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

Prof. Hugo Vieira Neto

2020/2

Recursos Compartilhados

- Acessos a recursos compartilhados que causem modificações em seus estados precisam ser protegidos, de forma a garantir exclusão mútua em ambiente preemptivo
- Em um RTOS, essa proteção pode ser realizada com objetos do tipo mutex
  - Obtenção antes de entrar na seção crítica
  - Devolução após sair da seção crítica

Estudo de Caso

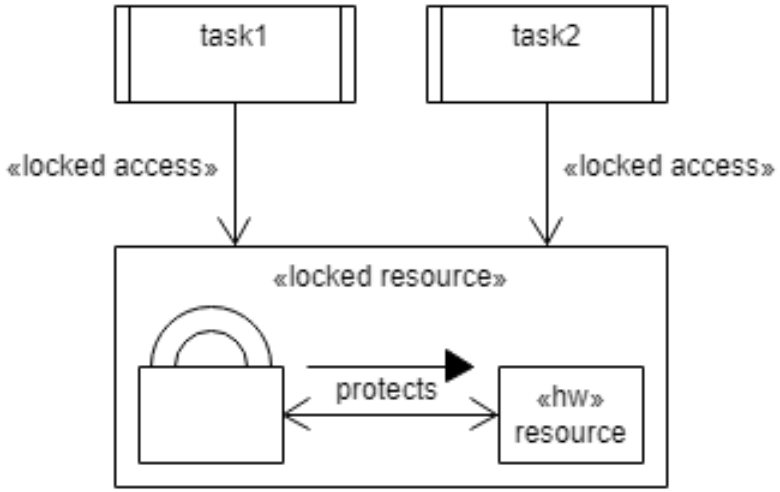
- As tarefas para acionamento individual dos LEDs no programa “tarefas”, **da forma como estão implementadas**, possuem um problema em potencial: uma condição de corrida
- Apesar de acionarem LEDs **diferentes** do kit, a porta de GPIO do microcontrolador aos quais estes estão conectados é a **mesma** (port N)
  - Os acionamentos dos LEDs mudam os estados do **mesmo** registrador de controle da porta de GPIO!

Por que o problema não se manifesta?

1. As tarefas possuem tempos de computação extremamente curtos e longos períodos de ativação ($C << T$)
2. Ainda, as tarefas possuem prioridades iguais e a política de escalonamento Round Robin não chega a atuar ($C << Tq$)
3. Ou seja, na prática não há possibilidade de preempção de uma tarefa pela outra durante a execução da seção crítica

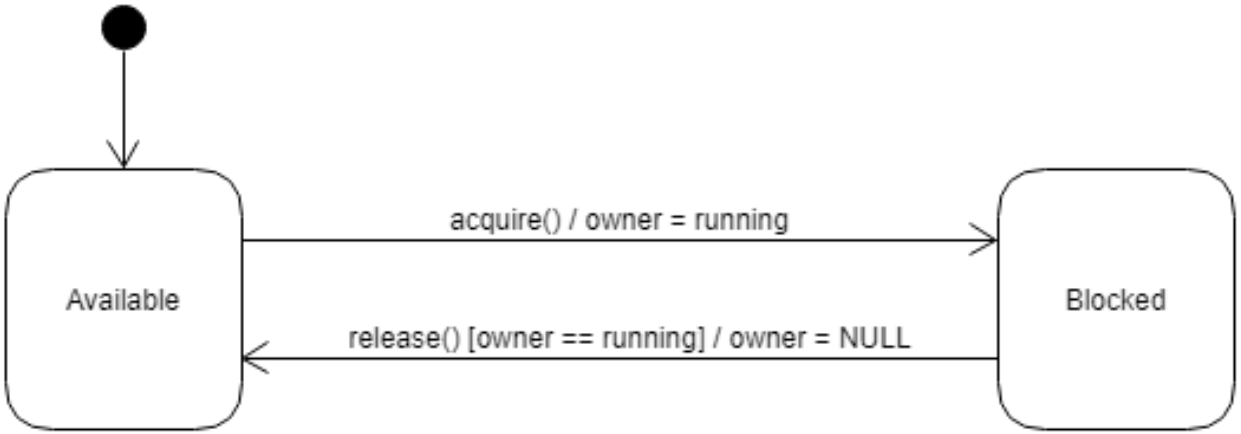
Mutexes

- Um mutex é um tipo especial de semáforo binário para acesso protegido a recursos compartilhados (exclusão mútua)
  - Memória, periféricos ou canais de comunicação



