáveis *flags* para indicar quais tarefas devem ser executadas.

Estas *flags* são alteradas pelas rotinas de interrupção.

Projeto com laço infinito

Prós

- ✓ Simplicidade.
- ✓ Atendimento rápido de interrupções.
- ✓ Facilidade para projeto de baixo consumo. Processador permanece em estado “dormente” e “acorda” para executar as interrupções.
- ✓ Consumo de memória RAM é facilmente gerenciável.

Contras

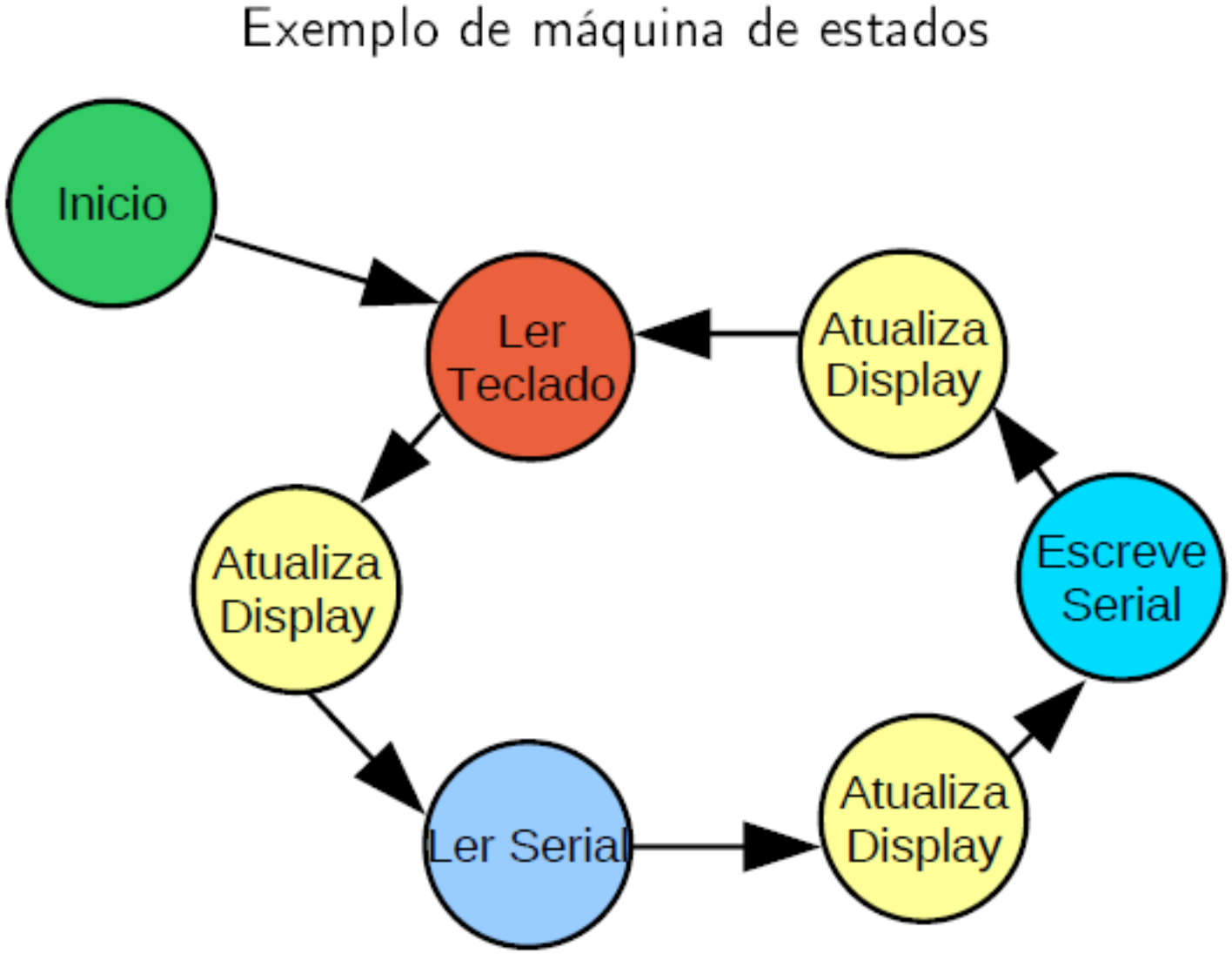
- ✗ Tempo de execução das rotinas não-determinístico
- ✗ Difícil manutenção conforme aumenta a complexidade do software.
- ✗ Não é facilmente escalável.
- ✗ Não recomendado para sistemas de tempo real, por ser difícil de garantir prazos para execução das tarefas.

Projeto com máquinas de estados finitos (FSM)

Uma técnica mais organizada de se projetar o software de um sistema embarcado é a utilização do modelo de uma máquina de estados finitos (FSM).

Nesta técnica, o sistema é separado em estados em que ele pode estar e cada estado executa uma das tarefas apenas.

Projeto com máquinas de estados finitos (FSM)



```
void main(void){
  char slot;
  //Inicializa...
  for(;;){  //inicio do loop infinito
  //********************* top-slot ***********************
  //************ início da maquina de estado ************
    switch(slot){
      case 0:
        LeTeclado();         slot = 1; break;
      case 1:
        AtualizaDisplay(); slot = 2; break;
      case 2:
        RecebeSerial();     slot = 3; break;
      case 3:
        AtualizaDisplay(); slot = 4; break;
      case 4:
        EnviaSerial();      slot = 5; break;
      case 5:
        AtualizaDisplay(); slot = 0; break;
      default:
        slot = 0; break;
    }
  //************* fim da maquina de estado **************
  //******************** bottom-slot ********************
  } //fim loop infinito (!?)
}
```

Implementação em C com switch-case

Projeto com FSM

Prós

- ✓ Simplicidade.
- ✓ Atendimento rápido de interrupções.
- ✓ Facilidade para projeto de baixo consumo. Processador permanece em estado “dormente” e “acorda” para executar as interrupções.
- ✓ Consumo de memória RAM é facilmente gerenciável.
- ✓ Mais escalável que o laço infinito, pois é mais fácil adicionar tarefas

Contras

- ✗ Tempo de execução das rotinas ainda é não-determinístico.
- ✗ Não recomendado para sistemas de tempo real, por ser difícil de garantir prazos para execução das tarefas.

Projeto com FSM temporizada

- Pode-se temporizar a FSM de forma a tornar o funcionamento do sistema mais previsível.

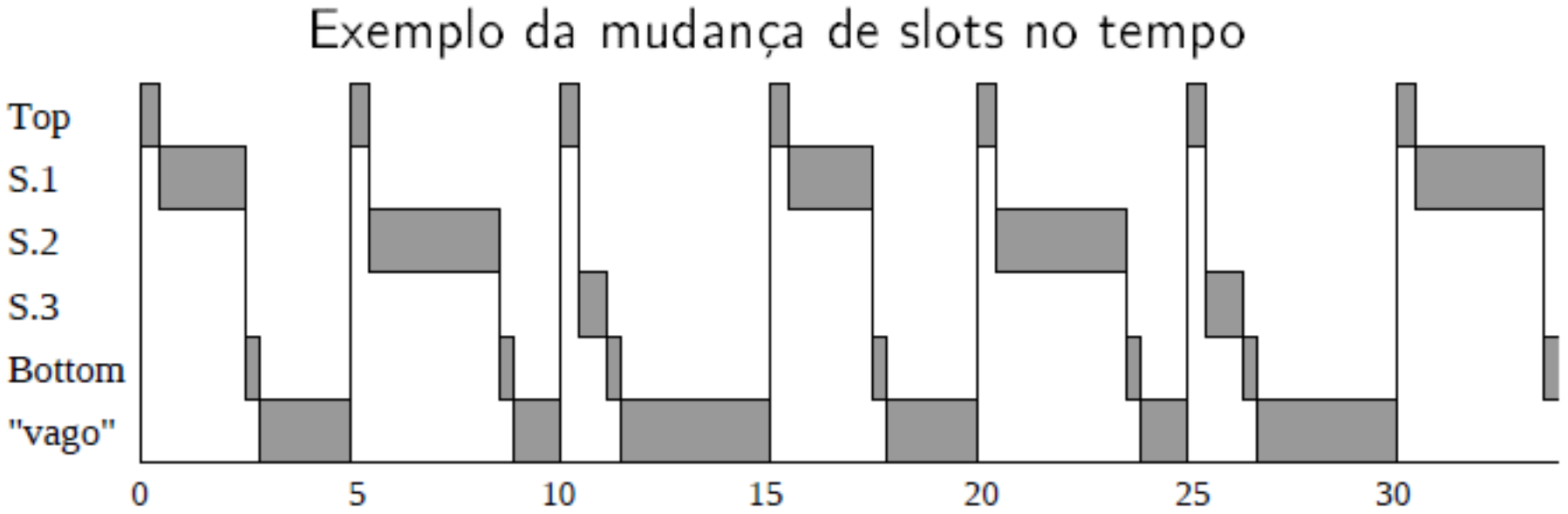
- Deste modo cada estado tem um tempo máximo constante de execução.

- Porém, ainda é difícil garantir que uma determinada tarefa será executa dentro de um determinado prazo.

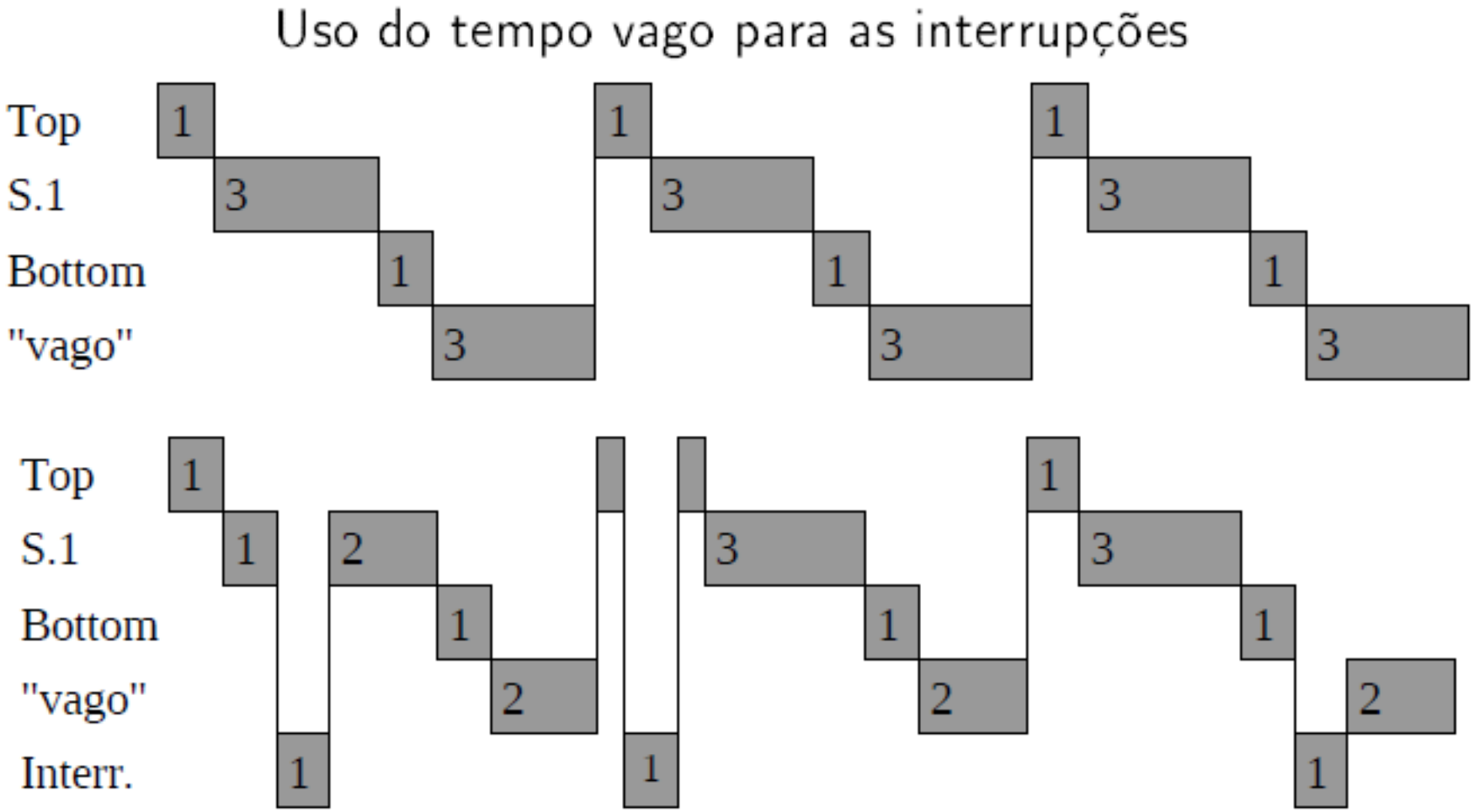
```
void main(void){
  char slot;
  //Inicializa...
  for(;;){  //inicio do loop infinito
  //********************* top-slot ************************
    ResetaTimer(5000);  //5 ms para cada slot
    AtualizaDisplay();
  //************ início da maquina de estado *************
    switch(slot){
      case 0:
        LeTeclado();        slot = 1; break;
      case 1:
        RecebeSerial();     slot = 2; break;
      case 2:
        EnviaSerial();      slot = 0; break;
      default:
        slot = 0; break;
    }
  //************ fim da maquina de estado ***************
  //******************* bottom-slot ***********************
    AguardaTimer();
  } //fim loop infinito (!?)
}
```

Implementação de FSM em C com switch-case e temporização

Projeto com FSM temporizada



Projeto com FSM temporizada



Projeto com sistema multitarefas

Embora nos projetos com **laço infinito** e **FSM**, o sistema realize diversas tarefas (como ler teclado, receber e enviar algum dado pela porta serial, atualizar *display*, etc...), apenas **uma tarefa pode ser executada de cada vez**!

