

Sistemas Embarcados

Gerenciamento e sincronia de tarefas Semáforos e MUTEX

Tiago Piovesan Vendruscolo

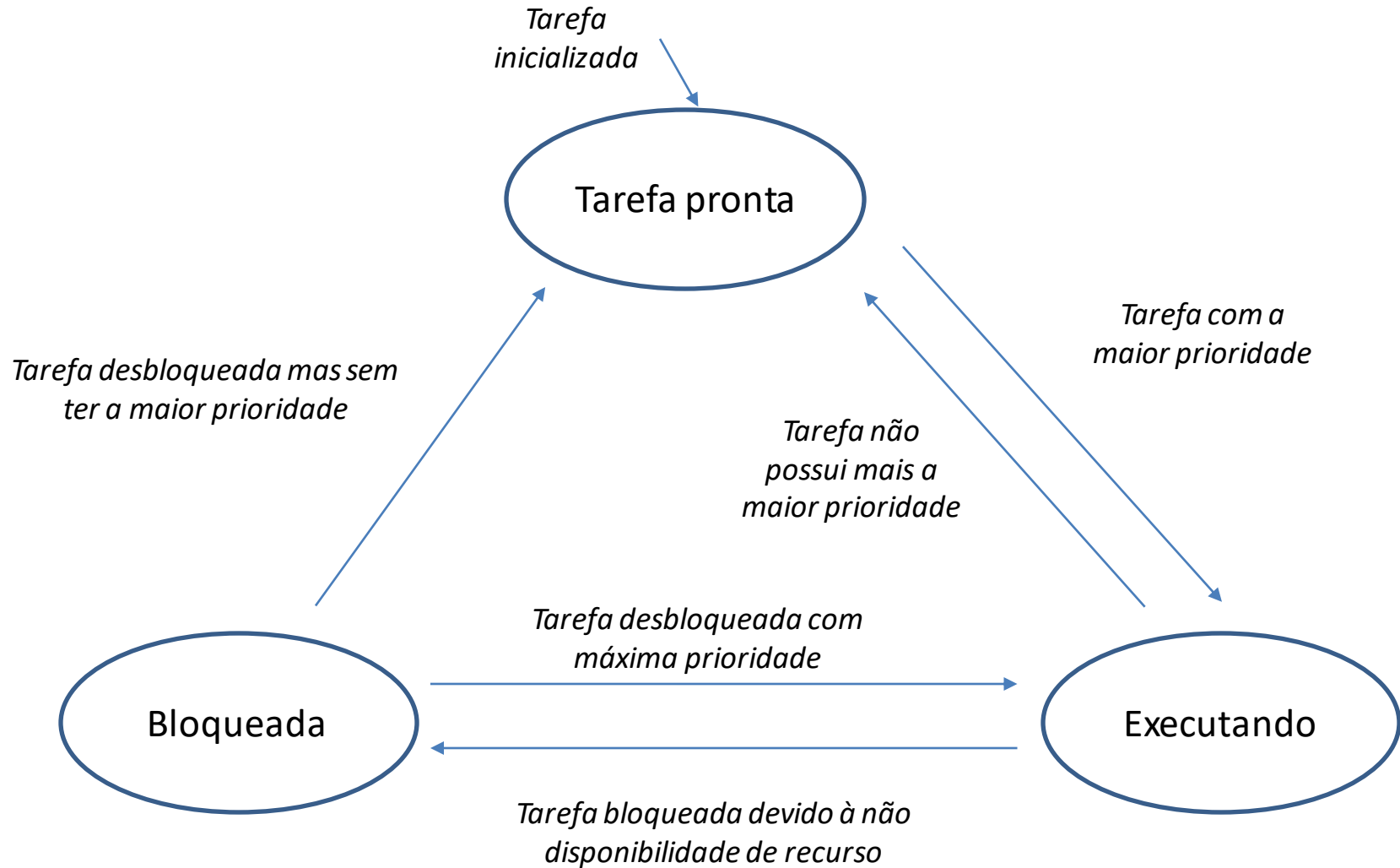


Esta licença permite que outros remixem, adaptem e criem a partir do trabalho para fins não comerciais, desde que atribuam o devido crédito aos autores originais. [4.0 international](https://creativecommons.org/licenses/by-nc-nd/4.0/)

- Atividades que podem ser executadas em paralelo devem ser implementadas em tarefas diferentes
- Funções com prioridades diferentes devem ser separadas em tarefas diferentes
- Em sistemas multitarefas com tarefas concorrentes, não é possível determinar a ordem em que os eventos irão ocorrer. Assim, uma tarefa deve sincronizar suas atividades com outras tarefas para que o sistema seja executado corretamente.
- A execução deve ser sincronizada para poderem coordenar o acesso aos recursos compartilhados no sistema
- Interrupções podem ser utilizadas para sincronizar ou transferir dados para uma ou mais tarefas.

- Vantagens em dividir o sistema em tarefas
 - *Controle mais preciso sobre partes do sistema com requisitos de tempo diferentes.*
 - *Torna a aplicação modular e limpa, facilitando o desenvolvimento, manutenção e testes.*
- Desvantagens em ter muitas tarefas
 - *Maior complexidade no gerenciamento de recursos compartilhados*
 - *Requer mais memória RAM, pois cada tarefa possui sua própria pilha*
 - *Maior consumo de CPU na troca de contexto*

Estado das tarefas



- *Seção crítica: parte do código que necessita ser executado sem interrupção com um recurso compartilhado.*
 - *Acesso à memória (variáveis globais)*
 - *Acesso à um canal de comunicação para leitura/escrita*
- *Como garantir acesso exclusivo a recursos compartilhados?*
 - *Desabilitar interrupções - mascaráveis*
 - *Aumento no tempo (latência) para o tratamento das interrupções*
 - *Desabilitar o escalonamento*
 - *Aumento da latência na execução de tarefas*
 - *Utilizar semáforos ou mutex*
 - *Mais complexo de implementar, utiliza mais tempo computacional*

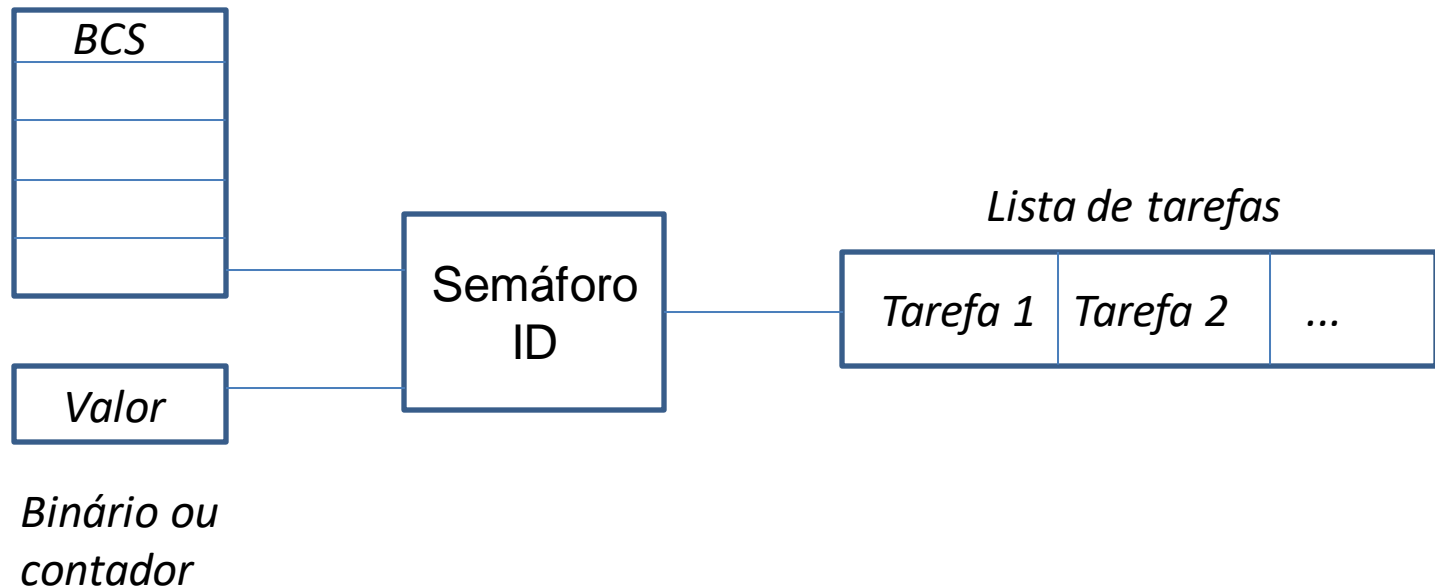
Formas de realizar o gerenciamento

- *O RTOS possui diversas formas de realizar o gerenciamento e/ou a sincronia das tarefas, entre elas, podemos citar:*
 - *Semáforos: binários, contadores e MUTEX*
 - *Fila de mensagens (Queues)*
 - *Software Timers*
 - *Grupos de eventos (event groups)*
 - *Task Notifications*