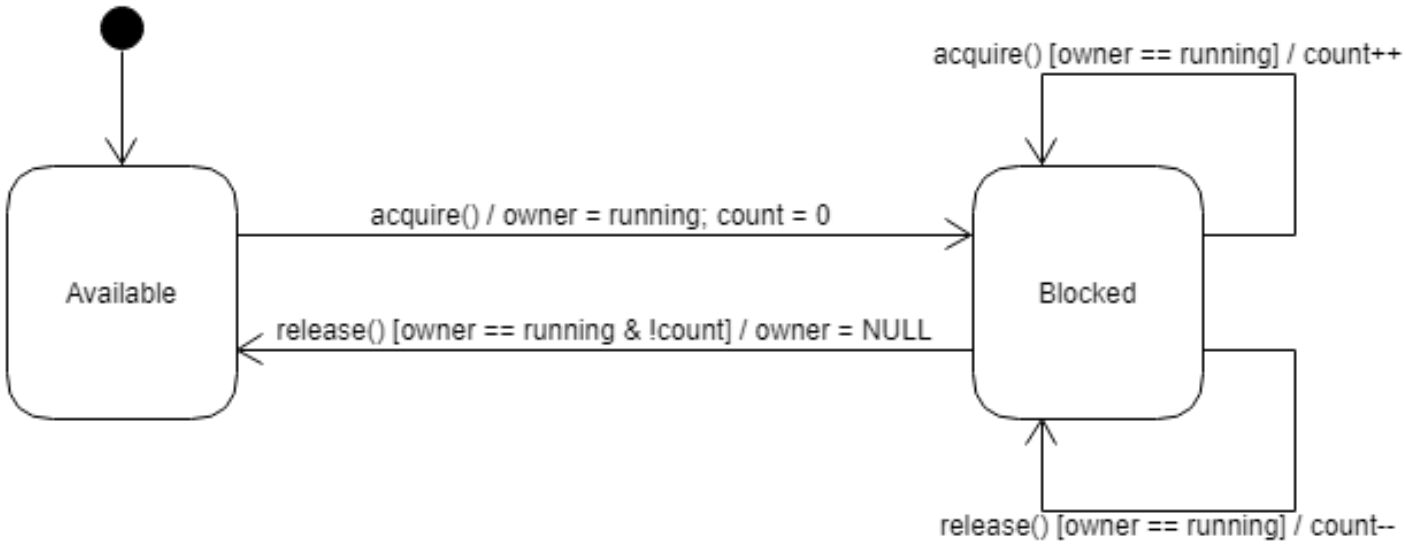
Estados Possíveis

- Um mutex pode estar “liberado” ou “bloqueado” por uma tarefa
  - Apenas a tarefa que bloqueou o mutex (tarefa proprietária) pode liberá-lo



Opções de Comportamento

- Recursivo
  - Uma mesma tarefa pode obter um mesmo mutex múltiplas vezes sem se autobloquear



Opções de Comportamento

- Herança de prioridade
  - A prioridade da tarefa proprietária de um mutex é aumentada **temporariamente** para a prioridade de outra tarefa bloqueada pelo mesmo mutex
- Robusto
  - O mutex será automaticamente liberado quando a sua tarefa proprietária for terminada

Gerenciamento de Mutexes

```
osMutexId_t osMutexNew(const osMutexAttr_t *attr)
```

- Cria e inicializa um mutex

```
osStatus_t osMutexAcquire(osMutexId_t mutex_id, uint32_t timeout)
```