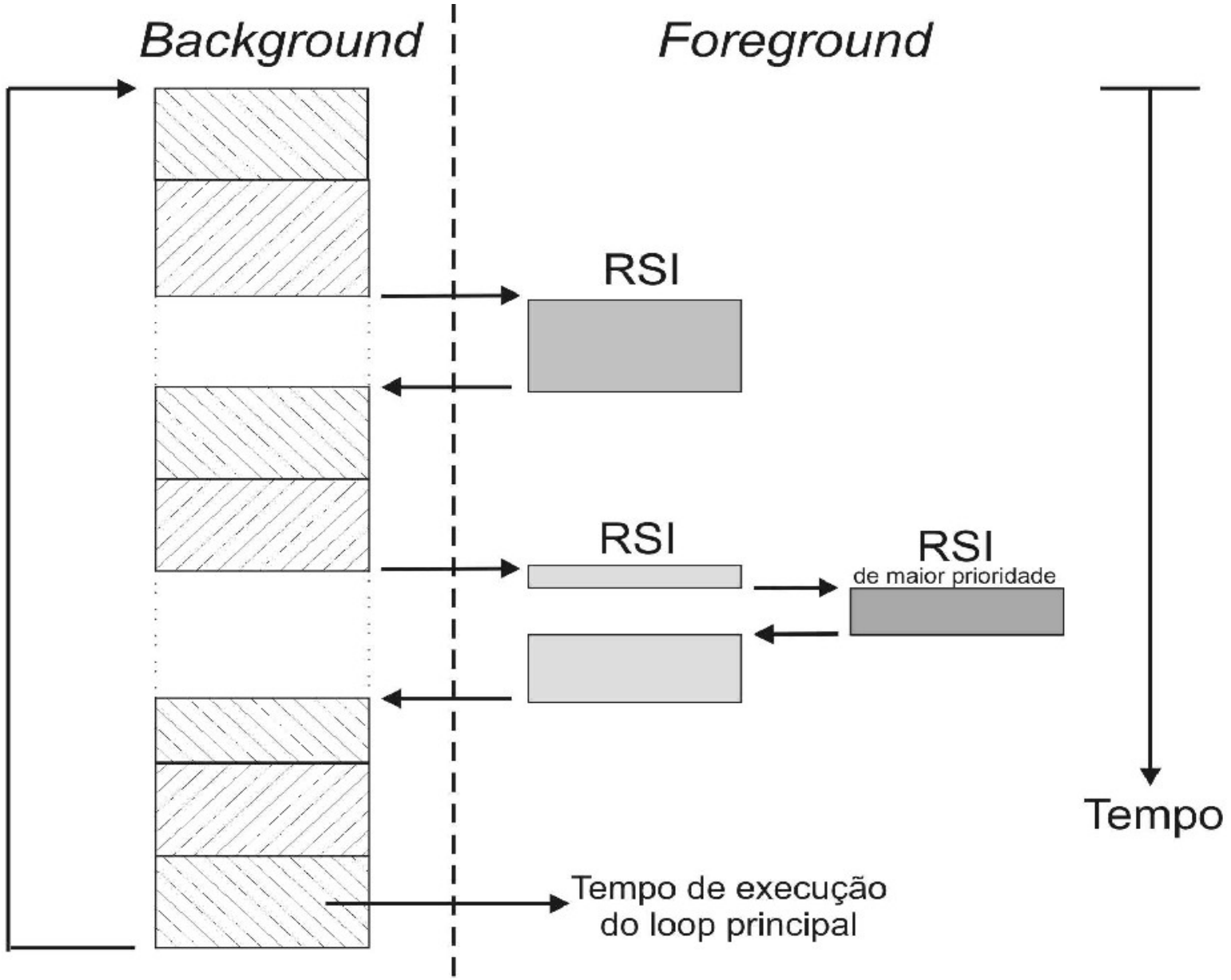
Isto é, se **uma tarefa está sendo executada**, o processador deverá terminá-la antes de poder **executar outra tarefa**.

Projeto com sistema multitarefas

Assim, mesmo que haja **uma tarefa mais prioritária** para ser executada pelo sistema embarcado, nas técnicas de projeto apresentadas não é possível **interromper uma tarefa** para executar outra em seu lugar. É como se houvesse apenas uma tarefa única!

Apenas rotinas de interrupções podem interromper uma tarefa.

Projeto com sistema multitarefas



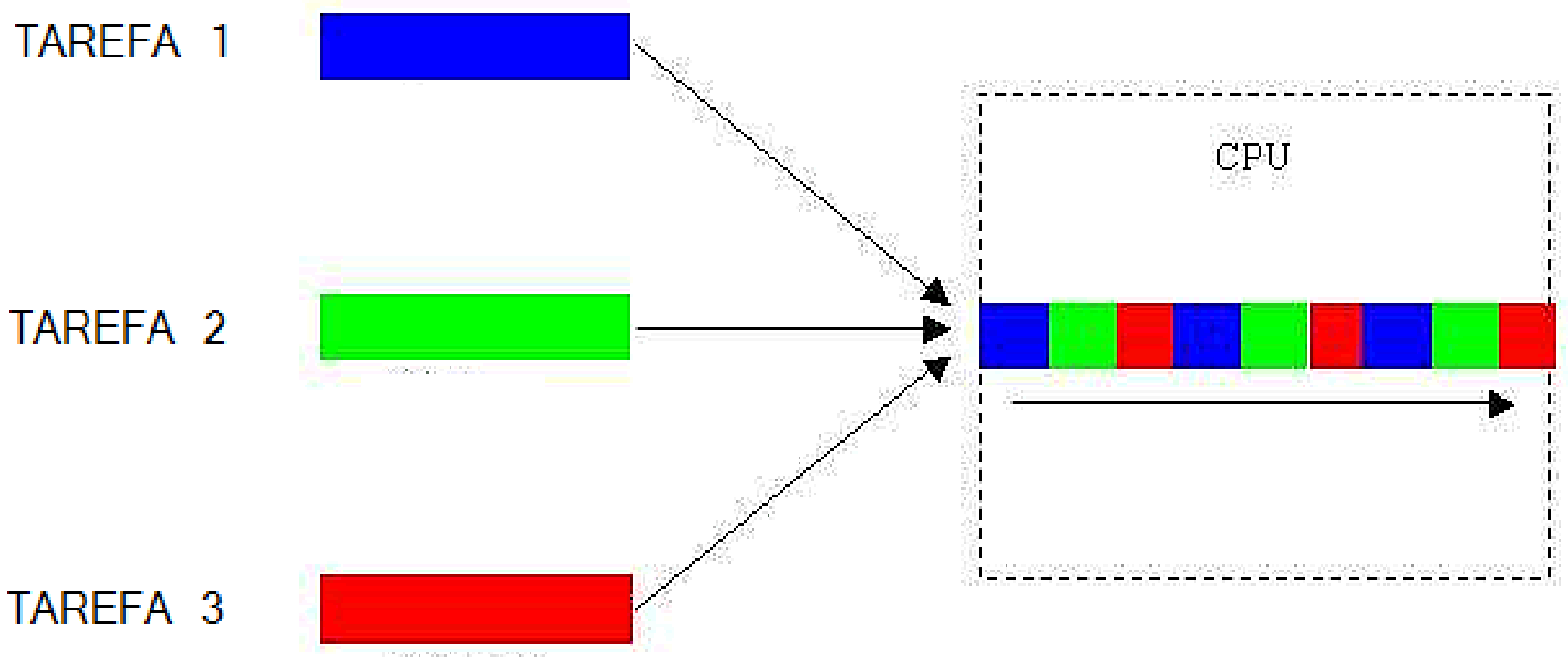
Projeto com sistema multitarefas

Interessantemente, em muitos microprocessadores e microcontroladores, rotinas de interrupções **de maior prioridade** podem interromper rotinas de interrupção de **menor prioridade**.

Se esta ideia for estendida para o **nível de tarefas** (isto é, tarefas puderem interromper outras tarefas), o sistema parecerá ter **muitas tarefas** sendo executadas concorrentemente!

Projeto com sistema multitarefas

Um sistema com este comportamento é denominado um **sistema multitarefas**!

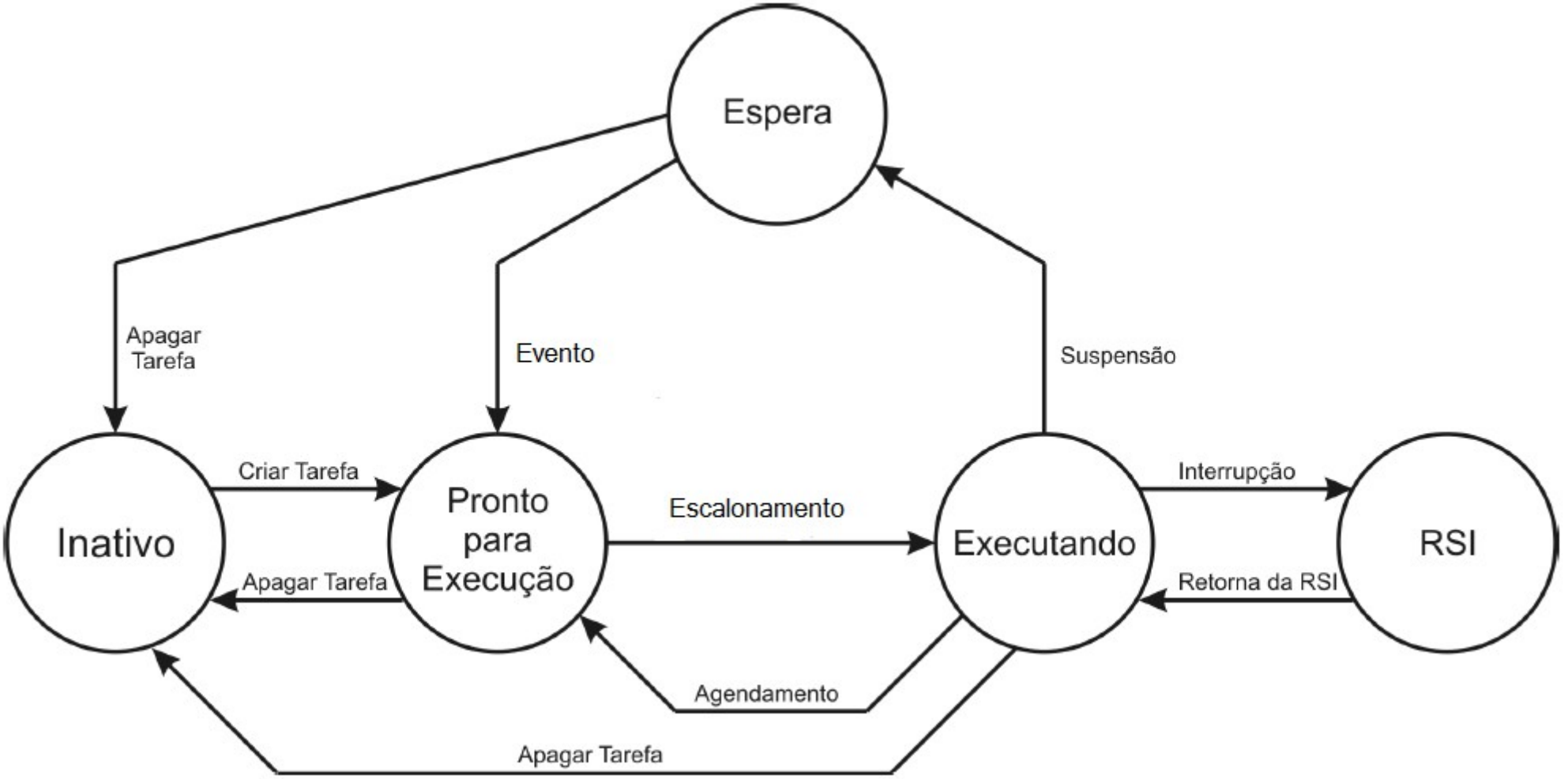


Projeto com sistema multitarefas

Em um projeto com sistema multitarefas, uma tarefa pode ser **suspensa ou interrompida** para que outra tarefa possa ser **executada em seu lugar!**

Projeto com sistema multitarefas

Uma tarefa pode passar por **diferentes estados** conforme o funcionamento do sistema.



Projeto com sistema multitarefas

Como uma tarefa pode passar por **vários estados**, é necessário armazenar estas informações na memória e gerenciá-las corretamente (**bloco de controle da tarefa**).

Projeto com sistema multitarefas

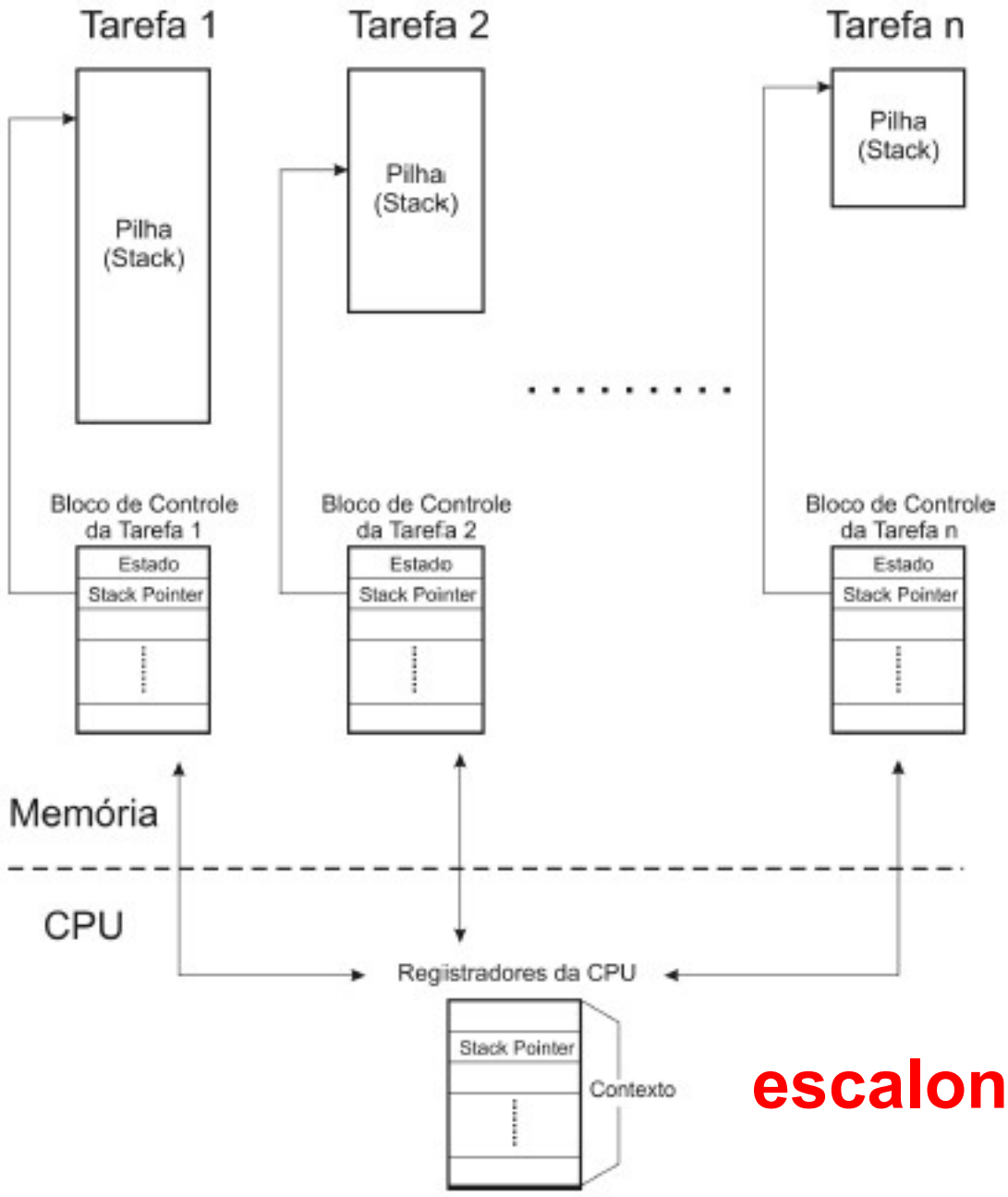
Se uma tarefa for **suspensa ou interrompida**, deve ser possível retornar para sua execução mais tarde sem ocorrer perda de dados. Isto é, as **variáveis, registradores** e **pilha** em uso devem ter seus valores recuperados (**contexto da tarefa**).