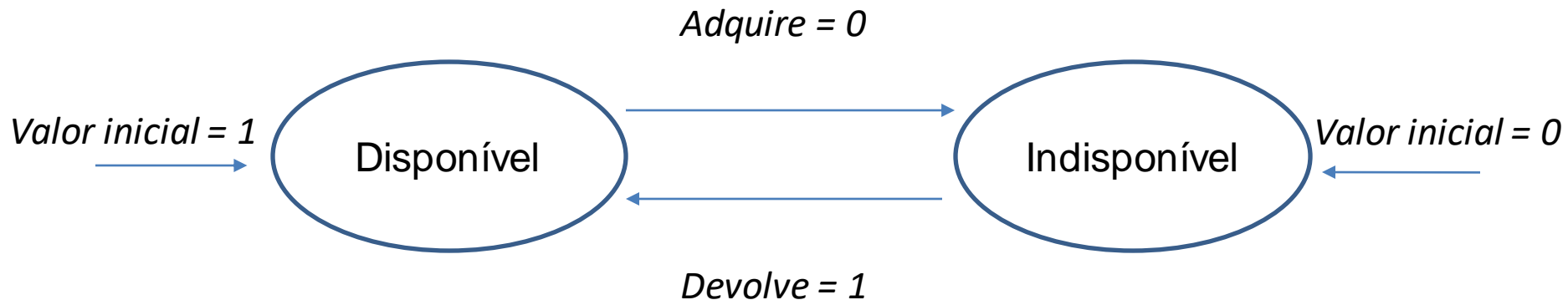
- *O semáforo é um dos mecanismos mais antigos a ser utilizados em sistemas operacionais. Ele consiste em uma estrutura de dados utilizada para:*
 - *Realizar o controle de acesso à recursos compartilhados*
 - *Sinalizar o momento de ocorrência de um evento*
 - *Sincronizar uma interrupção com uma tarefa*
 - *Permitir que duas ou mais tarefas sincronizem suas atividades.*
- *Um RTOS permite o uso de diversos semáforos, limitados ao consumo de memória RAM.*
- *Existem três tipos de semáforos fornecidos pelo kernel*
 - *Semáforos binários*
 - *Semáforos contadores*
 - *Semáforos de exclusão mútua (Mutex)*

Semáforos

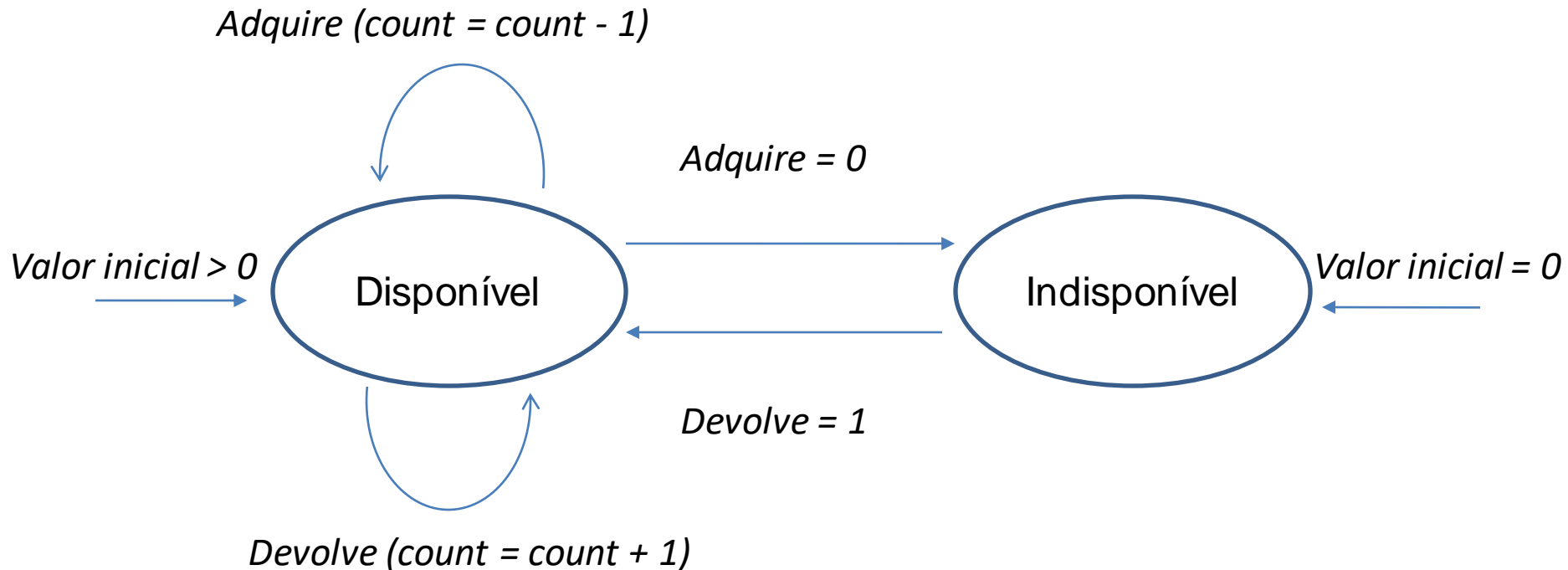
- *Ao alocar um semáforo, o núcleo do sistema operacional o associa a um bloco de controle de semáforo (BCS) que possui informações como a lista de espera de tarefas e o valor.*



- *Semáforo binário*
 - *Assume somente dois valores: 0 e 1, que correspondem respectivamente a fechado e aberto, ou indisponível e disponível.*
 - *Qualquer tarefa pode adquirir e liberar o semáforo.*
 - *No FreeRTOS os semáforos iniciam em zero (bloqueado), ou seja, precisam primeiro serem liberados para depois alguma tarefa adquirir.*

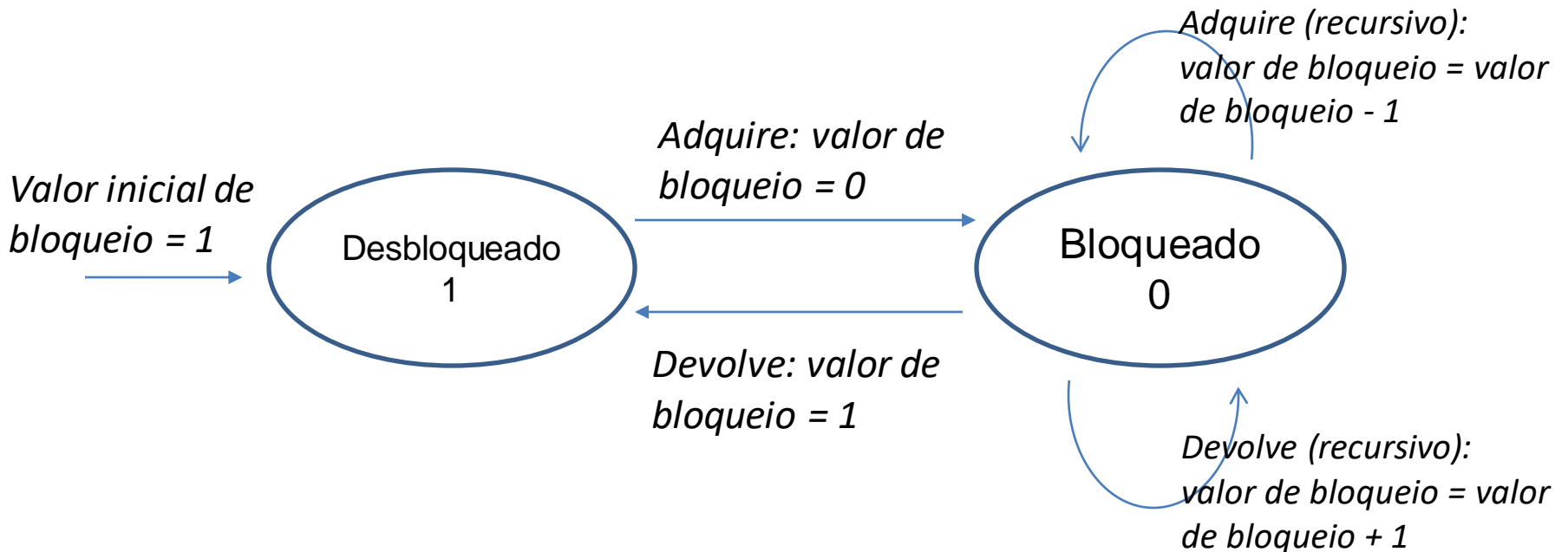


- *Semáforo contador*
 - *Utiliza um contador para permitir que ele seja adquirido e devolvido múltiplas vezes, definido pelo número do contador, que pode ser: 0 a 255, 0 a 65535, etc.*