- Bloqueia um mutex (se necessário, aguarda por um determinado limite de tempo)

```
osStatus_t osMutexRelease(osMutexId_t mutex_id)
```

- Libera um mutex que foi bloqueado

Gerenciamento de Mutexes

```
const char *osMutexGetName(osMutexId_t
mutex_id)
```

- Obtém o nome de um mutex

```
osThreadId_t osMutexGetOwner(osMutexId_t
mutex_id)
```

- Obtém o identificador da tarefa proprietária de um mutex

```
osStatus_t osMutexDelete(osMutexId_t
mutex_id)
```

- Deleta um mutex

Uso de Mutexes

1. Definição de um identificador para o mutex (variável global)

```
osMutexId_t mutex_id;
```

2. Definição de uma estrutura de atributos para o mutex (opcional)

```
const osMutexAttr_t Mutex_attr = {
  "myMutex",
  osMutexRecursive | osMutexPrioInherit,
  NULL, // memory for control block
  0U    // size of provided memory
};
```

Uso de Mutexes

3. Criação do mutex a partir da função `main`

```
mutex_id = osMutexNew(&Mutex_attr);
```

ou, para os atributos padrão

```
mutex_id = osMutexNew(NULL);
```

4. Uso do mutex a partir de uma tarefa

```
osMutexAcquire(mutex_id, osWaitForever);
// seção crítica
osMutexRelease(mutex_id);
```