Cada tarefa deve ter uma **pilha exclusiva!**

Projeto com sistema multitarefas

Como é possível ter **mais de uma tarefa pronta para a execução** é necessário ter uma forma de decidir qual será a **próxima tarefa** a ser executada (**escalonamento de tarefas**).

Projeto com sistema multitarefas



contextos das tarefas

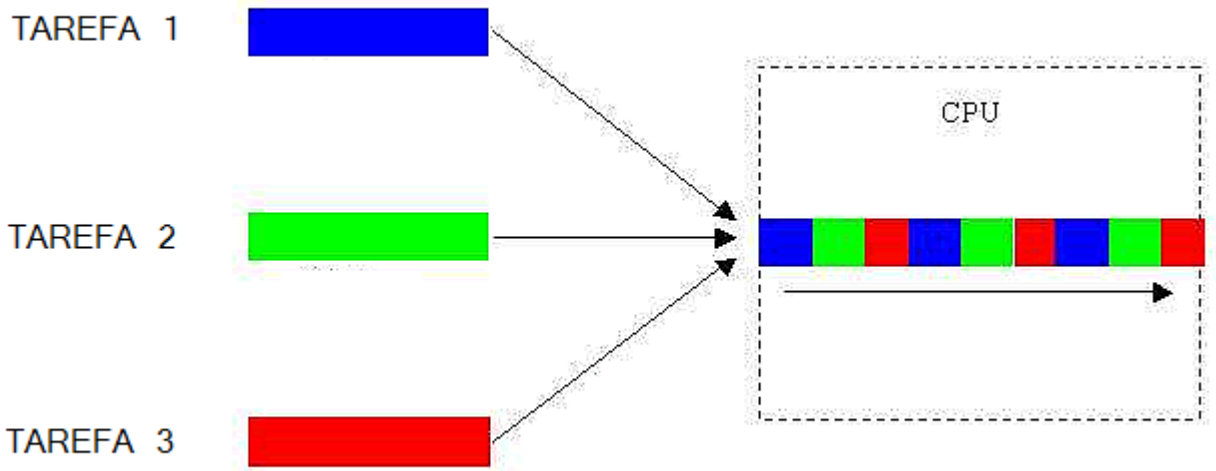
blocos de controle das tarefas

escalonamento das tarefas

Projeto com sistema multitarefas

Um **sistema multitarefas** opera como um sistema *foreground/background* com múltiplos *backgrounds.*

Um **sistema multitarefas** é parte mais básica de um **sistema operacional**.



Projeto com sistema multitarefas

Uma das vantagens de se utilizar um sistema multitarefas é a capacidade de **maximização do uso do processador**. Pois, toda vez que for necessário a uma tarefa esperar por um determinado evento, o processador pode ser usado por outra tarefa.

Porém, o que pode acontecer se **duas ou mais tarefas** tentarem utilizar **uma mesma função**?

Reentrância

- Uma **função reentrante** é uma função que pode ser utilizada por **mais de uma** tarefa sem a possibilidade de gerar **danificação nos dados**.

- Este tipo de função pode ser **interrompida a qualquer momento** e continuada mais tarde sem que ocorram perdas de dados.

- Para tanto, as funções reentrantes utilizam variáveis locais (registradores da CPU ou variáveis alocadas na pilha) ou variáveis globais protegidas.

Reentrância

A função abaixo é reentrante?

```
int teste;
void swap(int *x, int *y)
{
        teste = *x;
        *x = *y;
        *y = teste;
}
```

Reentrância

A função abaixo é reentrante?

```
void strcpy(char *dest, char *src)
{
        while(*src)
        {
                *dest++ = *src++;
        }
}
```

Reentrância

A função abaixo é reentrante?

```
void strcpy(char *dest, char *src)
{
        while(*src)
        {
                *dest++ = *src++;
        }
}
```

SIM!

Reentrância

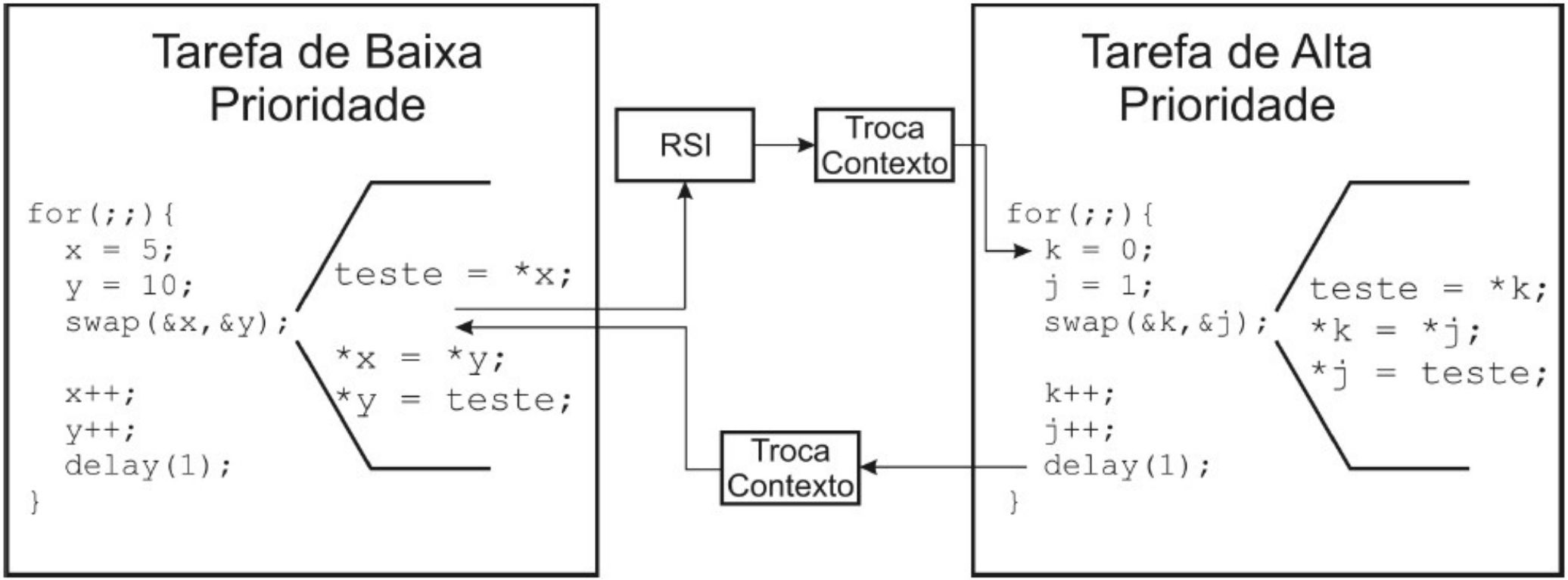
A função abaixo é reentrante?

```
int teste;
void swap(int *x, int *y)
{
        teste = *x;
        *x = *y;
        *y = teste;
}
```

NÃO!

Reentrância

A tarefa de baixa prioridade espera que “teste” valia 5, mas a outra tarefa (de alta prioridade) trocou o valor de “teste” para 0!



Reentrância

Modifique a função abaixo para torná-la uma função reentrante ?

```
int teste;
void swap(int *x, int *y)
{
        teste = *x;
        *x = *y;
        *y = teste;
}
```

Recursos compartilhados

- Um recurso é qualquer entidade utilizada por uma tarefa. Exemplos de recursos incluem dispositivos de entrada/saída, teclado, *display*, conversor A/D, etc, ou ainda, variáveis, estruturas, vetores, matrizes, etc...

- Os recursos que podem ser utilizados por mais de uma tarefa são chamados de **recursos compartilhados**.

- Cada tarefa deve obter **direito exclusivo de acesso** a um recurso compartilhado para evitar **corrupção de dados**.

Recursos compartilhados

- Exemplo: Imagine que duas tarefas compartilhem um *display* LCD alfanumérico.

- Se uma tarefa interromper o processo de escrita no *display* de uma outra tarefa, pode haver duas situações de corrupção de dados:

  - o protocolo de comunicação com o display poderá ser afetado.
  - a mensagem escrita na tela poderá conter erros, misturando as informações das duas tarefas.

Recursos compartilhados

- Exemplo: Imagine que duas tarefas compartilhem um *display* LCD alfanumérico.

- A tarefa 1 tenta escrever “Eu sou a tarefa 1!”
- A tarefa 2 tenta escrever “Eu sou a tarefa 2!”

- O resultado no display pode se parecer com: “Eu Eu sousou a a tartarefaefa 1! 2!”

Recursos compartilhados

- Por isso, ao projetar um sistema multitarefas, deve-se ter cuidado com o uso de recursos compartilhados.

- É importante assegurar que cada tarefa tenha **acesso exclusivo** ao recurso compartilhado.

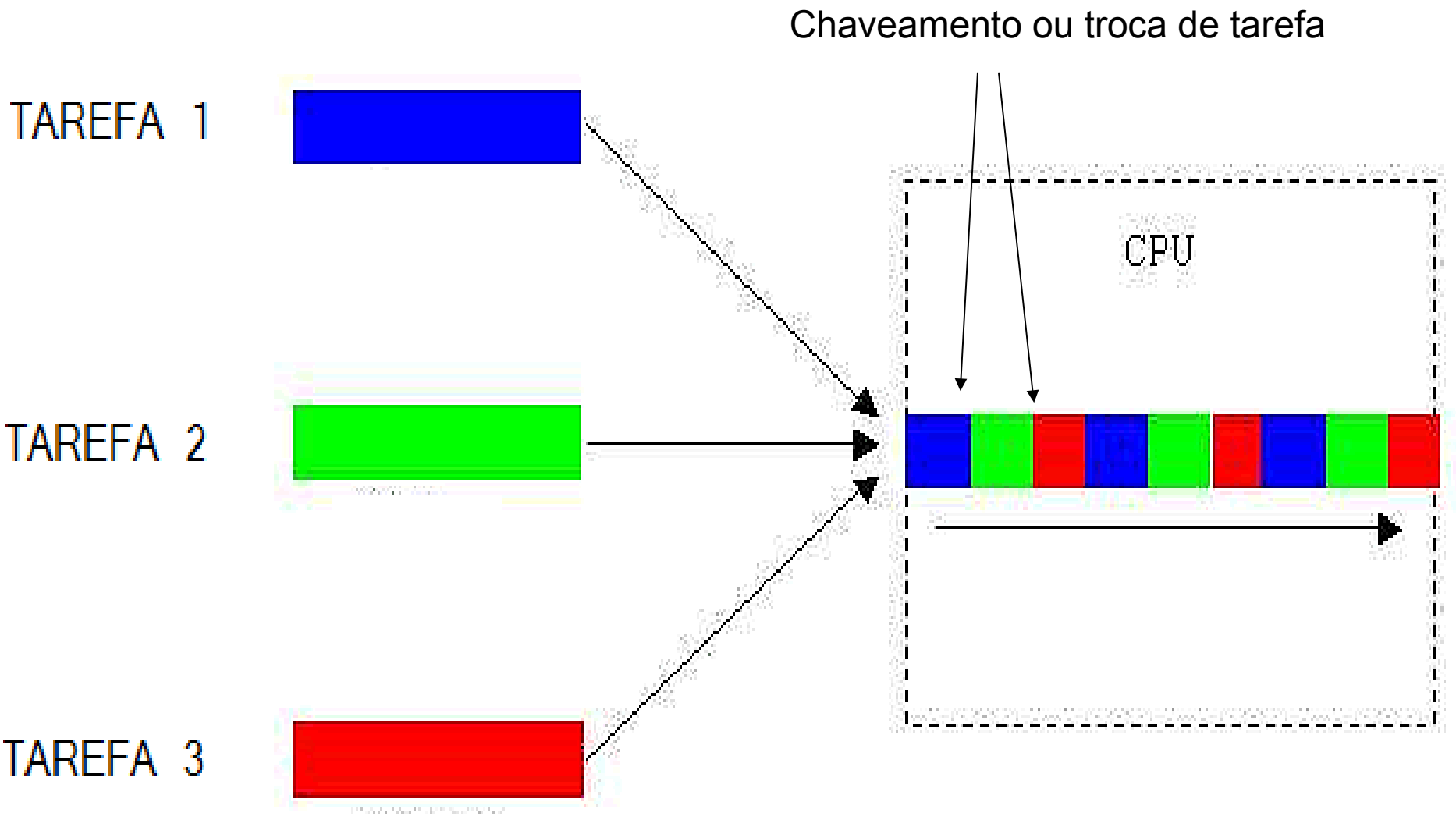
Gerenciamento de tarefas

- O **núcleo (ou kernel)** é a parte de um sistema multitarefa responsável pelo **gerenciamento de tarefas,** isto é, gerenciamento do tempo de utilização da CPU.

- Assim, o serviço mais importante fornecido pelo núcleo é o **chaveamento de tarefa** ou **chaveamento de contexto.**

- O **chaveamento de tarefa** ocorre quando uma tarefa passa a ser executada no lugar de outra tarefa.