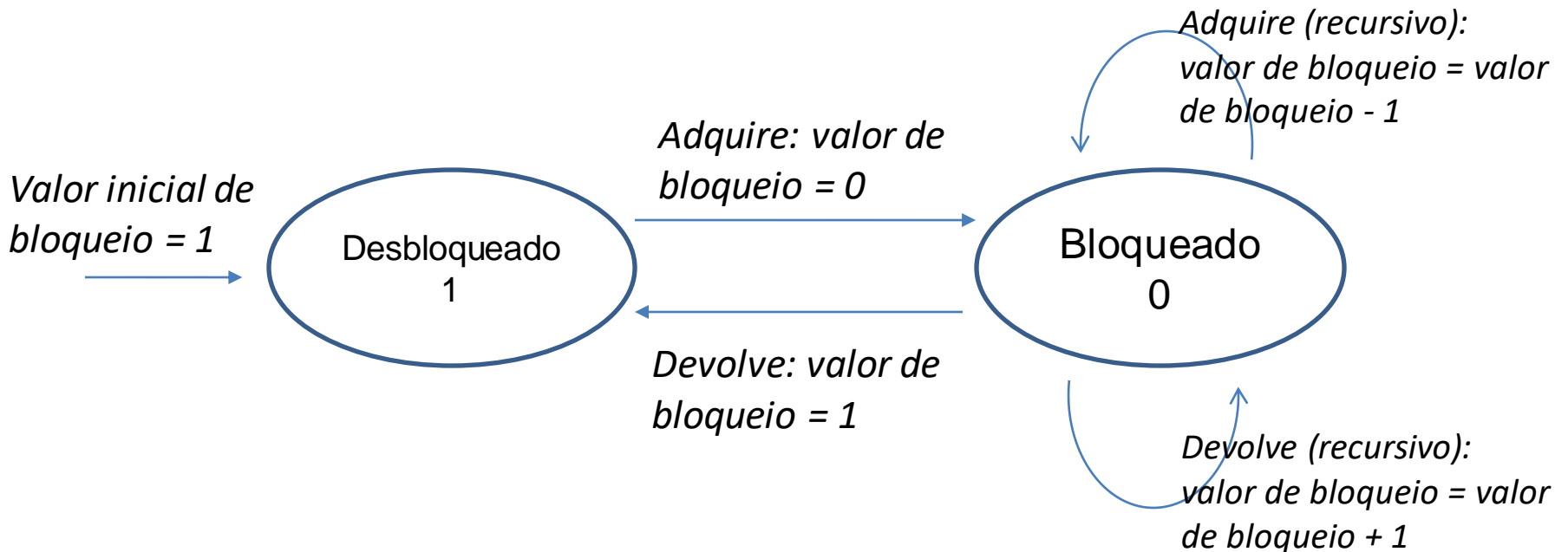
- *Semáforo Mutex*

- *É um tipo especial de semáforo binário que suporta acesso recursivo, posse, exclusão com segurança de uma tarefa e o uso de um ou mais protocolos para evitar problemas inerentes à exclusão mútua*
- *Os estados são: Bloqueado e desbloqueado*



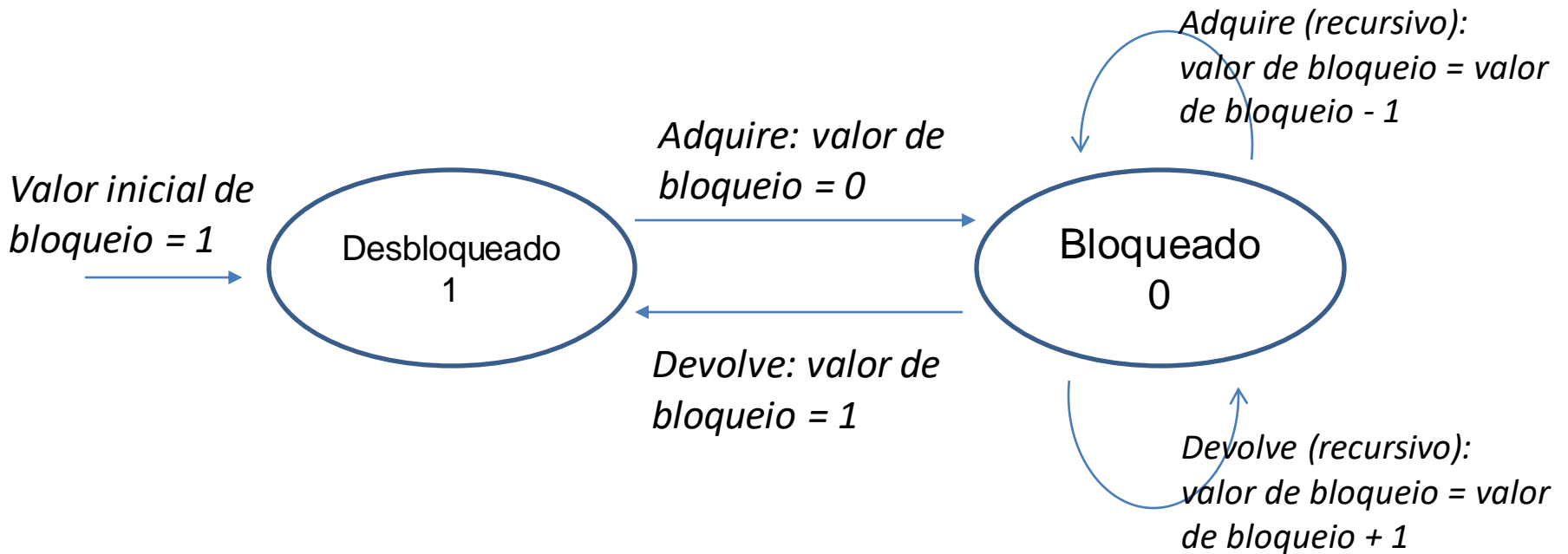
- *Semáforo Mutex*

- *Apenas a tarefa que adquiriu o mutex pode devolvê-lo (posse)*
 - *Garante que a tarefa não seja excluída*
- *Protocolos para evitar problemas inerentes à exclusão mútua, como a inversão de prioridade: Protocolo de herança de prioridade e Protocolo de teto de prioridade.*



- *Semáforo Mutex*

- *A tarefa de posse do mutex fica com prioridade máxima*
- *A mesma tarefa pode adquirir o mesmo mutex mais de uma vez (travando-o múltiplas vezes)*