Sem limite de tempo

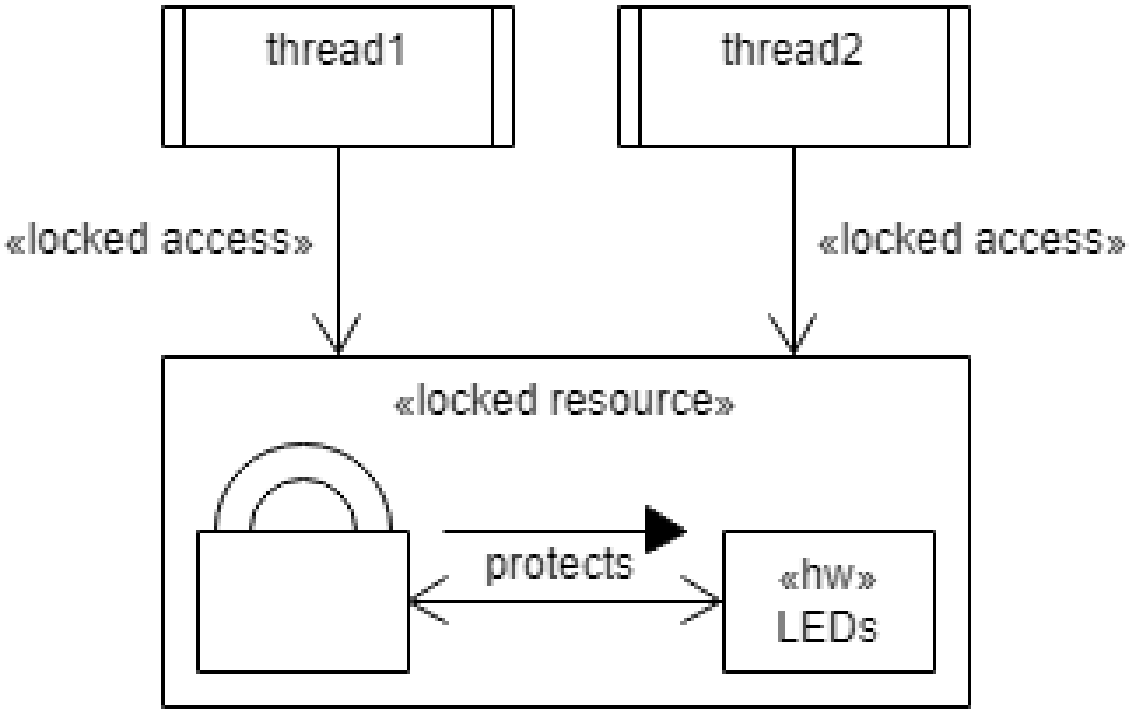
Estruturas de Atributos

- Há tipos para estruturas de atributos para todos os tipos de objeto do RTX:
  - `osThreadAttr_t` para tarefas
  - `osMutexAttr_t` para mutexes
  - `osSemaphoreAttr_t` para semáforos
  - `osEventFlagsAttr_t` para sinalizadores de eventos
  - `osMessageQueueAttr_t` para filas de mensagem
  - `osTimerAttr_t` para temporizadores
  - `osMemoryPoolAttr_t` para pools de memória

Exercício 3

1. Abra o projeto “tarefas” a partir da área de trabalho “EK-TM4C1294_RTOS_IAR8”.
2. Modifique o código-fonte no arquivo `tarefas.c` para garantir exclusão mútua (evitando condições de corrida) no acesso aos LEDs pelas tarefas `thread1` e `thread2`.
3. Aproveite para criar atributos com nomes para cada tarefa (ver documentação do RTX)

Diagrama de Objetos com Exclusão Mútua no Acesso aos LEDs



Exercício 4

1. Abra o projeto “tarefas” a partir da área de trabalho “EK-TM4C1294_RTOS_IAR8”.
2. Modifique o código-fonte no arquivo `tarefas.c` para que as duas tarefas periódicas sejam temporizadas com `osDelayUntil()`.
3. Crie dois mutexes diferentes para e utilize-os nas tarefas da forma descrita a seguir.

Exercício 4

Dois mutexes aninhados em ordem inversa nas tarefas:

```
void thread1(void *arg){
  uint8_t state = 0;
  uint32_t tick;
  while(1){
    osMutexAcquire(mutex1_id,
                   osWaitForever);
    tick = osKernelGetTickCount();
    state ^= LED1;
    osMutexAcquire(mutex2_id,
                   osWaitForever);
    LEDWrite(LED1, state);
    osMutexRelease(mutex2_id);
    osDelayUntil(tick + 100);
    osMutexRelease(mutex1_id);
  } // while
} // thread1
```

```
void thread2(void *arg){
  uint8_t state = 0;
  uint32_t tick;
  while(1){
    osMutexAcquire(mutex2_id,
                   osWaitForever);
    tick = osKernelGetTickCount();
    state ^= LED2;
    osMutexAcquire(mutex1_id,
                   osWaitForever);
    LEDWrite(LED2, state);
    osMutexRelease(mutex1_id);
    osDelayUntil(tick + 100);
    osMutexRelease(mutex2_id);
  } // while
} // thread2
```

Exercício 4

4. Verifique o comportamento do programa quanto ao acionamento dos LEDs D1 e D2 com períodos de ativação das tarefas **ligeiramente** diferentes:
   - $T_1 > T_2$ (ex: $T_1 = 101$, $T_2 = 100$)
   - $T_1 < T_2$ (ex: $T_1 = 100$, $T_2 = 101$)
5. Modifique a utilização dos mutexes nas tarefas da forma descrita a seguir.

Exercício 4

Dois mutexes aninhados na mesma ordem nas tarefas:

```
void thread1(void *arg){
  uint8_t state = 0;
  uint32_t tick;
  while(1){
    osMutexAcquire(mutex2_id,
                   osWaitForever);
    tick = osKernelGetTickCount();
    state ^= LED1;
    osMutexAcquire(mutex1_id,
                   osWaitForever);
    LEDWrite(LED1, state);
    osMutexRelease(mutex1_id);
    osDelayUntil(tick + 100);
    osMutexRelease(mutex2_id);
  } // while
} // thread1
```

```
void thread2(void *arg){
  uint8_t state = 0;
  uint32_t tick;
  while(1){
    osMutexAcquire(mutex2_id,
                   osWaitForever);
    tick = osKernelGetTickCount();
    state ^= LED2;
    osMutexAcquire(mutex1_id,
                   osWaitForever);
    LEDWrite(LED2, state);
    osMutexRelease(mutex1_id);
    osDelayUntil(tick + 100);
    osMutexRelease(mutex2_id);
  } // while
} // thread2
```

Exercício 4

6. Verifique o comportamento do programa quanto ao acionamento dos LEDs D1 e D2 novamente.
7. Quais as conclusões que se pode tirar em cada caso estudado?
   – Utilize as ferramentas de depuração do ambiente de desenvolvimento para investigar o caso!