Chaveamento de tarefas



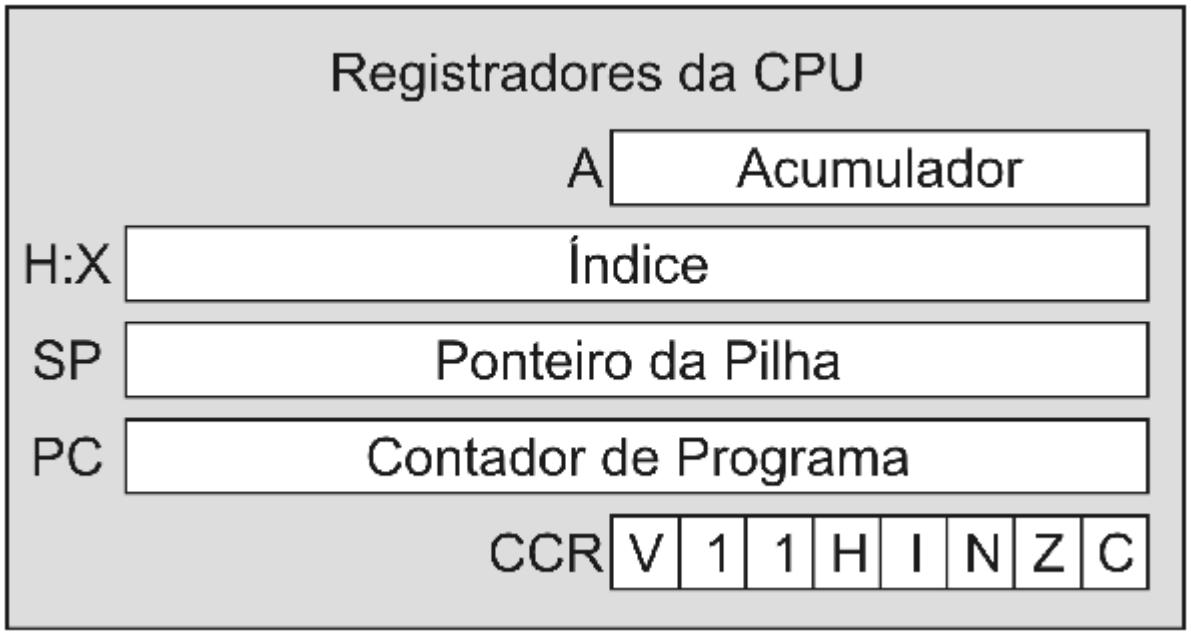
Chaveamento de tarefas

- Quando um núcleo multitarefa decide **executar uma tarefa diferente**, ele realiza o seguinte procedimento:
  - **salva o contexto** da tarefa em execução na área de armazenamento de contexto, ou seja, na pilha da tarefa.
  - **restaura o contexto** da nova tarefa a partir de sua pilha
  - passa a executar o código da nova tarefa.

Contexto da tarefa

- O contexto de uma tarefa é composto pelos **valores que estão armazenados nos registradores** do processador, incluindo-se o endereço da **próxima instrução (PC)** e o **endereço da pilha (SP)**, bem como todo o conteúdo armazenado pela tarefa na sua pilha.

Contexto do HCS08



Troca de contexto da tarefa

Código-exemplo para o chaveamento de contexto retirado do **BRTOS**

```
SelectedTask = OSSchedule();    // decide a próxima tarefa

if (currentTask != SelectedTask){  // próxima tarefa é diferente da atual ?
    OS_SAVE_CONTEXT();
    OS_SAVE_SP(); // salva contexto da tarefa atual

    ContextTask[currentTask].StackPoint = SPvalue;

    currentTask = SelectedTask; // troca a tarefa atual

    SPvalue = ContextTask[currentTask].StackPoint;

    OS_RESTORE_SP(); // restaura o contexto da próxima tarefa (agora atual)
    OS_RESTORE_CONTEXT();
}
```

Retirado de http://code.google.com/p/brtos/

Escalonamento de tarefas

- Nota-se que para realizar o chaveamento de contexto, o núcleo deve ter um procedimento para **decidir qual será a próxima tarefa a ser executada.**
- No código exemplo este procedimento é representado pela função OSSchedule();
- Caso a próxima tarefa selecionada seja diferente da atual, o núcleo realiza o procedimento para a troca de contexto.

Troca de contexto

- Nos sistemas multitarefas, há **dois possíveis eventos** que podem levar a execução do procedimento de troca de contexto:
  - A tarefa atual decide suspender-se e liberar o processador para outra tarefa, ou
  - Uma rotina de interrupção é usada para forçar a suspensão da tarefa atual e a troca por outra tarefa.

Troca de contexto

- Estas duas possibilidades para se fazer a troca de contexto permitem classificar os núcleos de sistemas multitarefas em:
  - COOPERATIVOS
  - PREEMPTIVOS

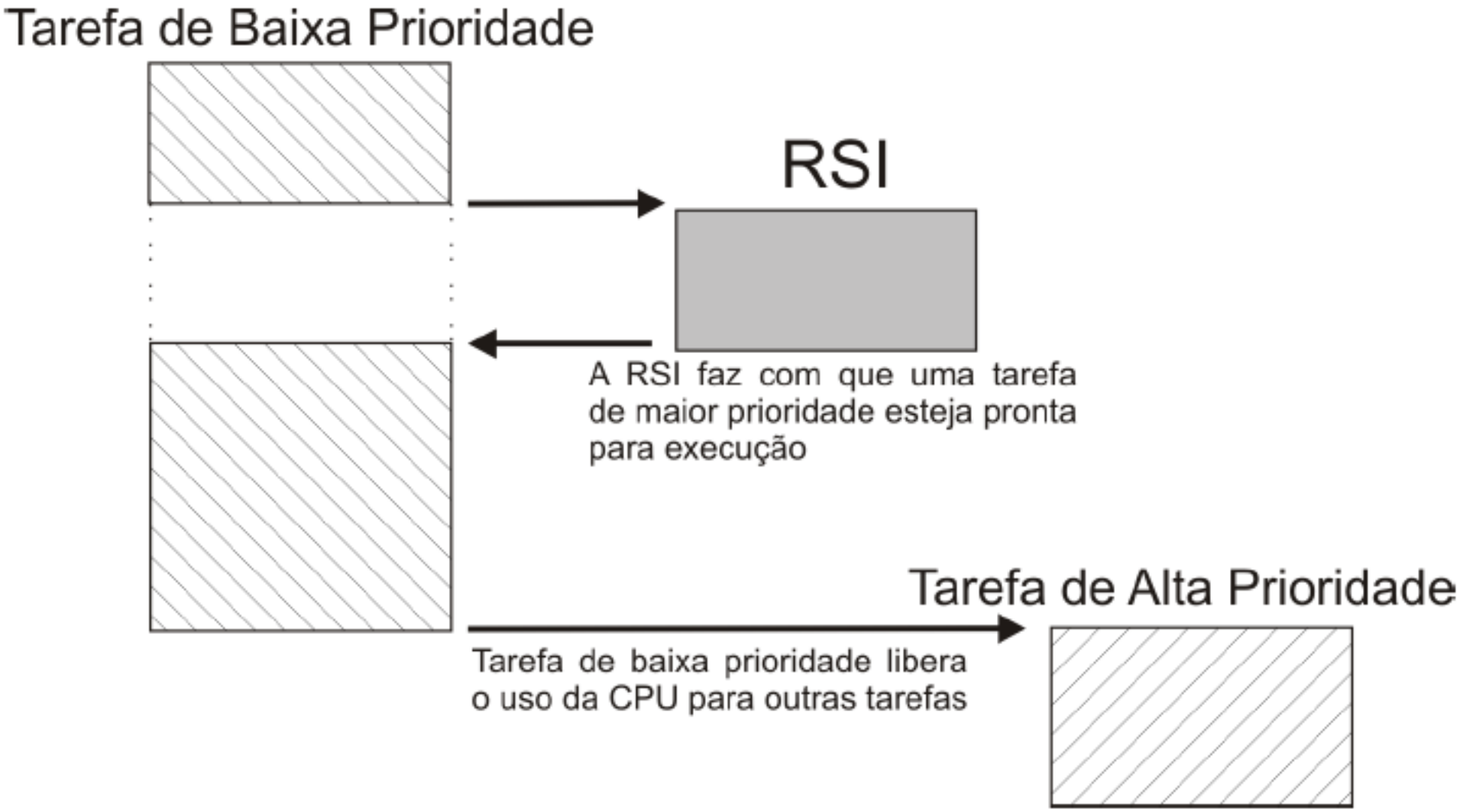
Núcleos cooperativos

- Nos **núcleos cooperativos**, as trocas de contexto ocorrem apenas quando a tarefa atual decide suspender-se e liberar o processador para outra tarefa.

- Assim, a tarefa atual suspende-se **através da chamada de serviço de troca de contexto** do núcleo, o qual pode ser implementada de duas formas:

  - Função
  - Interrupção por software.

Núcleos cooperativos



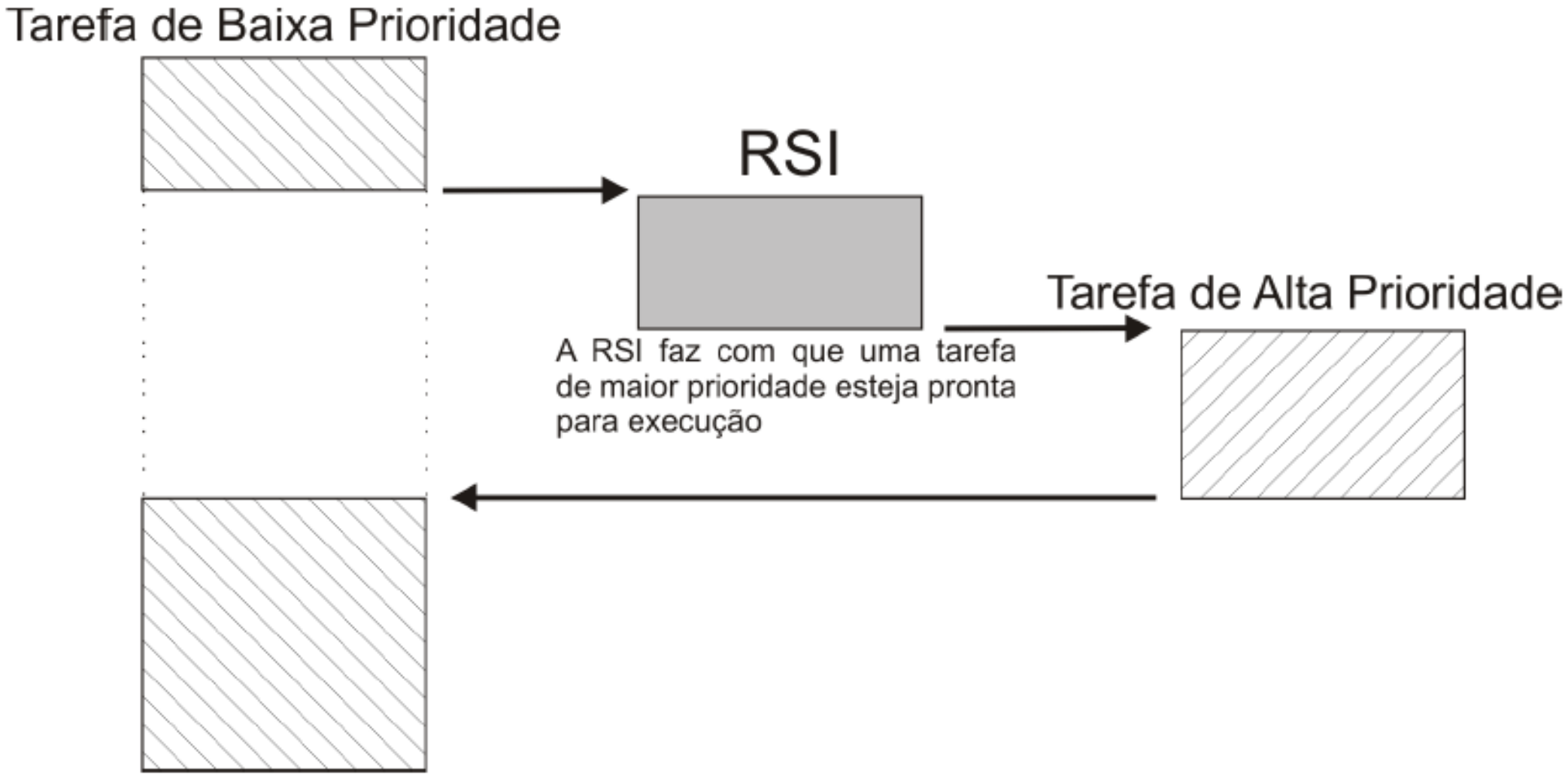
Núcleos preemptivos

- Nos **núcleos preemptivos**, as trocas de contexto podem ocorrer quando a tarefa atual decide suspender-se e liberar o processador para outra tarefa, ou através de uma interrupção que força a troca de contexto.

- Assim, a tarefa atual é suspensa e a interrupção deve realizar **a chamada de serviço de troca de contexto** do núcleo.

- Quando uma tarefa é “forçada a dar lugar” para outra, diz-se que ela foi **preemptida.**

Núcleos preemptivos



Exercício

- Crie um serviço de troca de contexto para um **núcleo cooperativo**, que realiza **apenas o salvamento do contexto atual**. O contexto deve ser salvo na pilha e o registrador de ponteiro da pilha (SP) deve ser salvo em uma variável global. Implemente-o utilizando:
  - uma função.
  - uma interrupção agendada por software.

Discuta as diferenças entre implementações.

Exercício

- Responda:
  - O que aconteceria se antes de retornar da função para o serviço de troca de contexto, o registrador do ponteiro de pilha (SP) fosse sobrescrito com outro valor?
  - Seria possível utilizar esta técnica para desviar a execução do código para uma outra função? O que seria necessário para fazê-lo? Implemente sua solução.

Bloco de controle de tarefas (TCB)

- Um sistema multitarefas pressupõe a existência simultânea de várias tarefas disputando o processador.
- Para facilitar a gerência da CPU, torna-se necessário representar tarefas através de estruturas de dados, normalmente conhecidas por blocos de controle de tarefas (*Task Control Block* – TCB).

Bloco de controle de tarefas (TCB)

- Um TCB é associado a uma tarefa no momento da criação da mesma.

- O TCB é uma estrutura de dados que é utilizada pelo sistema operacional para **manter o estado** de uma tarefa e permitir que a tarefa retorne a execução **exatamente no ponto** em que foi interrompida ou suspendeu-se.

Bloco de controle de tarefas (TCB)

- Os TCBs residem na memória de dados (RAM). As seguintes informações são normalmente armazenadas em um bloco de controle de tarefas:

  - Localização e tamanho na memória;
  - Estado relativo ao processador (pronta, executando, suspensa, etc);
  - Contexto da execução (valor do *stack pointer*);
  - Nome da tarefa (*string* contendo um nome fantasia para a tarefa ser identificada);
  - Prioridade.
  - Etc...