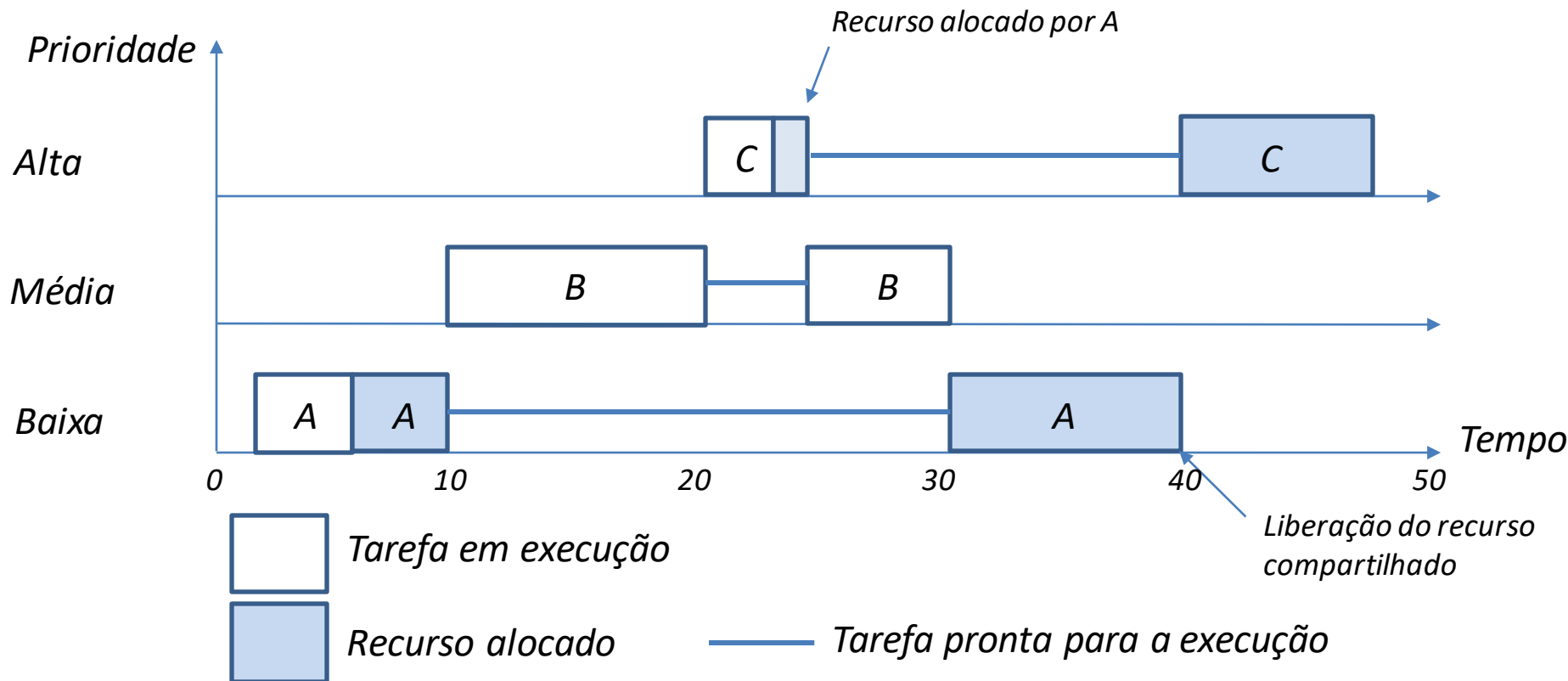
- *Operações básicas com semáforos*

Operação	Definição
Create	Cria o semáforo*
Delete	Deleta o semáforo
Take	Adquire o semáforo**
Give	Devolve o semáforo

- **De acordo com o semáforo utilizado, deve ser especificado:*
 - *Binário*
 - *Contador: estado inicial do contador e a ordem da fila de espera*
 - *Mutex: ordem da fila de espera e habilitar: protocolos para evitar inversão de prioridade, teto de prioridade, recursividade e exclusão com segurança.*
- ***Quando uma tarefa tenta adquirir o semáforo, temos três opções: 1) esperar, 2) esperar até um tempo máximo (timeout) 3) não esperar*

Inversão de prioridade

- *Ocorre quando uma tarefa de alta prioridade não pode ser executada devido ao bloqueio de um recurso compartilhado (e alocado) por uma tarefa de baixa prioridade, que foi preemptada por uma tarefa de prioridade média.*



■ *Caso da missão Mars Pathfinder*

- *Missão não tripulada (1997) para monitoramento de dados meteorológicos, imagens fotográficas, etc.*
- *Após alguns dias, o sistema começou ser reinicializado constantemente, inviabilizando o envio de informações meteorológicas para a terra*
- *Possuía três tarefas principais sendo executadas:*
 - *Tarefa periódica de controle de barramento 1553 (alta prioridade)*
 - *Tarefa de comunicação via rádio com a terra (média prioridade)*
 - *Tarefa de transferência de dados meteorológicos através do barramento 1553 (baixa prioridade).*

<https://www.thinglink.com/scene/657270067448250369>

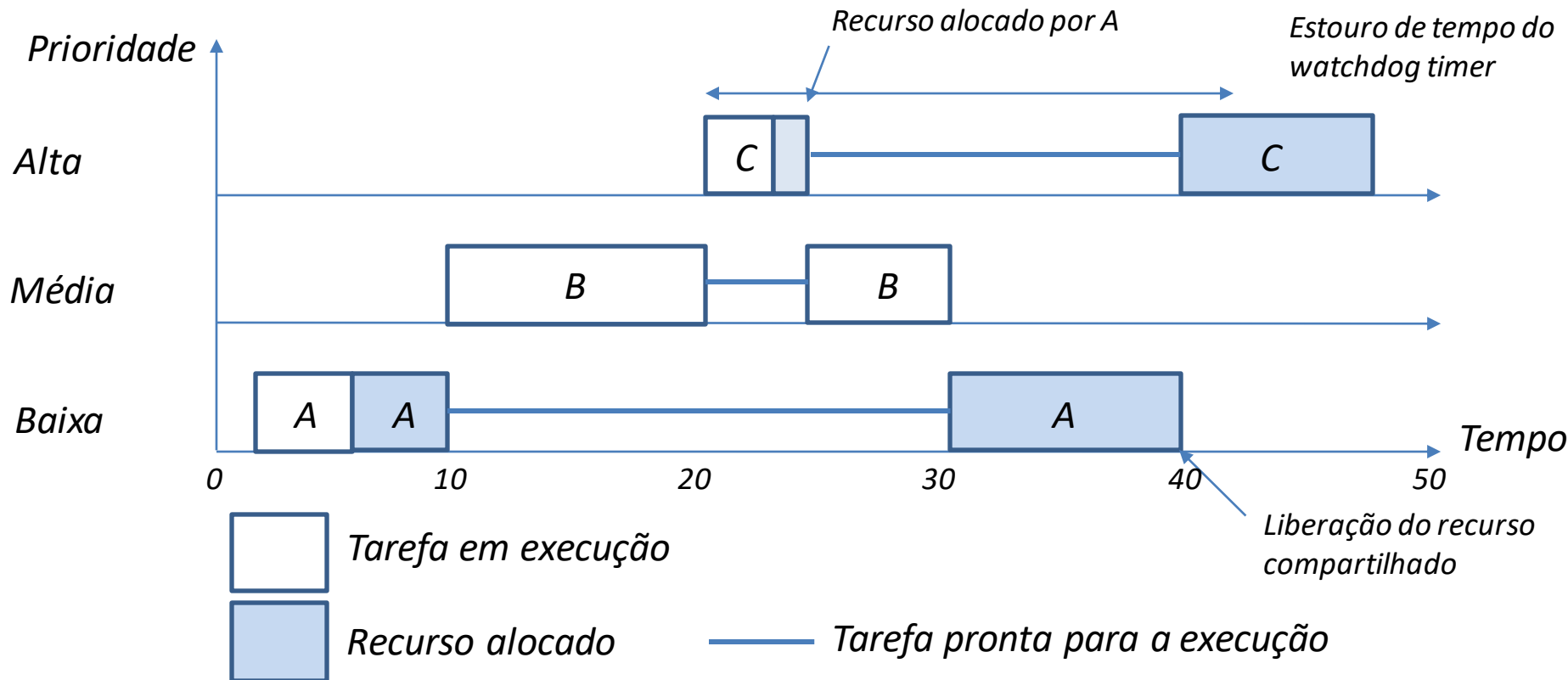


Inversão de prioridade

- A tarefa de alta prioridade possuía um watchdog timer
- Tarefa periódica de controle de barramento 1553 (alta prioridade) – watchdog timer
- Tarefa de comunicação via rádio com a terra (média prioridade)
- Tarefa de transferência de dados meteorológicos através do barramento 1553 (baixa prioridade).