Conceitos Importantes

- O uso de mutex não impede a preempção de uma tarefa, mas causa o seu bloqueio caso haja tentativa de acesso a um recurso que já esteja sendo usado por outra tarefa
- O desbloqueio de uma tarefa ocorre quando outra tarefa possuidora do recurso libera o acesso ou quando o limite de tempo de espera é atingido (timeout)

Tarefa Guardiã (*Gatekeeper*)

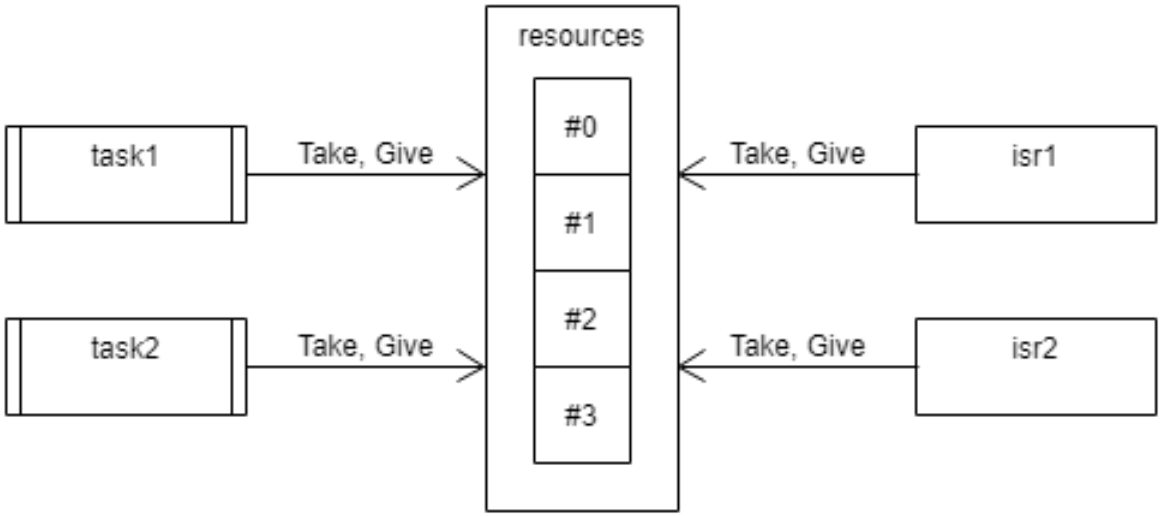
- É a única tarefa que tem acesso a um recurso do sistema (um periférico, por exemplo)
- Qualquer outra tarefa que queira acessar esse recurso terá que solicitar à tarefa guardiã por meio de comunicação entre elas
- Dessa forma, todos os acessos ao recurso são realizados por uma única tarefa e, portanto, não há necessidade de mutex para proteger