Bloco de controle de tarefas (TCB)

```
typedef struct ContextTag
{
    INT8S  Name[configMAX_TASK_NAME_LEN];
    INT16U StackPoint;
    INT16U StackInit;
    INT16U TimeToWait;
    INT8U  State;
    INT8U  SuspendedType;
    INT8U  Priority;
} TCB;
TCB Tab_Task[NUMBER_OF_TASKS];
```

Exercício

- Crie um **estrutura de dados TCB** e utilize-a para implementar funções para:
  - guardar os dados de uma tarefa no TCB.
  - iniciar a execução de uma tarefa.

Exercício

- Utilizando o TCB e a função para troca de contexto, implemente uma função que permita a uma tarefa **suspender-se em um determinado ponto de execução e retornar mais tarde**.

- Faça um programa em que **duas tarefas executem-se alternadamente**.

Criação/instalação de tarefas

- As tarefas em um sistema multitarefas são implementadas como um **laço infinito**. Assim, a troca de uma tarefa por outra pode ocorrer de duas formas:
  - Uma chamada de serviço do sistema (**função**) que coloca a tarefa em estado de espera ou suspensão.
  - Ou a **preempção** da tarefa por outra tarefa devido à ocorrência de um evento de **interrupção** (apenas para sistemas preemptivos).

Criação/instalação de tarefas

```
static void Task_1(void)
{
   /* task setup */
   int test;

   /* task main loop */

   for (;;)
   {
       // Escreva o código da tarefa aqui !!!
   }
}
```

Criação/instalação de tarefas

- Para que o sistema possa executar as tarefas pela primeira vez, os **dados de acesso à tarefa** (ex.: endereço do código, estado da pilha, endereço da pilha, ...) devem estar previamente gravados no bloco de controle de tarefas.

- Este processo é denominado de **criação ou instalação da tarefa** e é realizado através de uma chamada de serviço do sistema.

```
InstallTask(&Task_1,"Task Test",100,1);
```

Escalonamento de tarefas

O termo escalonamento identifica o procedimento de ordenar as tarefas prontas para execução.

O **escalonador** é o componente do núcleo do sistema responsável por **determinar**, em tempo de execução, qual será **a próxima tarefa a ser executada**, ou seja, é o componente responsável pela gestão do processador.

Código-exemplo para o escalonamento de tarefas retirado do **BRTOS**

```
SelectedTask = OSSchedule();    // decide a próxima tarefa
```

Escalonamento de tarefas

Há diveras técnicas de se realizar o escalonamento da tarefas, como:

- Fila FIFO (primeira na fila é executada)
- Round-Robin (execução cíclica com tempo definido)
- Por prioridade (executa-se a mais prioritária)

Escalonamento de tarefas

Fila FIFO:

- Tarefas que se tornam aptas a executar são inseridas no final da fila.
- A tarefa que está no início da fila é sempre a próxima a executar.
- Uma tarefa é executada até que libere explicitamente o processador ou termine sua execução.

Não é determinístico, pois os instantes de execução das tarefas variam dependendo da ordenação da fila.

Escalonamento de tarefas

Exemplo fila FIFO

Tarefas	Tempo de computação (Ci)
A	12
B	8
C	15
D	5

Tempo médio de espera para duas ordenações diferentes da fila.

Ordem A-B-C-D = $\frac{(0+12+20+35)}{4} = 16,75$

Ordem D-A-B-C = $\frac{(0+5+17+25)}{4} = 11,7$

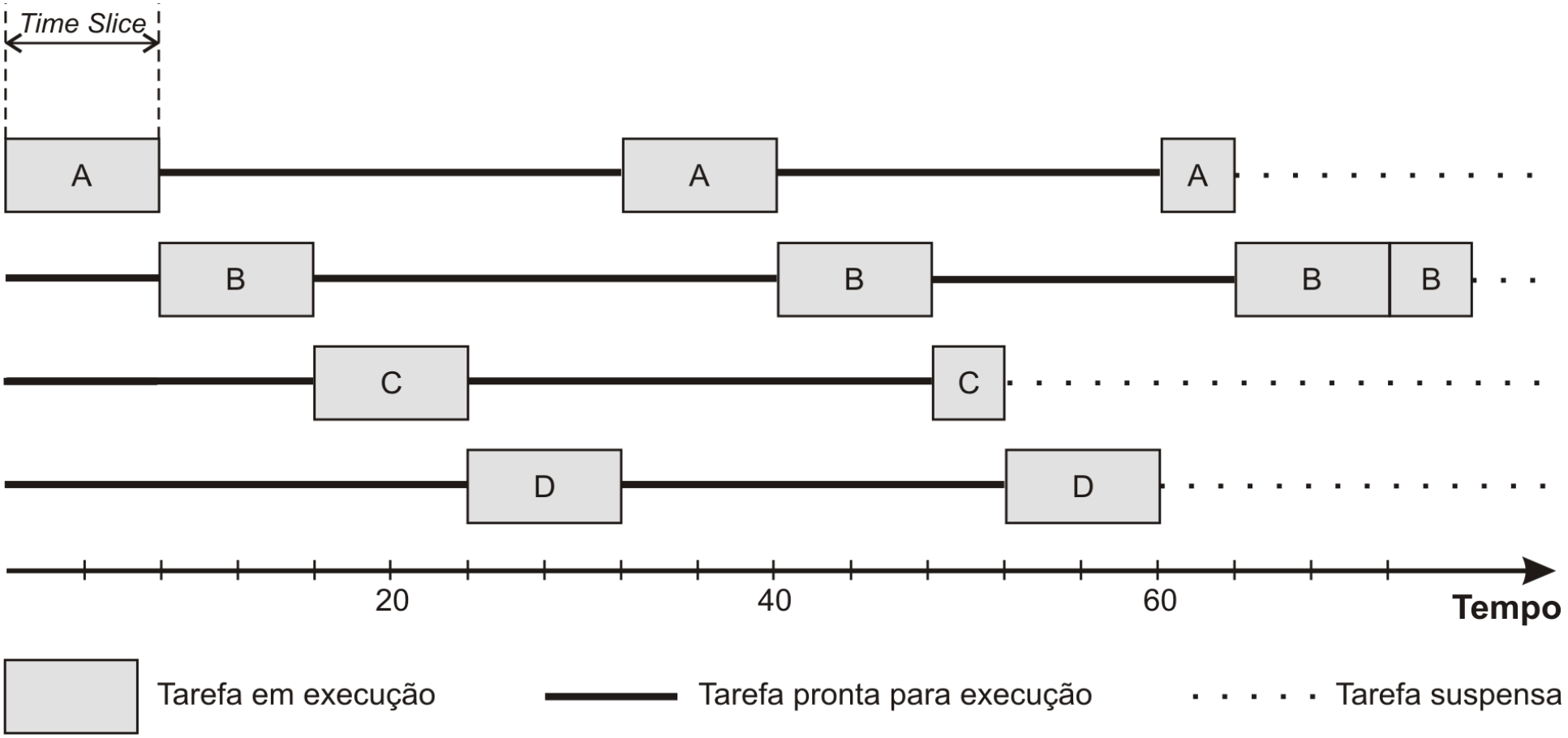
Escalonamento de tarefas

Round-Robin:

- execução cíclica com tempo máximo definido.
- provê a cada tarefa uma mesma quantidade de tempo de execução do processador.
- esta fração de tempo é conhecida por *time slice* (fatia de tempo) ou *quantum*.

Escalonamento de tarefas

Round-Robin:



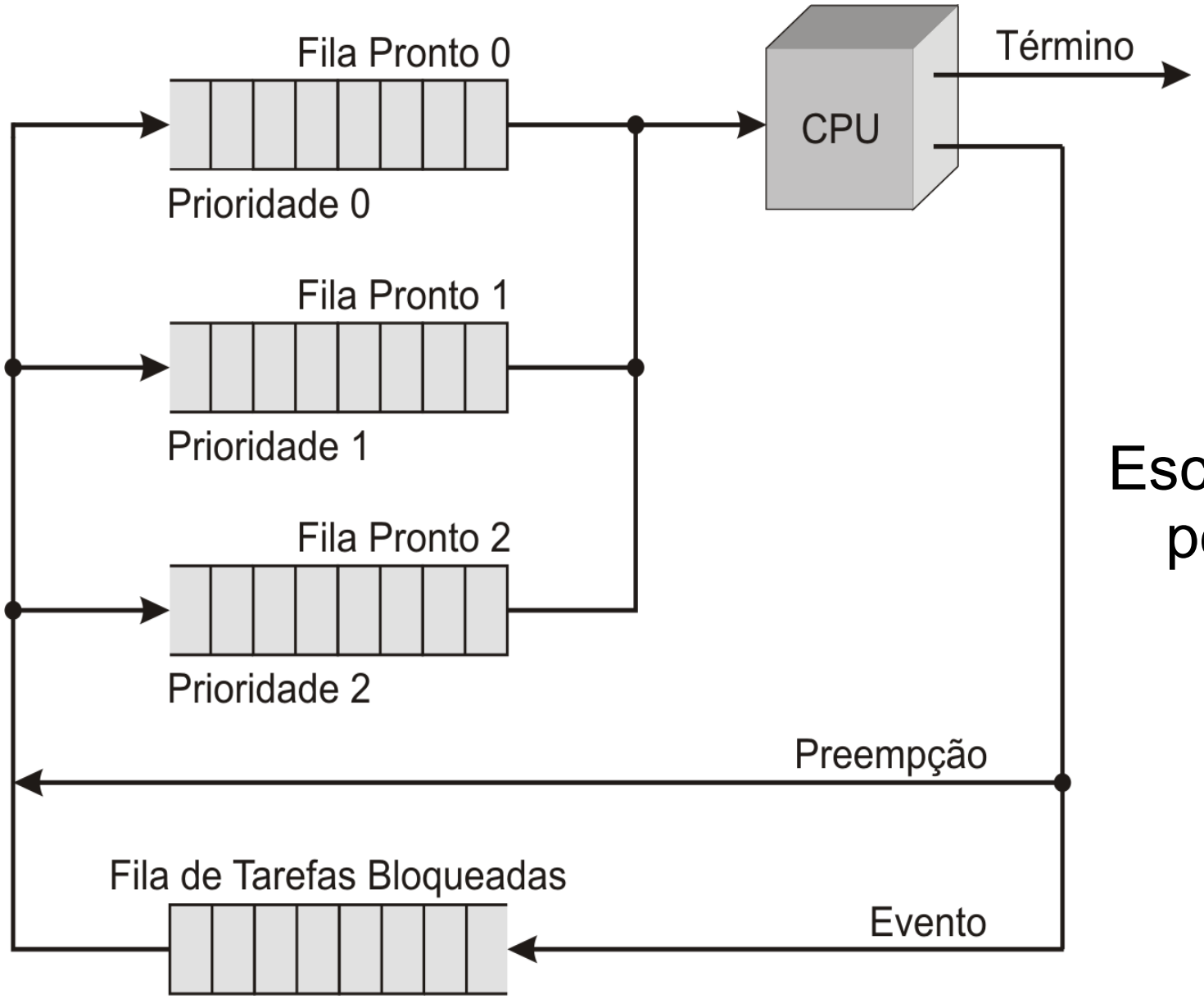
Escalonamento de tarefas

Por prioridade:

- as prioridades são utilizadas no escalonamento para determinar o grau de importância de uma determinada tarefa.
- executa-se sempre a tarefa mais prioritária que estiver pronta para execução.

Quando usado com núcleo preemptivo, aumenta o determinismo do sisten
Por isso é geralmente utilizado em sistemas operacionais de tempo real.

Escalonamento de tarefas



Escalonamento por prioridade

Escalonamento de tarefas

Escalonamento por prioridade:

```
for (prioridade = PRIORIDADE_MAXIMA; prioridade > 0;
    prioridade--)
{
        tarefa = Tab_prioridades[prioridade];
        if (TCB[tarefa].estado == PRONTA)
        {
            break;
        }
 }

proximo_stackpointer = TCB[tarefa].StackPointer;
```

Exercício

- Crie uma tabela com as “prioridades” das tarefas e escreva uma função de escalonamento por prioridade para selecionar a próxima tarefa a ser executada.

Sistemas de tempo real

Em um sistema de tempo real, as tarefas possuem prazos que devem ser cumpridos.

O não cumprimento da tarefa no prazo determinado pode levar a **consequências desastrosas** (*hard real time*) ou a um **menor de desempenho** do sistema (*firm* ou *soft real time*)

Sistemas de tempo real

Exemplos:

- injeção eletrônica e freio ABS (*hard*)
- controlador de reator nuclear (*hard*)
- controlador de avião (*hard*).
- sistemas de vídeo ao vivo (*soft*)
- servidor de transação de cartão de crédito/débito (*soft*).

Sistemas de tempo real

- Para um sistema de tempo real não basta ser **suficientemente rápido na maioria das vezes**, mas é necessário **garantir** um tempo máximo de execução no pior caso (WCET – *worst case executing time*).

Sistemas de tempo real

Pode-se classificar as tarefas de tempo real a partir das conseqüências do não cumprimento de seus prazos (*deadlines*):

- **Tarefas Críticas**: quando sua conclusão depois do deadline pode causar **falhas graves** no sistema de tempo real ou nas aplicações a que se destina.

Tarefas Não Críticas ou Brandas: A conclusão destas tarefas após seu deadline implica, no máximo, numa **diminuição de desempenho** do sistema.

Sistemas de tempo real

Os modelos de tarefa comportam dois tipos de tarefas segundo suas **freqüências de ativações**:

- **Tarefas Periódicas**: As ativações de uma tarefa ocorrem seqüencialmente, em um intervalo de tempo bem definido.

- **Tarefas Aperiódicas ou Tarefas Assíncronas**: As ativações de uma tarefa devem-se a eventos internos ou externos, definindo uma característica aleatória nessas ativações.