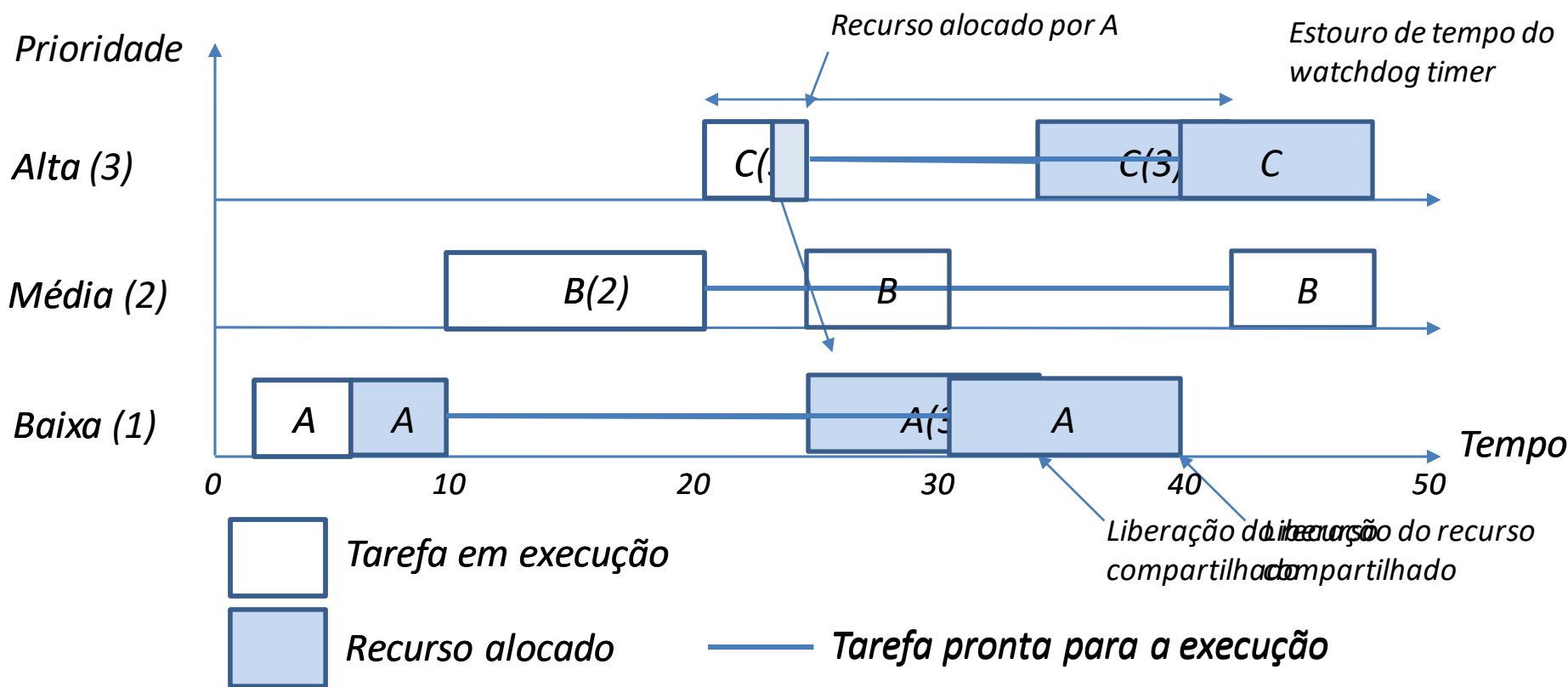
Como resolver esse tipo de problema??

Foi ativado um protocolo de herança de prioridades



Protocolo de Herança de Prioridade

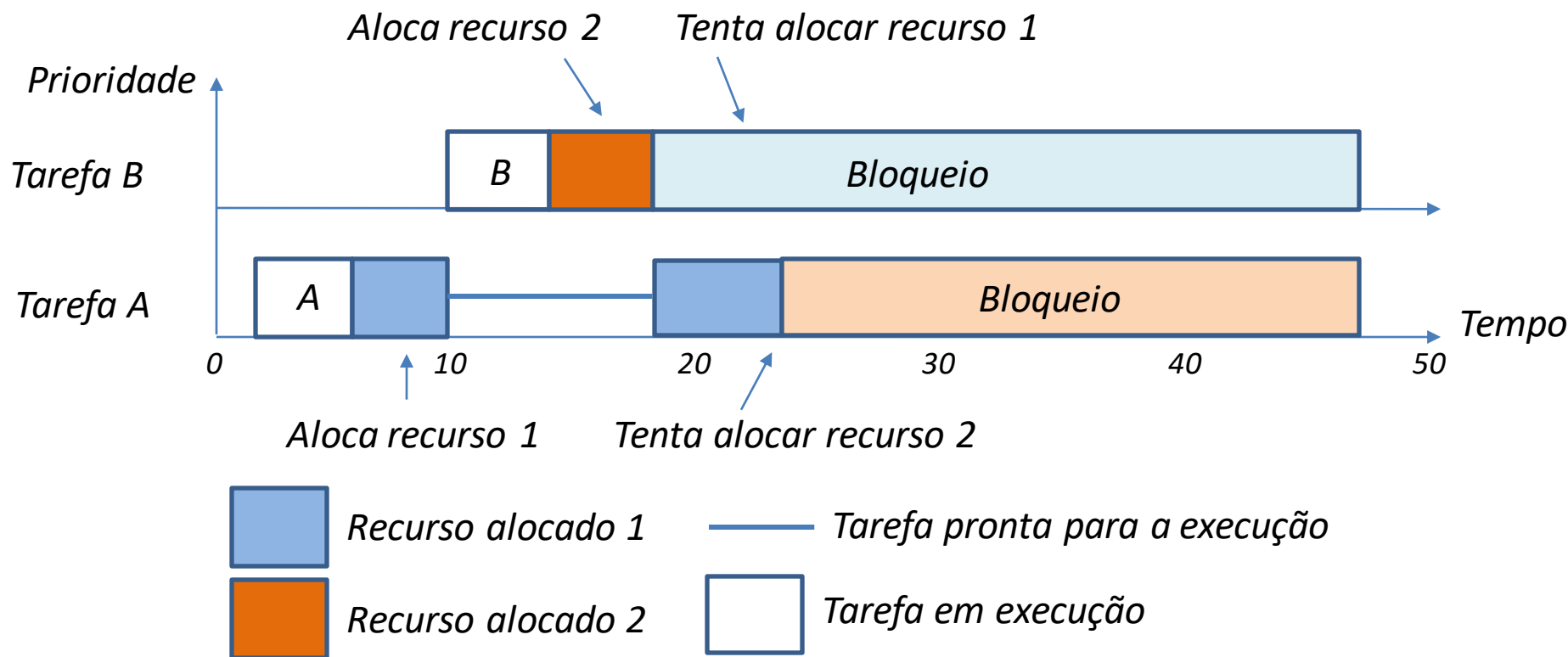
- Quando uma tarefa adquire o recurso compartilhado, a sua prioridade automaticamente é alterada para o mesmo nível da tarefa de mais alta prioridade que tenta acessar o mesmo recurso.
- Com isso, a tarefa possui uma prioridade estática e uma prioridade dinâmica.