Semáforos

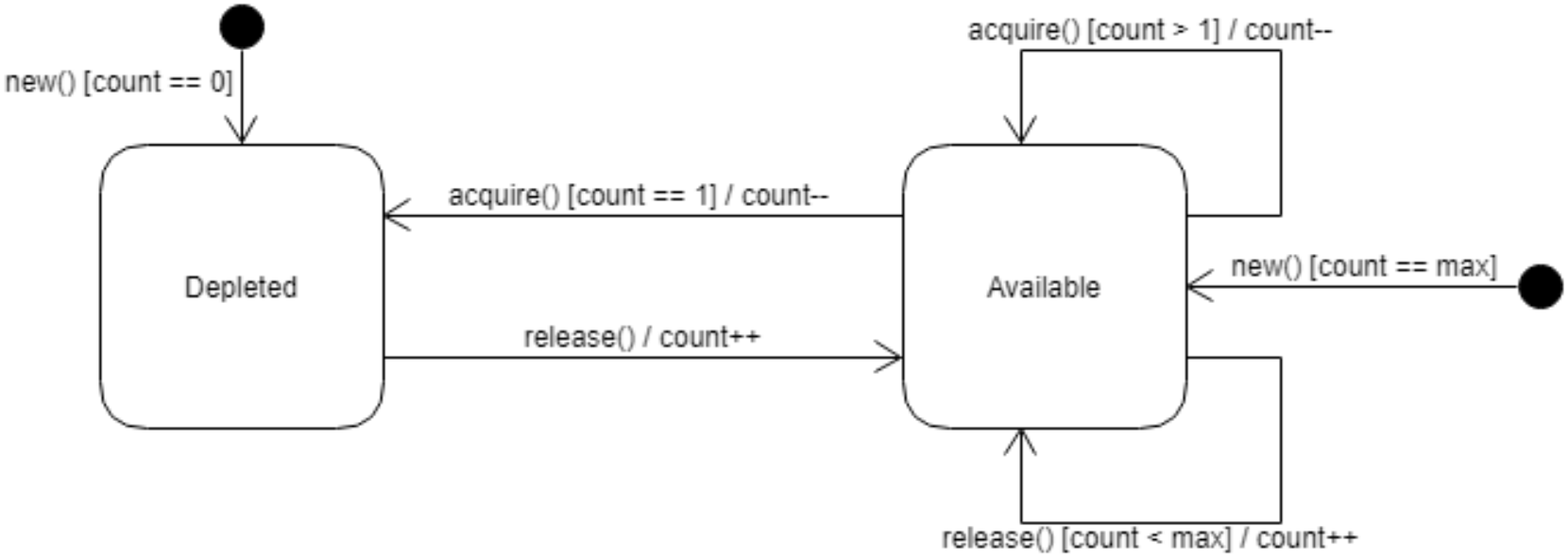
- Semáforos são usados para gerenciar e proteger o acesso a recursos compartilhados
  - Mutex: permite que apenas uma **tarefa** tenha acesso a um **único** recurso compartilhado por vez
  - Semáforo: permite que **tarefas ou ISRs** tenham acesso a um **conjunto** de recursos compartilhados
  - Como um semáforo bloqueado **não** fica atrelado a uma tarefa possuidora, **pode ser utilizado em ISRs**

Semáforos

- Semáforos contadores permitem o controle de acesso a um conjunto de recursos do mesmo tipo, como por exemplo: múltiplos canais de DMA, múltiplas interfaces seriais (UART), etc.



Estados Possíveis



Gerenciamento de Semáforos

```
osSemaphoreId_t osSemaphoreNew
(uint32_t max_count, uint32_t initial_count,
const osSemaphoreAttr_t *attr)
```

- Cria e inicializa um semáforo

```
osStatus_t osSemaphoreAcquire
(osSemaphoreId_t semaphore_id, uint32_t
timeout)
```

- Adquire um passe do semáforo (se necessário, aguarda por um determinado limite de tempo)*

```
osStatus_t osSemaphoreRelease
(osSemaphoreId_t semaphore_id)
```

- Devolve um passe ao semáforo até o limite máximo de contagem*

Gerenciamento de Semáforos

```
const char *osSemaphoreGetName
(osSemaphoreId_t semaphore_id)
```