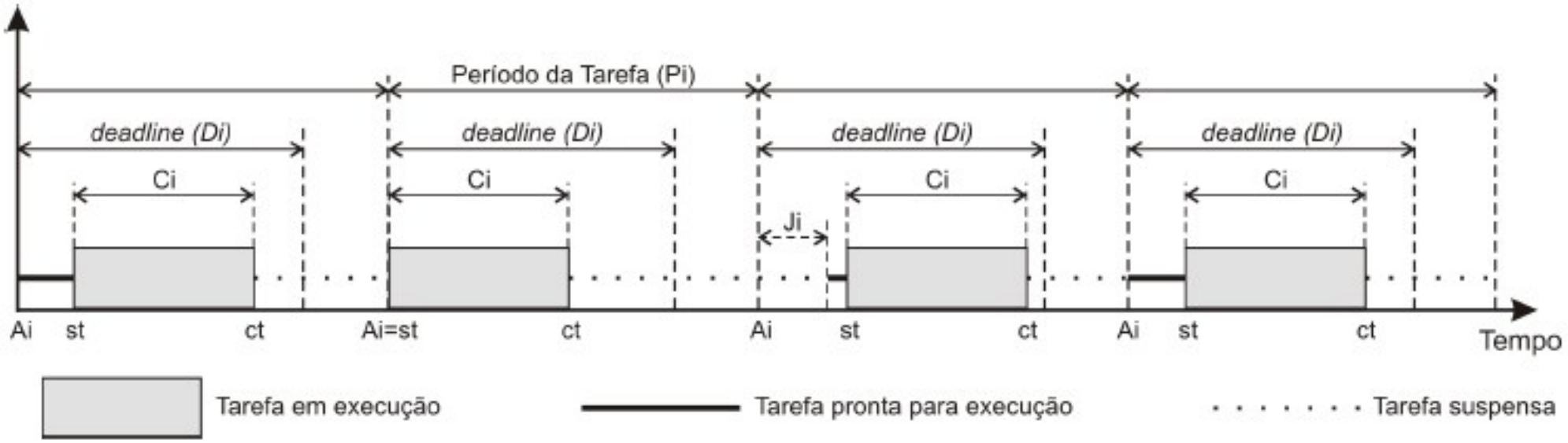
Sistemas de tempo real

Tempos característicos das tarefas:

- **Tempo de computação:** é o tempo necessário para a execução completa da tarefa.
- **Tempo de início**: instante de início do processamento da tarefa em uma ativação.
- **Tempo de término**: instante em que se completa a execução da tarefa ativa.
- **Tempo de chegada**: instante em que o escalonador toma conhecimento da ativação da tarefa.
- **Tempo de liberação**: instante do início da execução.

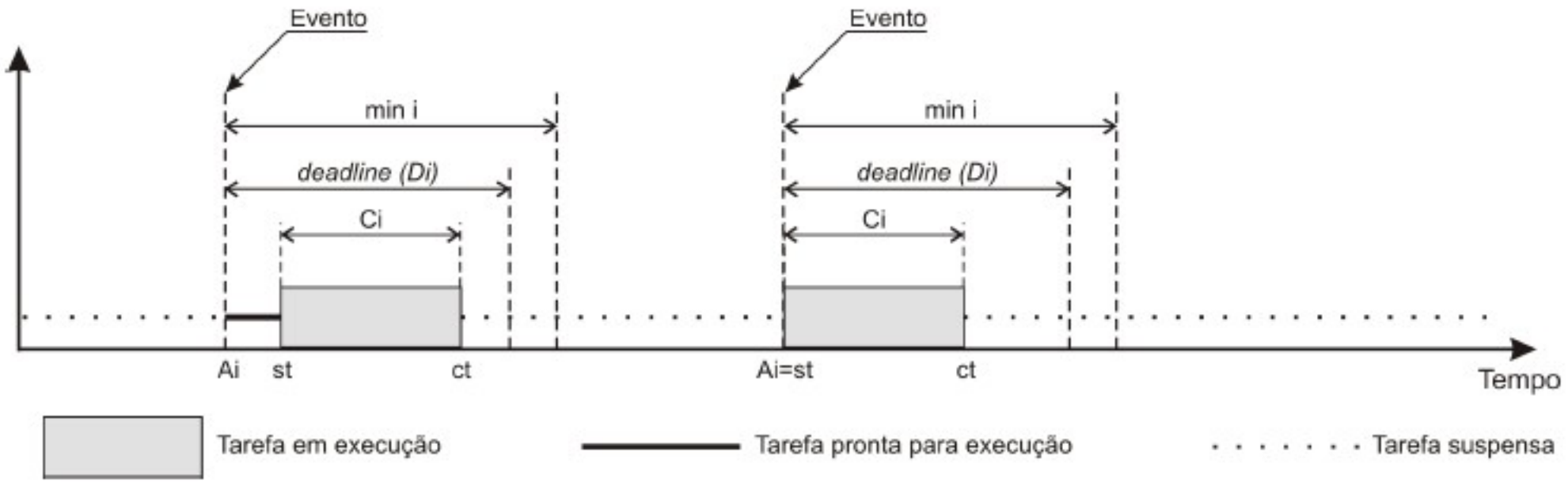
Sistemas de tempo real

Tempos característicos das tarefas periódicas: Ji, Ci, Pi, Di, onde Ci é o **tempo de computação** da tarefa, Pi é o **período** da tarefa, Di é o **deadline** e Ji é o *Release Jitter* (**máximo tempo de liberação**).



Sistemas de tempo real

Tempos característicos das tarefas aperiódicas: Ci, Di e min i, onde Ci é o **tempo de computação**, Di é o **deadline** medido a partir do instante do evento que gerou o processamento da tarefa e min i é o mínimo intervalo entre duas requisições consecutivas da tarefa.



Sistemas de tempo real

Geralmente, no projeto de um sistema de tempo real emprega-se um **sistema operacional de tempo real** (**RTOS**).

Um RTOS é, em suma, composto por:

- gerenciador multitarefa;
- gerenciador de tempo;
- recursos para sincronização e comunicação entre tarefas;
- e recursos adicionais existentes em sistemas operacionais, como sistemas de arquivos e *drivers* de dispositivos.

Exemplos de RTOS

Há vários RTOSes disponíveis comercialmente, para vários microcontroladores, como:

- FreeRTOS (gratuito, licença GPL modificada)
- uC/OS II e uC/OS III (empresa Micrium)
- ThreadX (empresa Express Logic)
- BRTOS (gratuito, code.google.com/p/brtos)

Sistemas de tempo real

Em um RTOS, o **gerenciamento das tarefas** geralmente é feito por um **núcleo preemptivo** com escalonamento dirigido por **prioridade**.

E o **gerenciamento de tempo** é feito com base em uma interrupção que ocorre periodicamente, denominada **marca de tempo (*timer tick*)**.

Esta pode ser gerada por uma **interrupção de temporizador** por estouro de tempo.

RTOS – Timer Tick

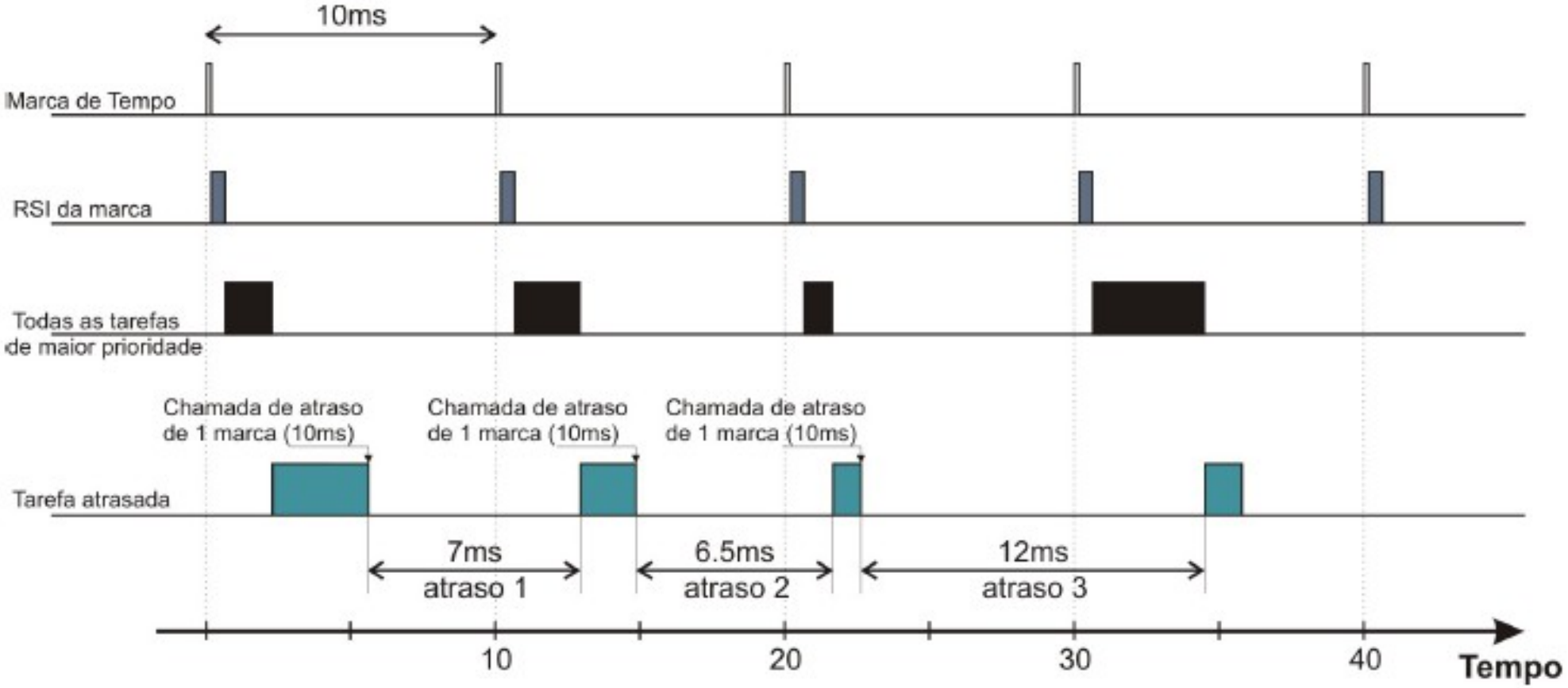
Utilizações da **marca de tempo (*timer tick*)** incluem a implementação de serviços de atrasos (*delay)* e estouro de tempo (*timeout*).

Exemplo: um serviço de atraso pode ser usado por uma tarefa que precisa aguardar um determinado tempo, como a escrita de um caractere em um *display* de LCD.

```
Display_escreve("Olá!");
OS_delay(10);    // aguarda 10 marcas de tempo
```

RTOS – Timer Tick

Geralmente, as marcas de tempo de RTOS variam entre **1ms a 10ms**. Valores menores aumentam a sobrecarga do sistema e valores maiores diminuem a resolução na contagem do tempo.



Exercício

- Crie uma interrupção de temporizador com período de 1ms.
- Utilize esta interrupção para implementar uma função de *delay* que coloca a **tarefa atual em estado de espera** quando chamada e cujo **argumento é o número de marcas de tempo** (ticks) em que a tarefa deve permanecer neste estado.

Comunicação entre tarefas

Muitas vezes é necessário que as tarefas tenham como trocar informações (se comunicar).

Como cada tarefa possui uma pilha exclusiva, a comunicação só pode ocorrer através de variáveis globais compartilhadas.

Porém, se o acesso de escrita e leitura a estas variáveis não for bem gerenciado, o resultado pode ser a corrupção dos dados!

Comunicação entre tarefas

Uma solução possível para gerenciar o acesso é o bloqueio/desbloqueio de interrupções.

→ Desabilita Interrupções

→ Realiza escrita ou leitura do dado

→ Reabilita Interrupções

Sincronização entre tarefas

Porém, além da comunicação, as tarefas podem necessitar **sincronização.**

Exemplos:

- ao realizar um primeiro processamento de dados em uma tarefa, pode ser necessário que outra tarefa de maior prioridade passe a ser executada utilizando os dados previamente processados.

- um evento externo detectado por uma interrupção deve dar início ao processamento de uma determinada tarefa.

Sincronização entre tarefas

Um mecanismo comum em sistema multitarefas (p. ex.: RTOS) que permite a comunicação e sincronização entre tarefas é a utiização de Semáforos.

Um semáforo funciona como um sinalizador, através do qual uma tarefa ou interrupção envia o sinal e outra tarefa recebe o sinal.

Semáforos

Assim, em um sistema multitarefas geralmente existem os seguintes serviços:

- Criação/inicialização de semáforo
- Envio ou sinalização de semáforo (POST)
- Espera/recepção de semáforo (PEND)
- Desativação do semáforo (opcional)

Semáforos

Podem ser de dois tipos:

- Binários:
  - só guardam a ocorrência do último evento – 0 ou 1.

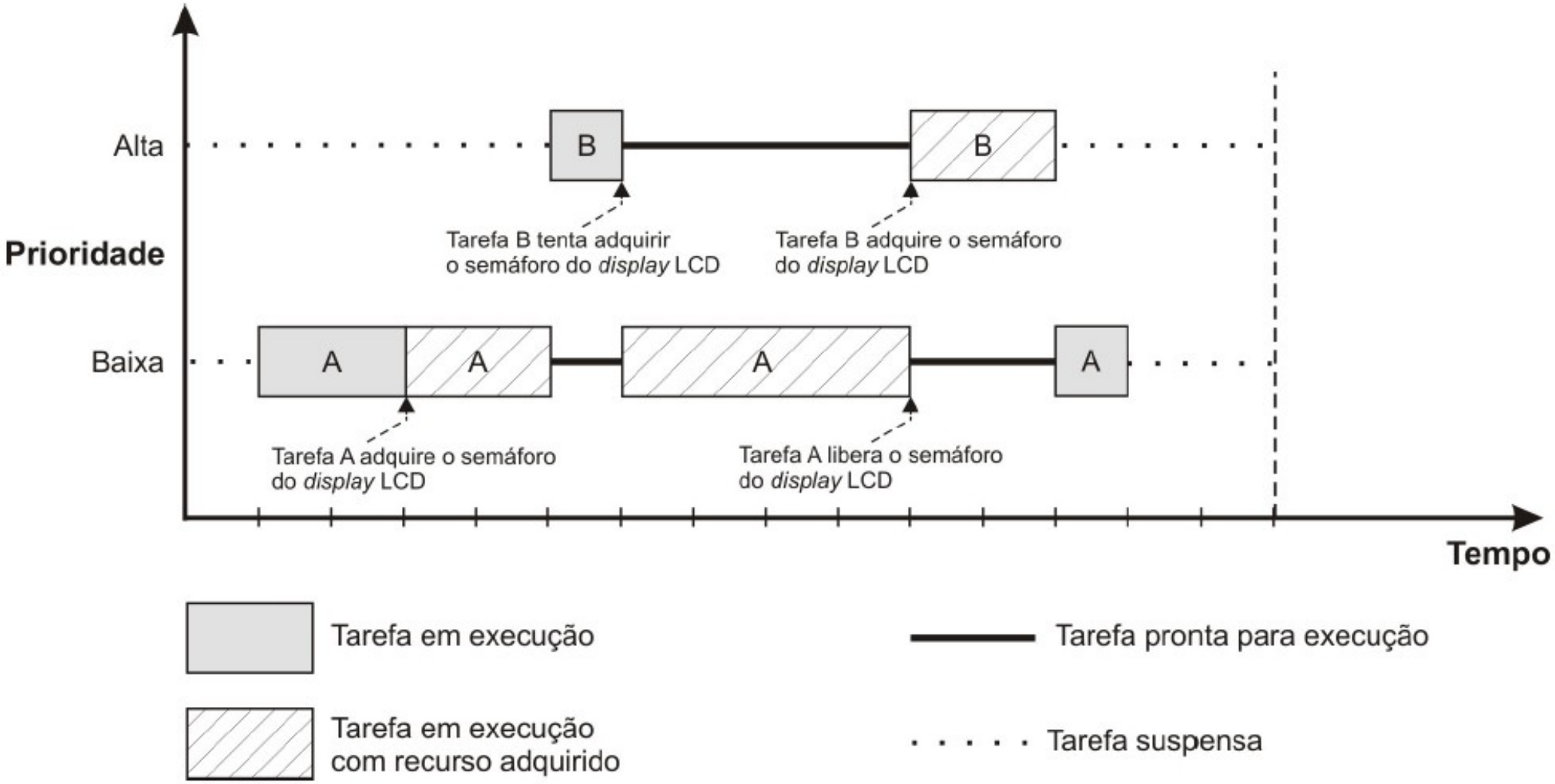
- Contadores:
  - guardam a contagem dos eventos ainda não recebidos - valor de 8, 16 ou 32 bits.