- *Quando uma tarefa adquire o recurso compartilhado, a sua prioridade automaticamente é alterada para o nível máximo entre todas as tarefas que possam utilizar esse recurso, até que o recurso seja liberado.*

Deadlock (impasse)

- *Ocorre quando duas ou mais tarefas precisam de recursos compartilhados que já estão adquiridos pela outra e acaba gerando um bloqueio infinito.*



- *Possíveis soluções*
 - *Adquirir previamente todos os recursos que serão utilizados na execução da tarefa.*
 - *Especificar um timeout na tentativa de alocação de um recurso, isso faz com que a tarefa desista da alocação e libere o recurso que estava utilizando.*

- *Overload ou sobrecarga acontece quando o processador e/ou o sistema operacional não consegue executar as tarefas nos requisitos de tempo solicitados.*
 - *Nessa situação, as tarefas de baixa prioridade correm o risco de nunca serem executadas*
 - *Solução: Otimização em tempo de execução ou troca/upgrade do hardware e/ou software.*

■ Utilizando Semáforos

- Semáforos são utilizados para gerenciar recursos compartilhados
- `#include <semphr.h>`
- Iniciar o handler do semáforo antes de criar as tarefas

<https://www.freertos.org/a00113.html>

■ Utilizando Semáforos

<https://www.freertos.org/a00113.html>

■ *Criando um semáforo Binário:*

```
SemaphoreHandle = xSemaphoreCreateBinary();
```

```
SemaphoreHandle_t xSemaphore;
```

```
void vATask( void * pvParameters )
{
    /* Attempt to create a semaphore. */
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL )
    {
        /* There was insufficient FreeRTOS heap available for the semaphore to
        be created successfully. */
    }
    else
    {
        /* The semaphore can now be used. Its handle is stored in the
        xSemaphore variable. Calling xSemaphoreTake() on the semaphore here
        will fail until the semaphore has first been given. */
    }
}
```

■ *Deletando um semáforo (para liberar memória)*

```
vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```


- Para obter/devolver um semáforo

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

```
/* A task that uses the semaphore. */  
void vAnotherTask( void * pvParameters )  
{
```

```
    /* ... Do other things. */
```

```
    if( xSemaphore != NULL )  
    {
```

```
        /* See if we can obtain the semaphore. If the semaphore is not  
        available wait 10 ticks to see if it becomes free. */
```

```
        if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )  
        {
```

Caso o semáforo seja obtido com sucesso, é feito o processamento utilizando o recurso protegido e ao final ele deve ser devolvido.

```
            xSemaphoreGive( xSemaphore );
```

```
        }  
    }  
    else  
    {
```

```
        /* We could not obtain the semaphore and can therefore not access  
        the shared resource safely. */
```

```
    }
```

```
}
```

```
}
```

Tempo de espera (*timeout*) para obter o semáforo. Utilize portMAX_DELAY se quiser aguardar infinitamente.

pdMS_TO_TICKS(ms)

Para devolver o semáforo (liberar o recurso)

Semáforos - Binário

- Exemplo 1: Crie duas tarefas, a primeira tarefa faz a leitura de uma entrada AD, obtém o “semaforo1” (semáforo binário) e imprime o valor na serial apenas quando o semáforo for obtido. A segunda tarefa é um LED (pino 7) piscando em 0,5Hz que libera um semáforo binário “semaforo1”. Ou seja, a contagem de tempo existirá apenas na tarefa 2. [Arquivo no Moodle](#)

```
#include "Arduino_FreeRTOS.h"
#include "task.h"
#include "semphr.h"

#define LED1 7
#define ENTRADA A0

/* Variáveis para armazenamento do handle das tasks */
TaskHandle_t xTarefaADCHandle;
TaskHandle_t xTarefa2Handle;

SemaphoreHandle_t xSemaforo1;

/*protótipos das Tasks*/
void vTarefaADC (void *pvParameters);
void vTarefa2 (void *pvParameters);
```

Exemplo 1:

```
void setup() {
    Serial.begin(9600);
    xSemaforol = xSemaphoreCreateBinary();
    xTaskCreate(vTarefaADC, "vTarefaADC", 128, NULL, 3, &xTarefaADCHandle);
    xTaskCreate(vTarefa2, "vTarefa2", 128, NULL, 1, &xTarefa2Handle);
}

void loop() {
    //As funções são executadas nas tarefas
}

void vTarefaADC(void *pvParameters){
    int Valor_ADC;
    while(1)
    {
        xSemaphoreTake(xSemaforol, portMAX_DELAY);
        Valor_ADC = analogRead(ENTRADA);
        Serial.println("Valor ADC: " + String(Valor_ADC));
    }
}

void vTarefa2(void *pvParameters){
    pinMode(LED1, OUTPUT);
    while(1)
    {
        digitalWrite(LED1, !digitalRead(LED1));
        Serial.println("LED: " + String(digitalRead(LED1)));
        vTaskDelay(pdMS_TO_TICKS(1000));
        xSemaphoreGive(xSemaforol);
    }
}
```

O semáforo é adquirido pela primeira vez após a tarefa 2 liberar o semáforo (inicia bloqueado), e após isso, só pode ser adquirido novamente quando for liberado pela vTarefa2 (isso só é possível no semáforo binário, o mutex deve ser liberado pela mesma tarefa que adquiriu).

Dessa forma, a tarefa vTarefaADC fica sincronizada com a vTarefa2 usando o semáforo.

Semáforos com interrupção

- Para obter/devolver um semáforo dentro de interrupções

```
xSemaphoreTakeFromISR(SemaphoreHandle_t xSemaphore,  
signed BaseType_t *pxHigherPriorityTaskWoken)
```

Setando pdTRUE, a tarefa tem prioridade em obter o semáforo caso ele já esteja sob domínio de outra tarefa

```
xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,  
signed BaseType_t *pxHigherPriorityTaskWoken)
```

Deve setar pdTRUE caso necessite liberar uma tarefa de maior prioridade que esteja em bloqueio por causa do recurso, por ex. uma tarefa que faria o tratamento da interrupção. Caso contrário, deixe NULL. A troca de contexto deve ser garantida (manual).

```
portYIELD_FROM_ISR();
```

Semáforos com interrupção

- Exemplo 2: Refaça o exercício anterior, no entanto, o semáforo será liberado dentro de uma rotina de interrupção, quando um botão for pressionado, ou seja, fará uma aquisição sempre que o botão for pressionado.

```
#define LED1 7
#define ENTRADA A0
#define BOTAO 2

/* Variáveis para armazenamento do handle das tasks */
TaskHandle_t xTarefaADCHandle;
TaskHandle_t xTarefa2Handle;

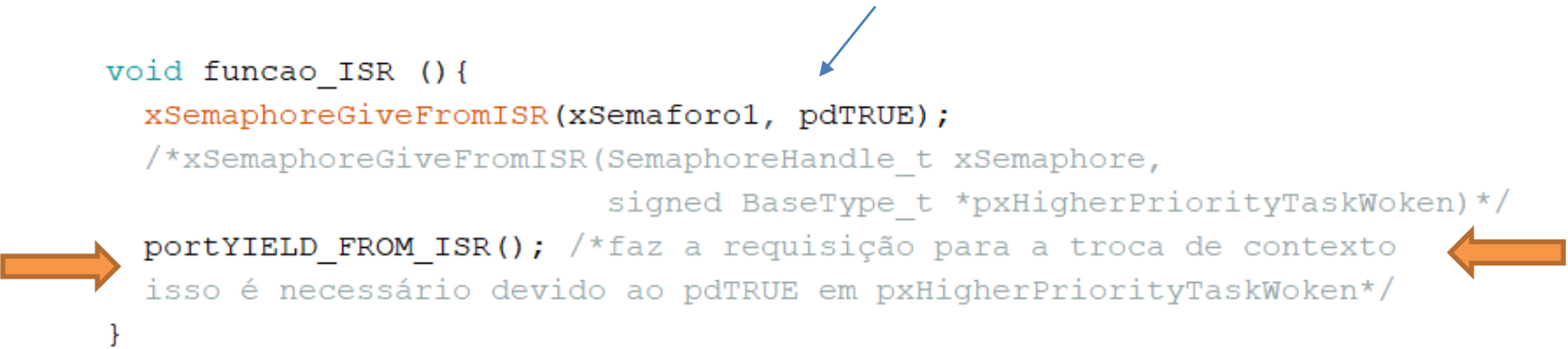
SemaphoreHandle_t xSemaforol;

/*protótipos das Tasks*/
void vTarefaADC (void *pvParameters);
void vTarefa2 (void *pvParameters);

void funcao_ISR () {
    xSemaphoreGiveFromISR(xSemaforol, pdTRUE);
    /*xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,
                           signed BaseType_t *pxHigherPriorityTaskWoken)*/
    portYIELD_FROM_ISR(); /*faz a requisição para a troca de contexto
    isso é necessário devido ao pdTRUE em pxHigherPriorityTaskWoken*/
}
```

Complete o resto do código

Utilizando o pdTRUE está fazendo uma rápida liberação da tarefa que está bloqueada pelo semáforo (nesse caso, a função vTarefaADC, no entanto, é necessário chamar a função portYIELD_FROM_ISR para garantir uma rápida troca de contexto



