Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Eletrônica (DAELN)

SISTEMAS EMBARCADOS

Acesso a recursos compartilhados

Prof. André Schneider de Oliveira
andreoliveira@utfpr.edu.br

Compartilhamento de recursos

- A parte do código de uma thread que realiza o acesso ao recurso compartilhado é denominada de **seção crítica**
- A **seção crítica** é uma seção do código que não pode ser interrompida por outro processo ***(preempção***)
- Por exemplo:
  - **escrita em uma memória compartilhada**
  - **acesso a um dispositivo de I/O**

Problemas com recursos compartilhados

- **Ausência de impasse (*deadlock*)** = Se dois ou mais processos tentarem entrar na sua seção crítica, ao menos entrará
- **Ausência de atrasos desnecessários** = se não houver outros processos na seção crítica, um processo não deve sofrer atrasos para entrar
- **Garantia de entrada** = todas as thread devem ter a oportunidade de acessar a sua seção crítica
- **Exclusão mútua** = Apenas um processo na seção crítica em determinado instante de tempo

Exemplo de deadlock

- Duas tarefas (task1 e task2) necessitam de dois recursos, acessados por S1 e S2

Task1	Task2
wait(&S1)	wait(&S2)
//possui R1	//possui R2
wait (&S2)	wait(&S1)
//esperando R2	//esperando R1

DEADLOCK!!

Seção crítica e exclusão mútua

- **Seção crítica:** parte do programa onde são efetuados acessos (para leitura e escrita) a recursos compartilhados por dois ou mais processos

- **Exclusão mútua:** método de sincronização que garante o acesso exclusivo a um recurso compartilhado
  - Um processo não terá acesso à região critica quando outro processo está utilizando essa região

Como Garantir Exclusão Mútua?

- **Suporte em hardware**
  - **Instrução Test-And-Set (TST)**
    - É uma instrução denominada de “atômica” ou interrompível
    - Consiste em uma sequencia de checar e atualizar um valor sem interrupções
  - **Desabilitar as interrupções de hardware antes de entrar nas seções críticas**
    - **Problemas:**
      - O processo pode não reabilitar as interrupções
      - As interrupções também desabilitam eventos de processos não conflitantes

```
Desabilita Interrupções
Região Crítica
Habilita Interrupções
```

Como Garantir Exclusão Mútua?

- **Soluções do software**
  - **Algoritmo de peterson**

```
int try0 = 0, try1 = 0;
int turn = 0; // Or 1

// Fork processes sharing variables try0, try1, turn
// Process 0                          // Process 1
try0 = 1;                             try1 = 1;
turn = 0;                             turn = 1;
while (try1 && !turn) { }             while (try0 && turn) { }
// Critical section                   // Critical section
try0 = 0;                             try1 = 0;
// Non-critical section               // Non-critical section
```

Como Garantir Exclusão Mútua?

- **Mutex - acesso exclusivo**
  - "bloqueia" um processo enquanto usa
  - "desbloqueia" quando libera
  - Kernel suspende as threads que necessitam de um recurso que está "bloqueado"

- **Semáforos - sequencializar o acesso**
  - gerenciamento por tokens
  - a thread faz a solicitação do token
    - é colocada em estado de waiting se o token não estiver disponível

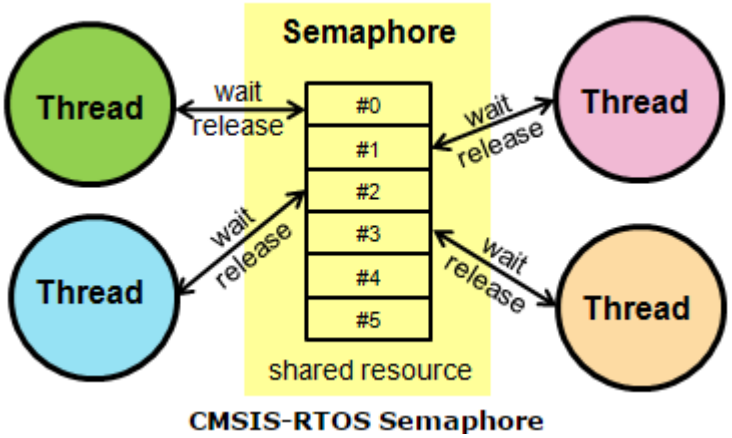
- **Mensagens**
  - buffer
  - subsistema de comunicação (send, receive) assíncrono