Exemplos de uso de semáforos

```
// Declara semáforo
Semaforo *DataAccessSem;


void data_access(void)
{
  // Adquire semáforo sem timeout
  SemPEND(DataAccessSem,0);

  // Aqui é seguro acessar os dados compartilhados

  // Libera o semáforo
  SemPOST(DataAccessSem);
}
```

Utilização de semáforo para acessar dados compartilhados

Exemplos de uso de semáforos



Exemplo de semáforo controlando acesso a recurso compartilhado

Exemplos de uso de semáforos

Exemplo de função de um *driver* de display LCD

```
char write_lcd(char *string, int timeout)
{
  Adquire o semáforo do display LCD();
  while(*string)
  {
    escreve_caracter(*string,timeout);
    if (timeout){
      Libera o semáforo do display LCD();
      return (error);
    }else{
      string++;
    }
  }
  Libera o semáforo do display LCD();
  return (OK);
}
```

Recurso com semáforo encapsulado

Outros recursos comunicação/sincronização

Semáforos de Exclusão Mútua (Mutex)

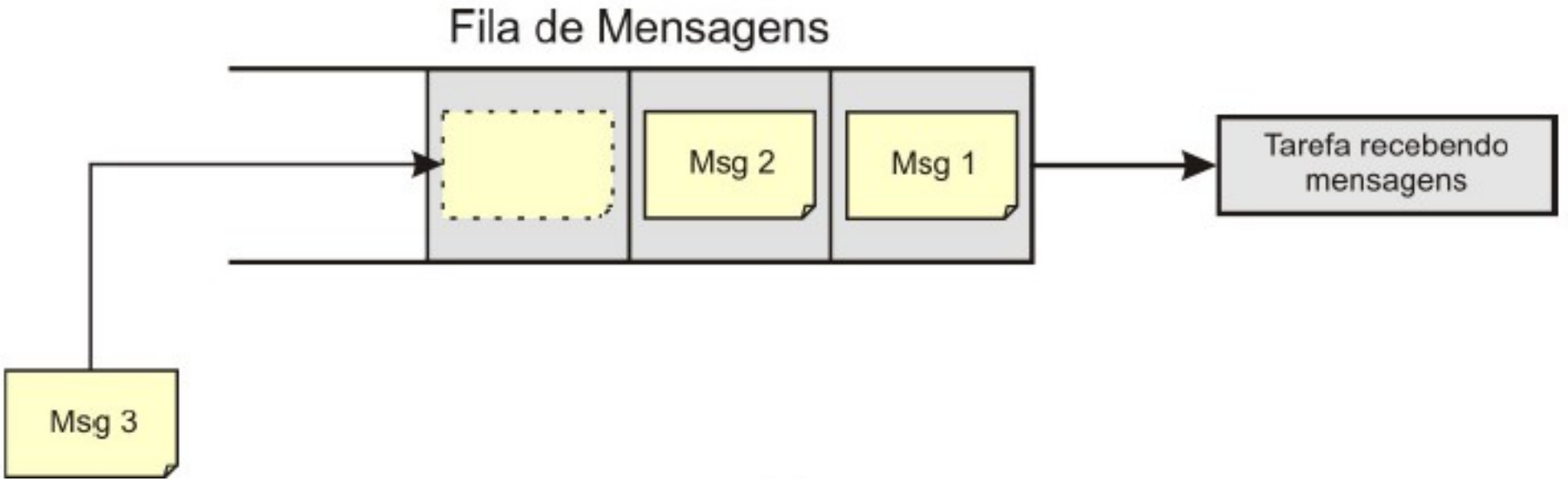
Caixas de Mensagem (*Message Mailboxes*)

Filas de Mensagens (*Message Queues*)

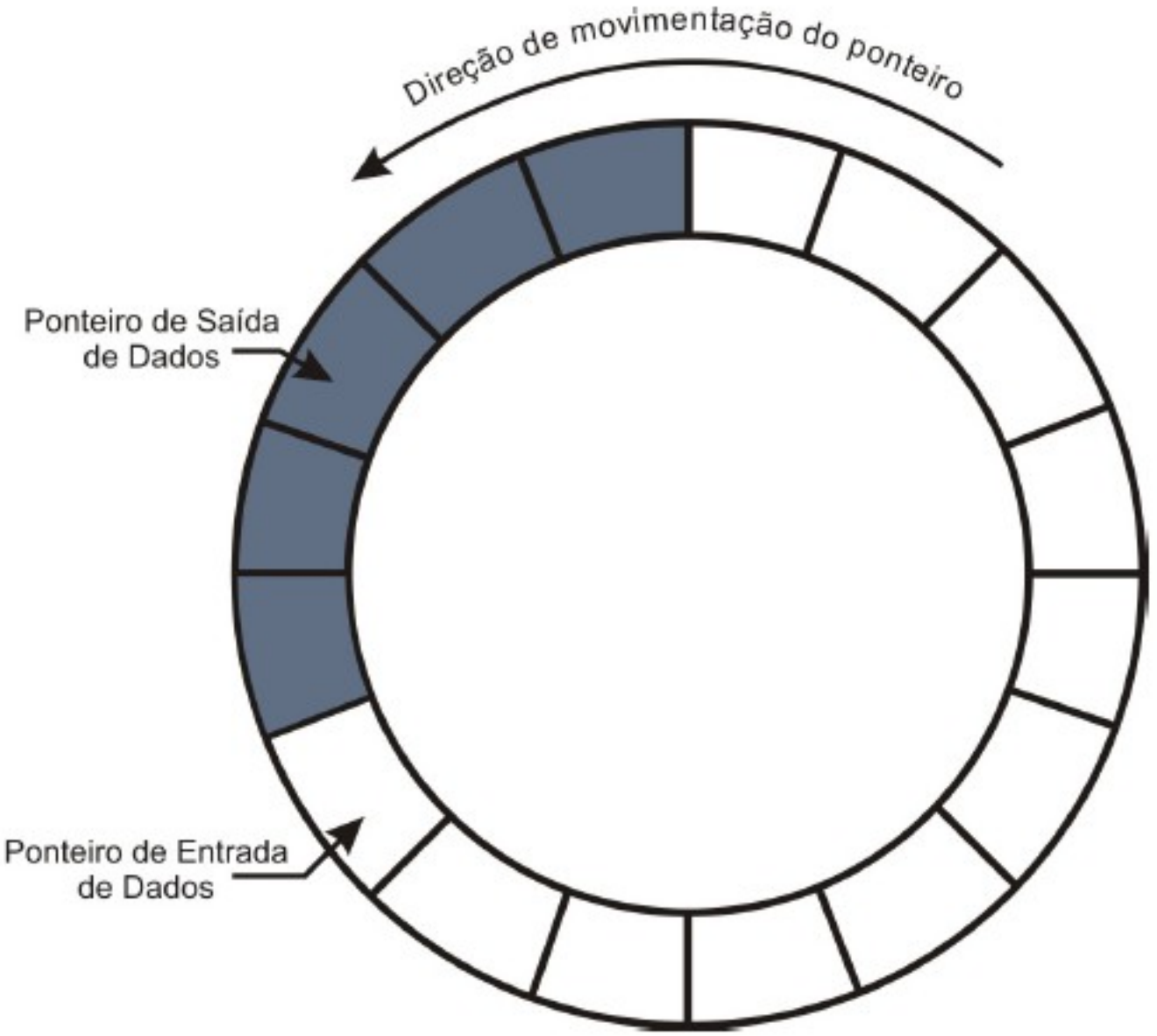
Recurso com semáforo encapsulado

Filas de mensagens

Funciona como a junção de um semáforo contador com um buffer circular, que além de sinalizar o evento, permite o envio de dados de uma tarefa ou interrupção para outra tarefa.