```

void setup() {
    Serial.begin(9600);
    pinMode(BOTAO, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(BOTAO), funcao_ISR, FALLING);
    xSemaforo1 = xSemaphoreCreateBinary();
    xTaskCreate(vTarefaADC, "vTarefaADC", 256, NULL, 2, &xTarefaADCHandle);
    xTaskCreate(vTarefa2, "vTarefa2", 128, NULL, 1, &xTarefa2Handle);
}

void loop() {
    //As funções são executadas nas tarefas
}

void vTarefaADC(void *pvParameters){
    int Valor_ADC;
    while(1)
    {
        xSemaphoreTake(xSemaforo1, portMAX_DELAY);
        Valor_ADC = analogRead(ENTRADA);
        Serial.println("Valor ADC: " + String(Valor_ADC));
    }
}

void vTarefa2(void *pvParameters){
    pinMode(LED1, OUTPUT);
    while(1)
    {
        digitalWrite(LED1, !digitalRead(LED1));
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

```

- Utilizando Semáforos

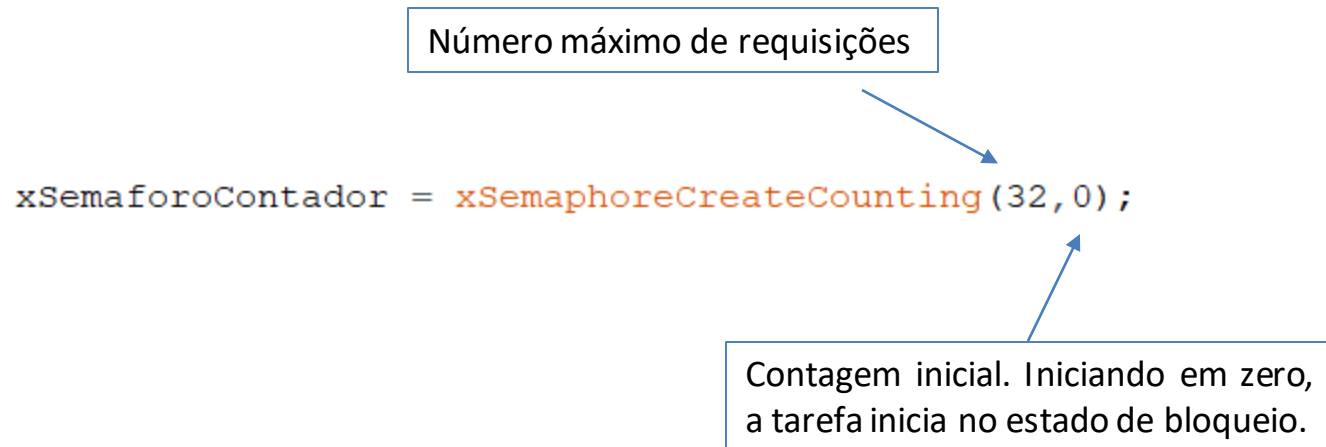
- *Criando um semáforo Contador:*

```
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount,  
                                             UBaseType_t uxInitialCount);
```

```
void vATask( void * pvParameters )  
{  
    SemaphoreHandle_t xSemaphore;  
  
    /* Create a counting semaphore that has a maximum count of 10 and an  
    initial count of 0. */  
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );  
  
    if( xSemaphore != NULL )  
    {  
        /* The semaphore was created successfully. */  
    }  
}
```

Semáforo contador

- Com o semáforo binário, temos que ele só pode ser adquirido uma vez. No entanto, caso a interrupção fosse ativada 10 vezes em um intervalo que o processador só consegue processar uma amostra, as outras 9 requisições seriam perdidas. Nesse caso, o ideal é utilizar o semáforo contador, que terá um número X de requisições “tickets” que a interrupção poderá enfileirar.

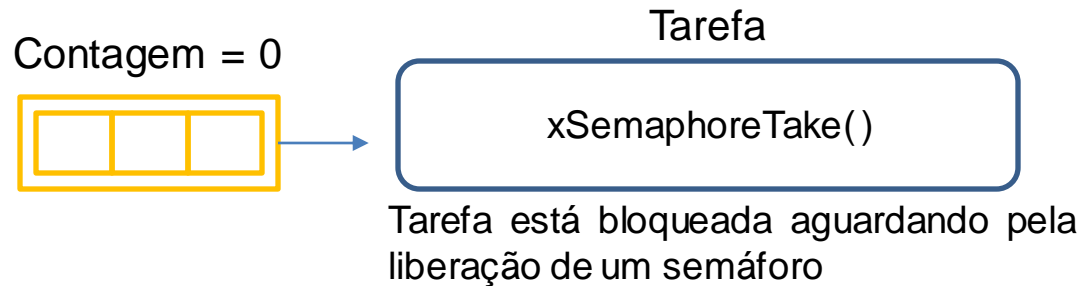


- Como saber o “ticket” atual em execução:

```
UBaseType_t ticket_semaforo;  
ticket_semaforo = uxSemaphoreGetCount(SemaphoreHandle_t);
```


Semáforo contador

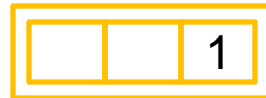
```
xSemaforoContador = xSemaphoreCreateCounting(3,0);
```



Interrupção

xSemaphoreGiveFromISR()

Contagem = 1



Tarefa

xSemaphoreTake()

A tarefa é desbloqueada para processamento. Após o processamento, a fila é zerada e a tarefa volta para o estado de bloqueio

Interrupção

xSemaphoreGiveFromISR()

Contagem = 1



Tarefa

xSemaphoreTake()

A tarefa é desbloqueada para processamento. Após o processamento do primeiro evento, a tarefa executa o segundo evento da fila, e assim sucessivamente até zerar a fila e a tarefa voltar para o estado de bloqueio

Semáforo contador

- Exemplo 3: Faça o código do exemplo 2 utilizando um semáforo contador com tamanho 32. Imprima na serial a requisição sendo processada no momento e também o valor atual do AD. Coloque um delay de 1000ms na tarefa do AD.

```
/* Variáveis para armazenamento do handle das tasks */
TaskHandle_t xTarefaADCHandle;
TaskHandle_t xTarefa2Handle;

SemaphoreHandle_t xSemaforoContador;

/*protótipos das Tasks*/
void vTarefaADC (void *pvParameters);
void vTarefa2 (void *pvParameters);

void funcao_ISR () {
    xSemaphoreGiveFromISR(xSemaforoContador, pdTRUE);
    /*xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,
                           signed BaseType_t *pxHigherPriorityTaskWoken)*/
    portYIELD_FROM_ISR(); /*faz a requisição para a troca de contexto
    isso é necessário devido ao pdTRUE em pxHigherPriorityTaskWoken*/
}
```

Faça o teste clicando várias vezes no botão e analisando como ele vai fazendo o tratamento das requisições.

Semáforo contador

```
void setup() {  
    Serial.begin(9600);  
    pinMode(BOTAO, INPUT_PULLUP);  
    attachInterrupt(digitalPinToInterrupt(BOTAO), funcao_ISR, FALLING);  
    xSemaforoContador = xSemaphoreCreateCounting(32, 0);  
    xTaskCreate(vTarefaADC, "vTarefaADC", 256, NULL, 2, &xTarefaADCHandle);  
    xTaskCreate(vTarefa2, "vTarefa2", 128, NULL, 1, &xTarefa2Handle);  
}
```

```
void loop() {  
    //As funções são executadas nas tarefas  
}
```

```
void vTarefaADC(void *pvParameters){  
    UBaseType_t ticket_semaforo;  
    int Valor_ADC;  
    while(1)  
    {  
        xSemaphoreTake(xSemaforoContador, portMAX_DELAY);  
        Valor_ADC = analogRead(ENTRADA);  
        Serial.print("Requisições faltantes: ");  
        ticket_semaforo = uxSemaphoreGetCount(xSemaforoContador);  
        Serial.print(ticket_semaforo);  
        Serial.println(" Valor ADC atual: " + String(Valor_ADC));  
        vTaskDelay(pdMS_TO_TICKS(1000));  
    }  
}  
  
void vTarefa2(void *pvParameters){  
    pinMode(LED1, OUTPUT);  
    while(1)  
    {  
        digitalWrite(LED1, !digitalRead(LED1));  
        vTaskDelay(pdMS_TO_TICKS(500));  
    }  
}
```

Faça o teste clicando várias vezes no botão e analisando como ele vai fazendo o tratamento das requisições.