- Obtém o nome de um semáforo

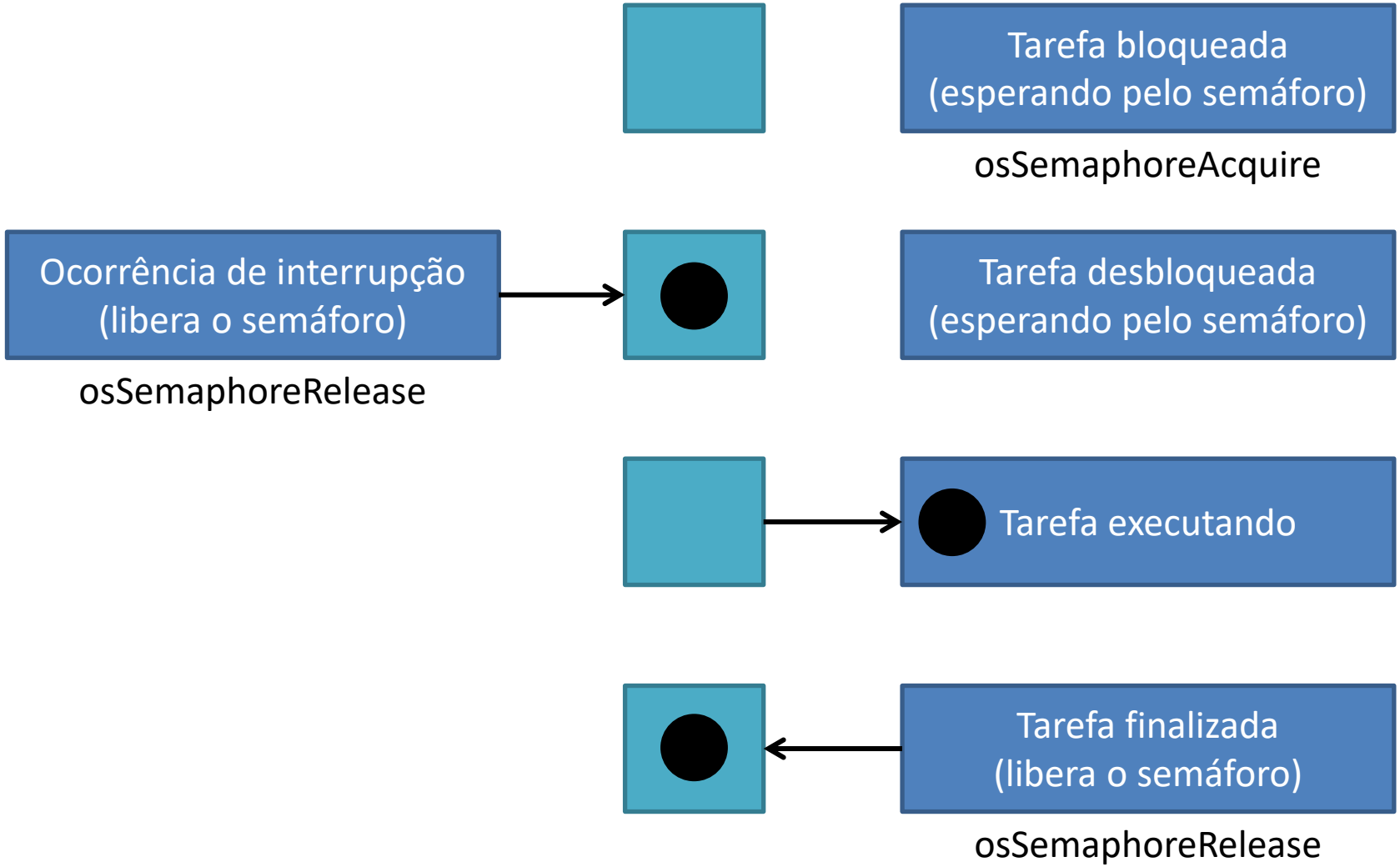
```
uint32_t osSemaphoreGetCount(osSemaphoreId_t
semaphore_id)
```

- Obtém a contagem corrente de passes do semáforo*

```
osStatus_t osSemaphoreDelete(osSemaphoreId_t
semaphore_id)
```

- Deleta um semáforo

Exemplo: Semáforo Binário



Uso de Semáforos

1. Definição de um identificador para o semáforo (variável global)

```
osSemaphoreId_t sema_id;
```

2. Criação do semáforo a partir da função `main`

```
sema_id = osSemaphoreNew(max, ini, NULL);
```

3. Uso do semáforo a partir de uma tarefa

```
osSemaphoreAcquire(sema_id, osWaitForever);
// seção crítica
osSemaphoreRelease(sema_id);
```

Uso de Semáforos

3. Uso do semáforo a partir de uma ISR

```
osSemaphoreAcquire(sema_id, 0);
// seção crítica
osSemaphoreRelease(sema_id);
```

Não pode bloquear!