Semáforos no RTOS

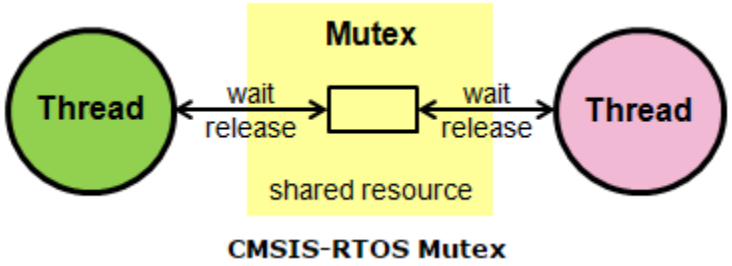
- **Binário (sincronização)**
  - Um único recurso compartilhado do mesmo tipo (teto de contagem = 1)
- **Contador (sincronização)**
  - Múltiplos recursos compartilhados do mesmo tipo (teto de contagem > 1)
- **Mutex (exclusão mútua)**
  - Binário, mas somente a tarefa dona (*owner*) do semáforo pode desligá-lo

Semáforos

- O CMSIS-RTOS prevê dois tipos de semáforo:
  - **osSemaphore** para sincronização entre tarefas ou entre tarefa e ISR
  - **osMutex** para exclusão mútua – apenas a tarefa que solicitou **osMutexWait** pode solicitar **`osMutexRelease`**



CMSIS-RTOS Semaphore



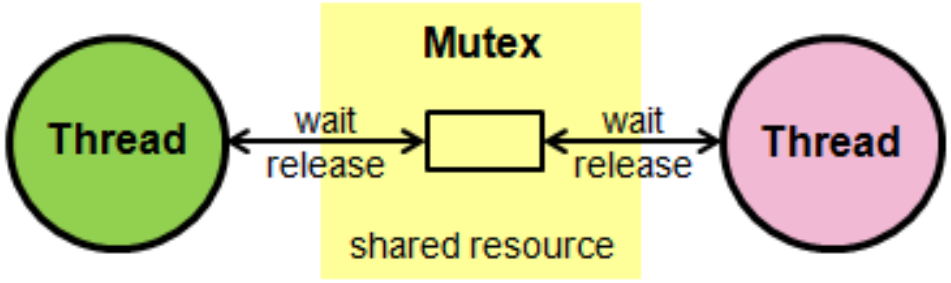
CMSIS-RTOS Mutex

Mutex

- A exclusão mútua (amplamente conhecida como Mutex) é usada em vários sistemas operacionais para gerenciamento de recursos compartilhados

- Diversos recursos em um sistema embarcados podem ser usados repetidamente, mas apenas **uma thread** pode acessá-lo a cada vez

- Mutexes são usados para proteger o acesso a um recurso compartilhado permitindo um acesso único



Mutex

- Um mutex é uma versão especial de um semáforo, ou seja, também opera com tokens
- Entretanto, possui apenas um token que representa o recurso compartilhado (token binário)
- A vantagem de um mutex é que ele introduz a “propriedade” da thread
  - Quando uma thread adquire o token mutex ele torna-se seu proprietário, ou seja, apenas o proprietário pode liberar o mutex
  - A próxima thread se torna o novo “proprietário” sem latência
  - As operações de mutex podem ser aninhadas

Gerenciamento de Mutexes

- **osMutexCreate** - Define e inicializa um mutex
- **osMutexWait** - Obtém um mutex ou aguarda até que este esteja disponível
- **osMutexRelease** - Libera um mutex
- **osMutexDelete** - Deleta um mutex

Gerenciamento de Mutexes

osMutexId **osMutexCreate** (const osMutexDef_t * mutex_def)

- cria o elemento de Mutex

Retorno

- Retorna o ID do Mutex ou NULL no caso de erro

```
#include "cmsis_os.h"

osMutexDef (MutexIsr);                                  // Mutex name definition

void CreateMutex (void)  {
osMutexId mutex_id;

  mutex_id = osMutexCreate  (osMutex (MutexIsr));
  if (mutex_id != NULL)  {
    // Mutex object created
  }
}
```

Gerenciamento de Mutexes

osStatus **osMutexWait** (osMutexId mutex_id, uint32_t millisec)

- Espera até que o Mutex esteja disponível ou que ocorra um "overflow"

Parâmetros

- **mutex_id = identificador do Mutex**
- **millisec = tempo máximo de espera em millisegundos**