```
12:22:53.279 -> Requisições faltantes: 6 Valor ADC atual: 604  
12:22:54.359 -> Requisições faltantes: 5 Valor ADC atual: 604  
12:22:55.490 -> Requisições faltantes: 4 Valor ADC atual: 604  
12:22:56.613 -> Requisições faltantes: 3 Valor ADC atual: 604  
12:22:57.739 -> Requisições faltantes: 2 Valor ADC atual: 603  
12:22:58.817 -> Requisições faltantes: 1 Valor ADC atual: 603  
12:22:59.938 -> Requisições faltantes: 0 Valor ADC atual: 604
```

Semáforo contador

- Exercício 1: A partir do exemplo 3, teste se o semáforo foi adquirido no tempo de 1 segundo, caso contrário, imprima “nenhuma requisição solicitada”.

```
if (xSemaphoreTake(xSemaforo_Handle, tempo_de_espera) == pdTRUE)
{
    //Faça algo
}

void vTarefaADC(void *pvParameters){
    UBaseType_t ticket_semaforo;
    int Valor_ADC;
    while(1)
    {
        if (xSemaphoreTake(xSemaforoContador, pdMS_TO_TICKS(1000)) == pdTRUE)
        {
            Valor_ADC = analogRead(ENTRADA);
            Serial.print("Requisições faltantes: ");
            ticket_semaforo = uxSemaphoreGetCount(xSemaforoContador);
            Serial.print(ticket_semaforo);
            Serial.println(" Valor ADC atual: " + String(Valor_ADC));
            vTaskDelay(pdMS_TO_TICKS(1000));
        }
        else
        {
            Serial.println("Nenhuma requisição solicitada");
        }
    }
}
```

- Utilizando Semáforos

- *Criando um semáforo Mutex: Possuem protocolo de herança de prioridade*

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

```
SemaphoreHandle_t xSemaphore;
```

```
void vATask( void * pvParameters )
```

```
{
```

```
    /* Create a mutex type semaphore. */
```

```
    xSemaphore = xSemaphoreCreateMutex();
```

```
    if( xSemaphore != NULL )
```

```
    {
```

```
        /* The semaphore was created successfully and  
        can be used. */
```

```
    }
```

```
}
```

Semáforo MUTEX

- A principal e mais importante diferença do semáforo MUTEX é ser mutuamente exclusivo. Ou seja, se a tarefa1 adquire ele, apenas a tarefa1 poderá devolvê-lo. Isso é bastante útil porque garante a segurança de que um determinado recurso não seja liberado por outra tarefa em paralelo antes que ele tenha desempenhado completamente sua tarefa.
- Alguns cuidados que devem ser tomados:
 - *Prioridades: Escolha corretamente a prioridade das tarefas que disputam pelo mesmo semáforo, uma escolha inadequada por resultar com que apenas a tarefa de maior prioridade assuma o controle do semáforo.*
 - *Não utilize mais de um semáforo para um mesmo recurso compartilhado.*
 - *Utilize a macro portMAX_DELAY apenas em caso de necessidade real, caso contrário poderá gerar um deadlock.*

Semáforo MUTEX

- Exemplo 4: Faça 2 tarefas que imprimam na serial o momento em que adquiram e em que liberam o MUTEX.

```
/* Variáveis para armazenamento do handle das tasks */
TaskHandle_t xTarefa1Handle;
TaskHandle_t xTarefa2Handle;

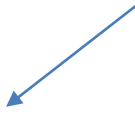
SemaphoreHandle_t xSemaforoMUTEX;

/*protótipos das Tasks*/
void vTarefa1 (void *pvParameters);
void vTarefa2 (void *pvParameters);

void setup() {
    Serial.begin(9600);
    xSemaforoMUTEX = xSemaphoreCreateMutex();
    xTaskCreate(vTarefa1, "vTarefa1",256, NULL, 1, &xTarefa1Handle);
    xTaskCreate(vTarefa2, "vTarefa2",256, NULL, 3, &xTarefa2Handle);
}

void loop() {
    //As funções são executadas nas tarefas
}
```

Se alterar a prioridade, vai alterar a ordem inicial de execução



Semáforo MUTEX

Exemplo 4:

```
void vTarefa1(void *pvParameters){
    while(1)
    {
        if (xSemaphoreTake(xSemaforoMUTEX, portMAX_DELAY) == pdTRUE){
            Serial.println("Tarefa_1 adquirindo o semáforo");
            vTaskDelay(pdMS_TO_TICKS(500));
            Serial.println("Tarefa_1 liberando o semáforo");
            xSemaphoreGive(xSemaforoMUTEX);
        }
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}

void vTarefa2(void *pvParameters){
    while(1)
    {
        if (xSemaphoreTake(xSemaforoMUTEX, portMAX_DELAY) == pdTRUE){
            Serial.println("Tarefa_2 adquirindo o semáforo");
            vTaskDelay(pdMS_TO_TICKS(500));
            Serial.println("Tarefa_2 liberando o semáforo");
            xSemaphoreGive(xSemaforoMUTEX);
        }
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}
```

Aguarda indefinidamente para adquirir o semáforo, isso pode gerar deadlock



- Exemplo 4: Teste o comportamento da serial com e sem o MUTEX

Com o MUTEX

```
-> Tarefa_1 adquirindo o semáforo  
-> Tarefa_1 liberando o semáforo  
-> Tarefa_2 adquirindo o semáforo  
-> Tarefa_2 liberando o semáforo  
-> Tarefa_1 adquirindo o semáforo  
-> Tarefa_1 liberando o semáforo  
-> Tarefa_2 adquirindo o semáforo  
-> Tarefa_2 liberando o semáforo  
-> Tarefa_1 adquirindo o semáforo  
-> Tarefa_1 liberando o semáforo  
-> Tarefa_2 adquirindo o semáforo  
-> Tarefa_2 liberando o semáforo
```

Sem o MUTEX (comente os xSemaphoreTake)

```
-> Tarefa_2 adquirindo o semáforo  
-> Tarefa_1 adquirindo o semáforo  
-> Tarefa_2 liberando o semáforo  
-> Tarefa_1 liberando o semáforo  
-> Tarefa_2 adquirindo o semáforo  
-> Tarefa_1 adquirindo o semáforo  
-> Tarefa_2 liberando o semáforo  
-> Tarefa_1 liberando o semáforo
```

■ Informações extras

```
xSemaphoreCreateRecursiveMutex()
```

- Utilizado para criar um MUTEX recursivo (a mesma tarefa pode adquirir várias vezes)
- Setar configUSE_RECURSIVE_MUTEXES em '1' no FreeRTOSConfig.h (Arduino é 1 por padrão)
- Se a tarefa adquirir (take) o semáforo 5 vezes, ela precisará liberar (give) exatamente 5 vezes

```
xSemaphoreTakeRecursive()  
xSemaphoreGiveRecursive()
```

- Utilizado quando o desenvolvedor quer escolher a quantidade de memória alocada pelo MUTEX.

```
xSemaphoreCreateMutexStatic()  
xSemaphoreCreateRecursiveMutexStatic()
```

- Para maiores detalhes específicos de cada PORT, é importante consultar o arquivo semphr.h do FreeRTOS.

Referências

- DENARDIN, G. W.; BARRIQUELLO, C. H. Sistemas Operacionais de Tempo Real e sua Aplicação em Sistemas Embarcados. 1ª edição, São Paulo, Blucher, 2019.
- DENARDIN, G. W. Notas de aula de sistemas embarcados. UTFPR.
- STALLINGS, William. Operating systems: internals and design principles. 5.ed. Upper Saddle River: Pearson Prentice Hall. 2004.
- TANENBAUM, Andrew. Sistemas operacionais modernos. Rio de Janeiro: LTC. 1999.
- BACURAU, R.M. Notas de aulas de projeto de sistemas embarcados. UNICAMP, 2020. Disponível em: <https://sites.google.com/site/rodrigobacurau/cursos-2020-1/es670---projeto-de-sistemas-embarcados>
- What Really Happened On Mars <https://www.microsoft.com/en-us/research/people/mbj/just-for-fun/>
- <https://www.freertos.org/>