Exemplo: Produtor-Consumidor

- Comunicação entre tarefas

```
#define BUFFER_SIZE 10

osSemaphoreId_t vazio_id, cheio_id;

void main(void){
  ...
  vazio_id = osSemaphoreNew(BUFFER_SIZE, BUFFER_SIZE, NULL);
  cheio_id = osSemaphoreNew(BUFFER_SIZE, 0, NULL);
  ...
} // main
```

Exemplo: Produtor-Consumidor

Tarefa produtora de dados:

```
void produtor(void){
  while(1){
    osSemaphoreAcquire(vazio_id,
                       osWaitForever);
    // produz dados
    osSemaphoreRelease(cheio_id);
  } // while
} // produtor
```

Tarefa consumidora de dados:

```
void consumidor(void){
  while(1){
    osSemaphoreAcquire(cheio_id,
                       osWaitForever);
    // consome dados
    osSemaphoreRelease(vazio_id);
  } // while
} // consumidor
```

Exemplo

- O projeto “prodcons” da área de trabalho “EK-TM4C1294_RTOS_IAR8” implementa um sistema de duas tarefas que utilizam o modelo produtor-consumidor (dois semáforos contadores) para se comunicar por meio de um buffer circular global com 8 elementos de 8 bits cada (`uint8_t`).

Exemplo

- A tarefa produtora realiza uma contagem binária crescente de 0 a 15, enviando o valor corrente dessa contagem ao buffer circular.
- A tarefa consumidora recebe o valor da contagem do buffer circular e aciona os LEDs do kit de desenvolvimento com a sua representação binária (bit 3 no LED4, bit 2 no LED3, bit 1 no LED2 e bit 0 no LED1).

Exercício 5

- As duas tarefas (produtora e consumidora) no projeto “prodcons” são periódicas com períodos de ativação Tp e Tc, respectivamente.

1. Verifique qual é a frequência de acionamento dos LEDs do kit de desenvolvimento quando:
   - Tp = 500ms, Tc = 1s
   - Tp = 2s, Tc = 250ms
   - Tp = 500ms, Tc = 500ms

Exercício 5

2. Qual é o fator que determina a frequência de acionamento dos LEDs nesse sistema?
3. Qual é o impacto do tamanho do buffer quando ambas as tarefas são periódicas?
4. Desenhe um diagrama de objetos para a arquitetura desse sistema.
5. Desenhe um diagrama de atividades ou um diagrama de máquina de estados para cada tarefa desse sistema.

Exercício 6

- Com base no projeto “prodcons” da área de trabalho “EK-TM4C1294_RTOS_IAR8”, implemente um programa concorrente em que a entidade produtora de dados seja uma ISR e a entidade consumidora de dados seja uma tarefa.

Exercício 6

- Comportamento do software:
  - O pressionamento do botão SW1 do kit deverá causar uma requisição de interrupção.
  - A cada atendimento de interrupção pela ISR correspondente, uma variável global deverá ter o seu valor incrementado e colocado no buffer de comunicação com a tarefa.
  - A tarefa deverá retirar o valor do buffer de comunicação e apresentar o valor binário dos seus 4 bits menos significativos nos 4 LEDs do kit.

Exercício 6

- Efeito desejado: a cada pressionamento do botão SW1 deverá haver um incremento na contagem de 4 bits apresentada nos LEDs D1 a D4 do kit.
- Utilize as funcionalidades da biblioteca driverlib (TivaWare) para configurar a interrupção de GPIO no port em que os botões do kit estão conectados.