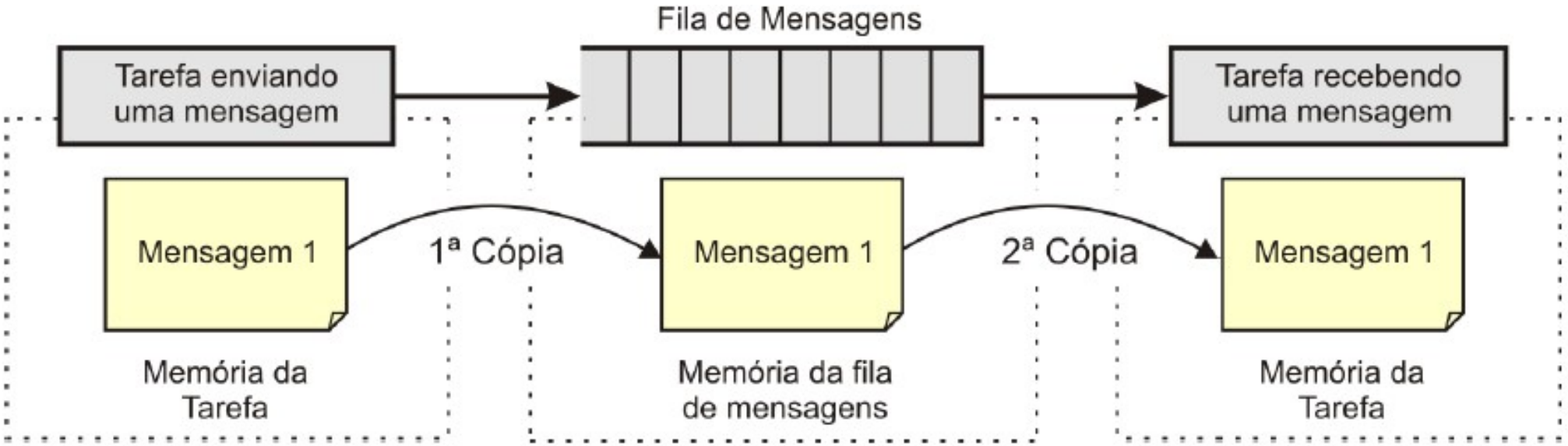


Filas de mensagens

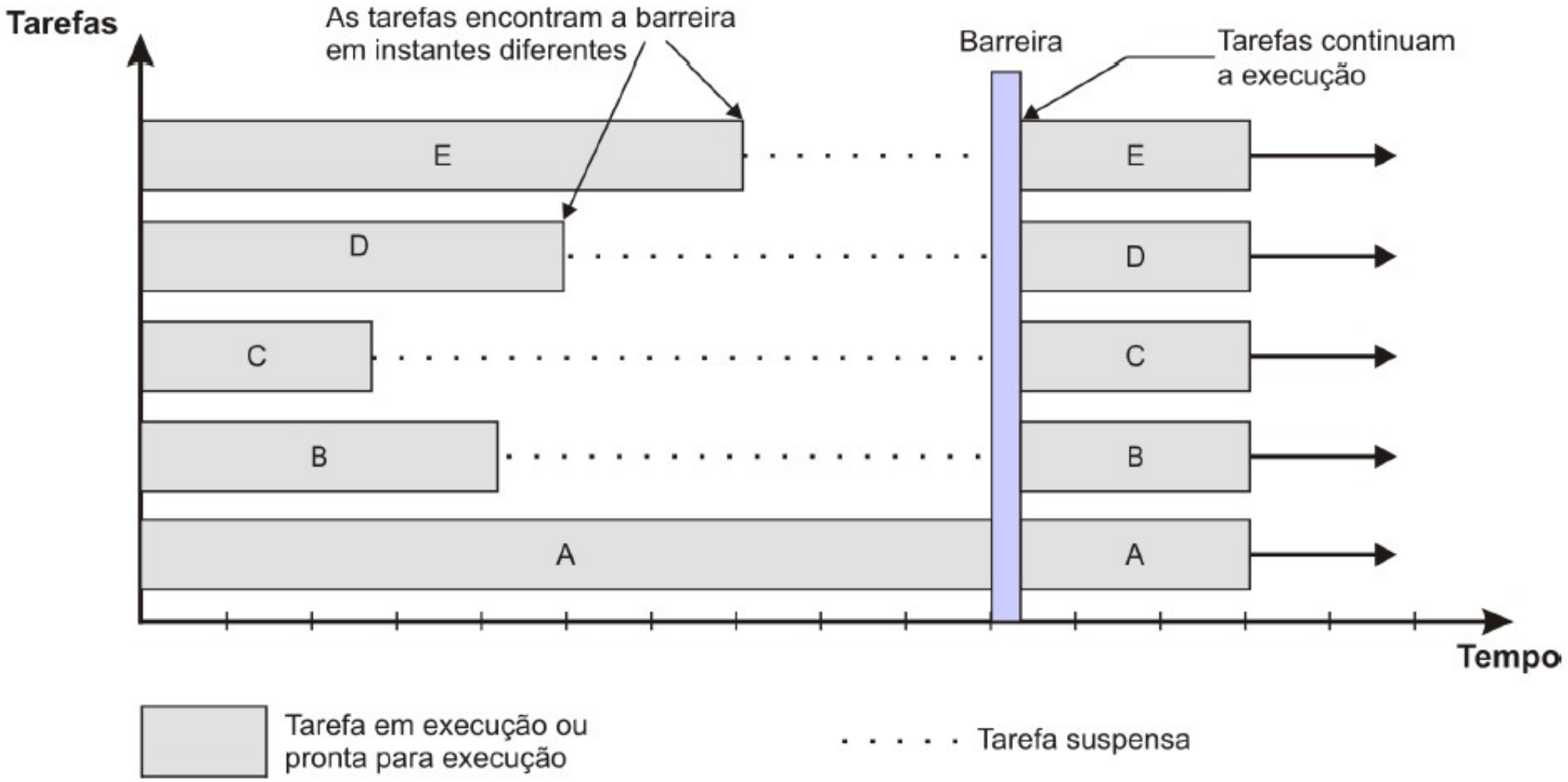


Operação de uma fila de mensagens

Filas de mensagens

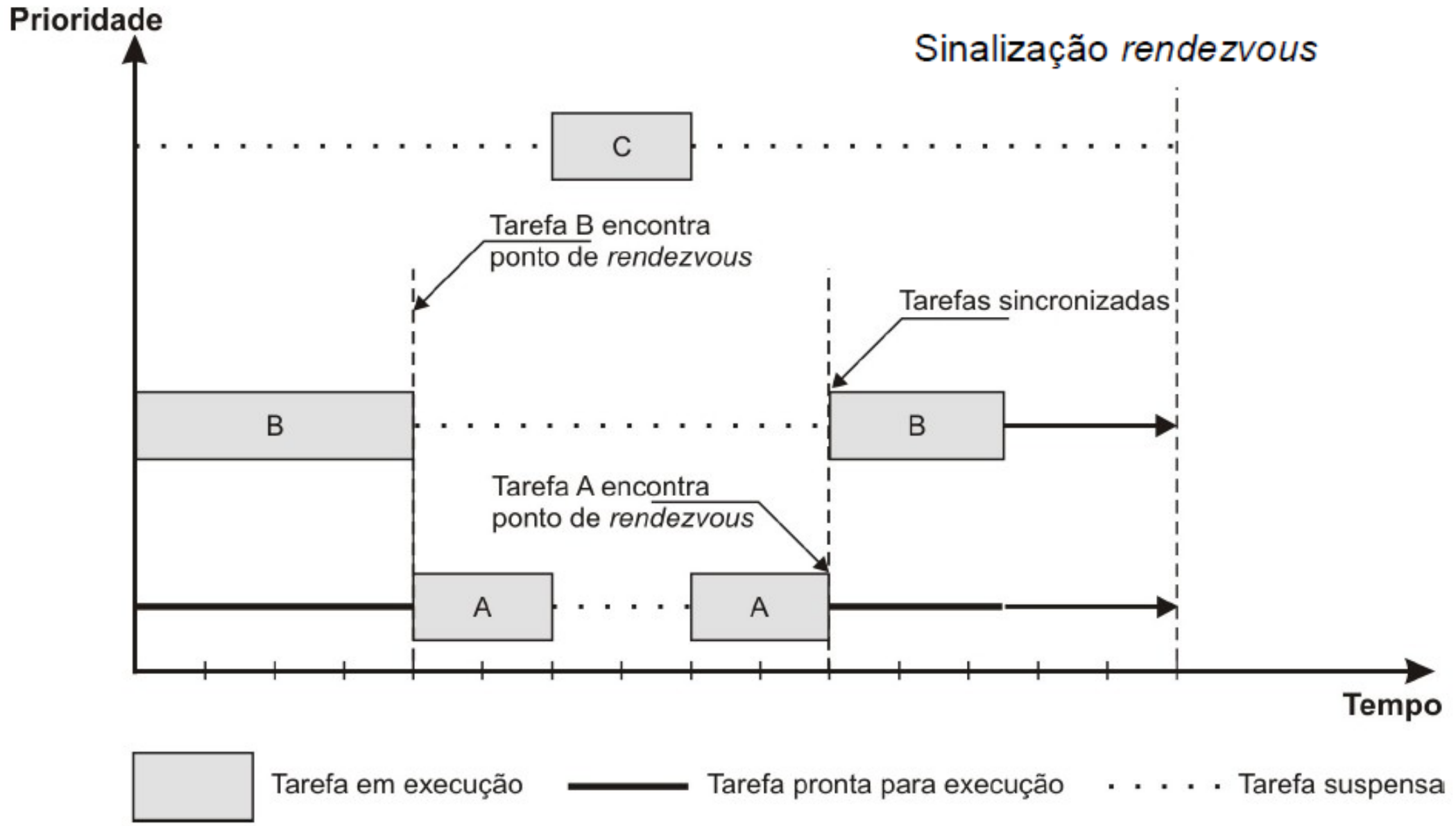


Técnicas de sincronização e comunicação entre tarefas

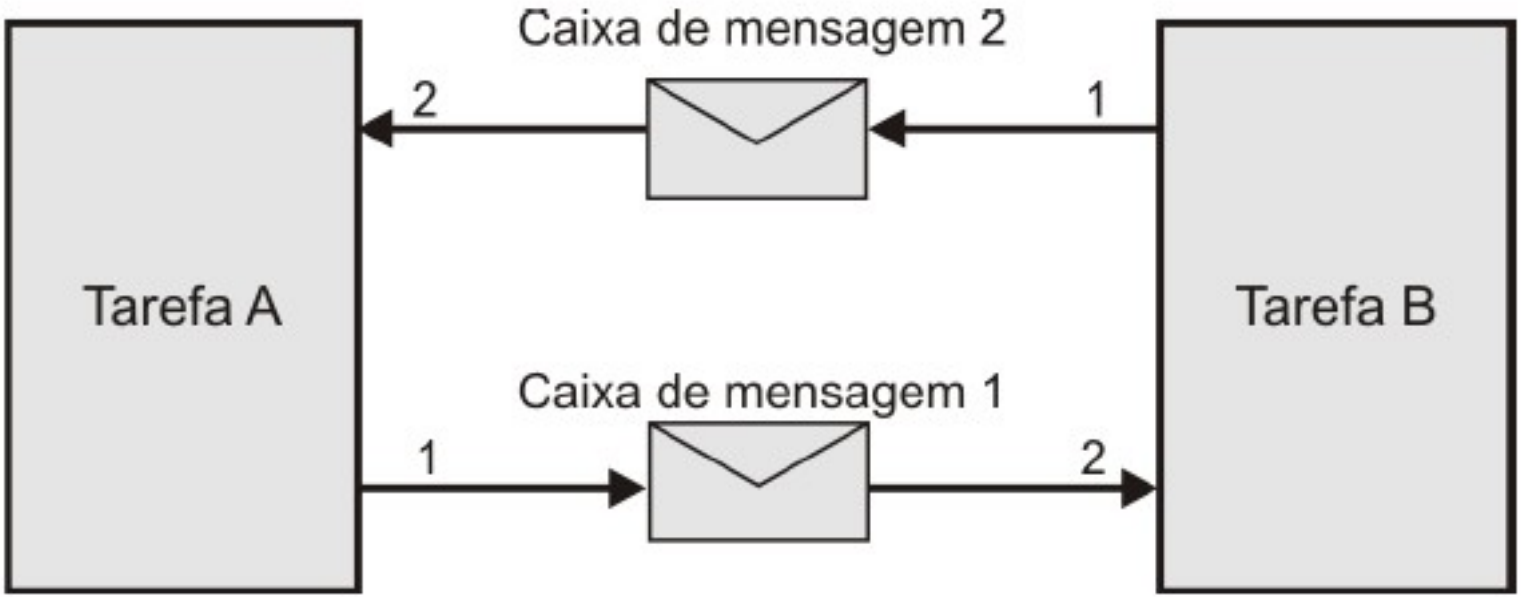


Sincronização por barreira

Técnicas de sincronização e comunicação entre tarefas

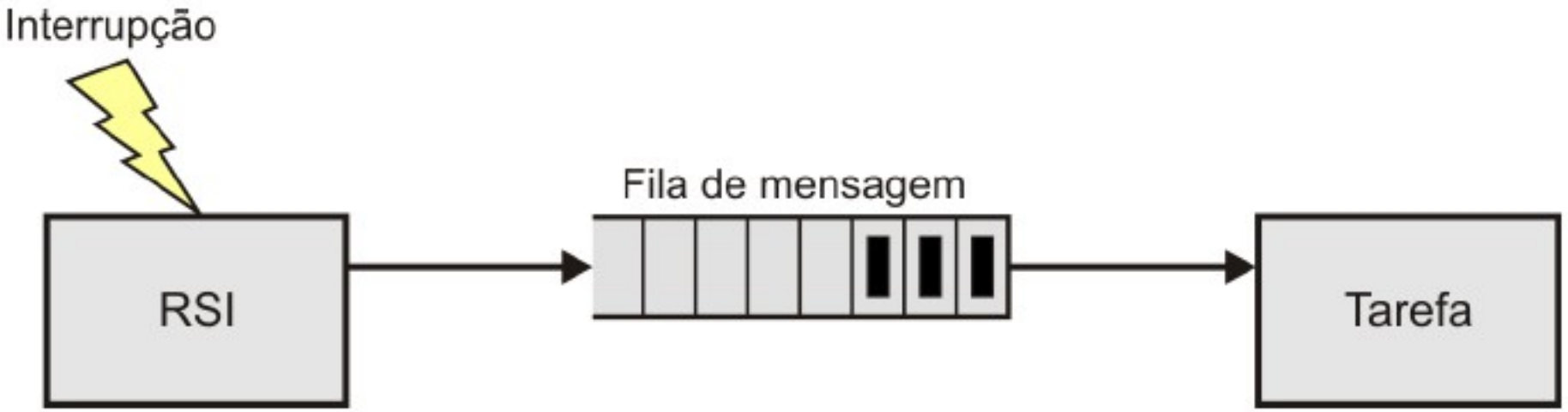


Técnicas de sincronização e comunicação entre tarefas



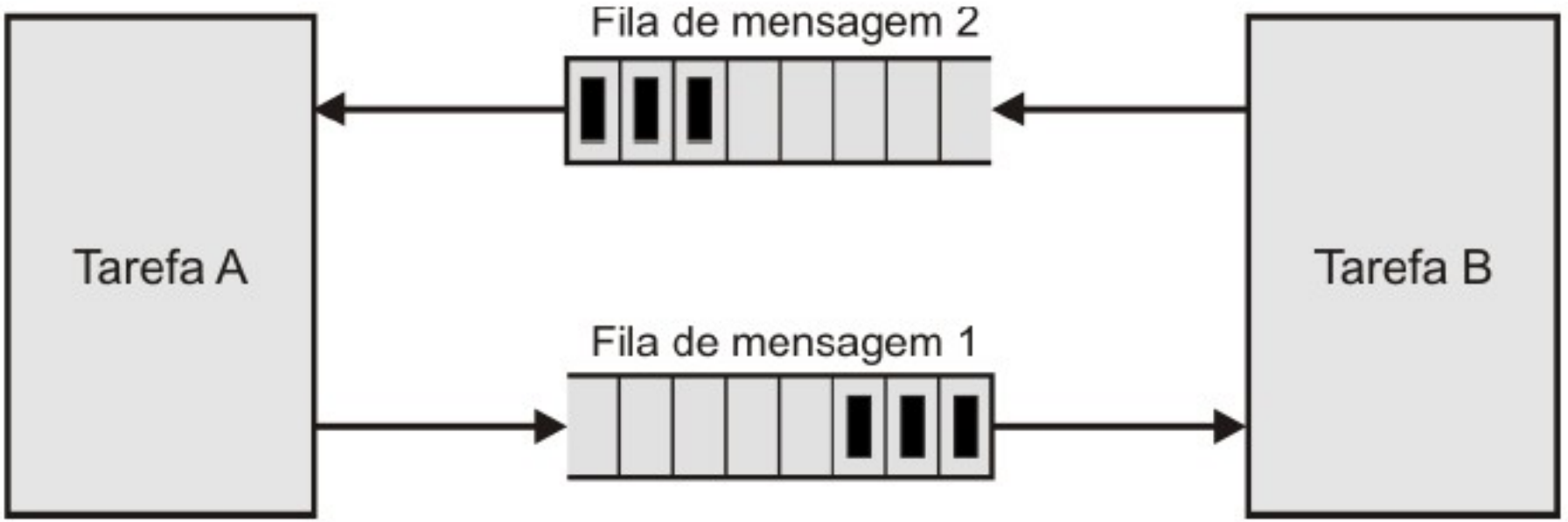
Sinalização *rendezvous* com troca de dados por caixa de mensagem

Técnicas de sincronização e comunicação entre tarefas



Comunicação não vinculada utilizando fila de mensagem

Técnicas de sincronização e comunicação entre tarefas



Comunicação fortemente acoplada utilizando fila de mensagem

Referências

[1] Jean J. Labrosse, ***µC/OS – The Real Time-Kernel***, CMP Books, 2nd ed., 2002.

[2] Jean-Marine Farines, Joni S. Fraga, Rômulo S. de Oliveira, ***Sistemas de Tempo Real***, IME-USP, 2000.

[3] Qing Li, Carolyn Yao, ***Real-Time Concepts for Embedded Systems***, CPM Books, 2003.

[4] Stuart R. Ball, ***Embedded Microprocessor Systems – Real World Design***, Newnes Books, 3rd. ed., 2002.

[5] Edward L. Lamie, ***Real-Time Embedded Multithreading***, CPM Books, 2005.

[6] Jean J. Labrosse, ***Embedded Systems Building Blocks***, R&D Books, 2nd ed., 2002.

[7] Gustavo W. Denardin e Carlos H. Barriquello, ***Brazilian Real-Time Operating System***, 2009. Disponível em: http://code.google.com/p/brtos/

[8] Expresslogic (Desenvolvedora do ThreadX® RTOS), ***Two Approaches - Unified and Segmented***. Disponível em: http://rtos.com/articles/18835.

[9] Ralph Moore (Desenvolvedor do SMX RTOS), ***Deferred Interrupt Processing Improves System Response***, 2005. Disponível em: http://www.smxrtos.com/articles/techppr/defint.htm.

[10] Ralph Moore (Desenvolvedor do SMX RTOS), ***Link Service Routines for Better Interrupt Handling***, 2005. Disponível em: http://www.smxrtos.com/articles/lsr_art/lsr_art.htm.