Status and Error Codes

- osOK = o Mutex foi obtido com sucesso
- osErrorTimeoutResource = o Mutex não foi obtido dentro do tempo máximo
- osErrorParameter = o parâmetro mutex_id está incorreto
- osErrorResource = o Mutex não pode ser obtido dentro do tempo especificado
- osErrorISR = não pode ser chamada de uma ISR

```
#include "cmsis_os.h"

osMutexDef (MutexIsr);

void WaitMutex (void)  {
osMutexId mutex_id;
osStatus status;

  mutex_id = osMutexCreate  (osMutex (MutexIsr));
  if (mutex_id != NULL)  {
    status  = osMutexWait    (mutex_id, 0);
    if (status != osOK)  {
      // handle failure code
    }
  }
}
```

Gerenciamento de Mutexes

osStatus **osMutexRelease** (osMutexId mutex_id)

- Libera o Mutex e as demais threads que estão aguardando o Mutex passam para estado de ready
- Retorna o status da solicitação

Status and Error Codes

- osOK =Mutex foi liberado com sucesso
- osErrorParameter = o parâmetro mutex_id está incorreto
- osErrorResource = o Mutex não foi obtido anteriormente
- osErrorISR = não pode ser chamada de uma ISR

```
#include "cmsis_os.h"

osMutexDef (MutexIsr);                          // Mutex name definition
osMutexId mutex_id;                             // Mutex id populated by the function CreateMutex()
osMutexId CreateMutex (void);                   // function prototype that creates the Mutex

void ReleaseMutex (osMutexId mutex_id)  {
osStatus status;

  if (mutex_id != NULL)  {
    status = osMutexRelease(mutex_id);
    if (status != osOK)  {
      // handle failure code
    }
  }
}
```

Gerenciamento de Mutexes

osStatus **osMutexDelete** (osMutexId mutex_id)

- Deleta o objeto Mutex
- Retorna o status da solicitação

Status and Error Codes

- osOK =Mutex foi deletado com sucesso
- osErrorParameter = o parâmetro mutex_id está incorreto
- osErrorResource = todos os tokens foram liberados
- osErrorISR = não pode ser chamada de uma ISR

```
#include "cmsis_os.h"

osMutexDef (MutexIsr);                                  // Mutex name definition
osMutexId mutex_id;                                     // Mutex id populated by the function CreateMutex()
osMutexId CreateMutex (void);                           // function prototype that creates the Mutex

void DeleteMutex (osMutexId mutex_id)  {
osStatus status;

  if (mutex_id != NULL)  {
    status = osMutexDelete(mutex_id);
    if (status != osOK)  {
      // handle failure code
    }
  }
}
```

Uso dos Mutexes

1. Definição da estrutura de dados do Mutex (costuma ser uma **variável global**)

```
osMutexDef(mut);
```

2. Definição de um ponteiro para o Mutex (também costuma ser uma **variável global**)

```
osMutexId mut_id;
```

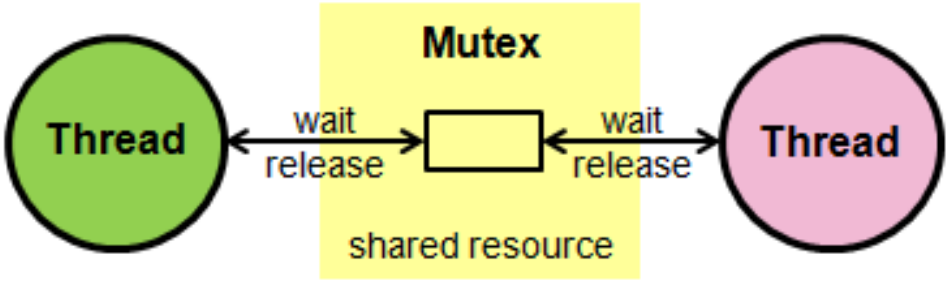
3. Inicialização do Mutex a partir da função main

```
mut_id = osMutexCreate(osMutex(mut));
```

4. Uso do Mutex sem timeout a partir de uma tarefa

```
osMutexWait(mut, osWaitForever);
//seção crítica
osMutexRelease(mut);
```

Trabalhando com Mutexes



1. Declare the mutex container and initialize the mutex:

```
osMutexDef (uart_mutex);     // Declare mutex
osMutexId  (uart_mutex_id); // Mutex ID
```

2. Create the mutex in a thread:

```
uart_mutex_id = osMutexCreate(osMutex(uart_mutex));
```

3. Acquire the mutex when peripheral access is required:

```
osMutexWait(uart_mutex_id, osWaitForever);
```

4. When finished with the peripheral access, release the mutex:

```
osMutexRelease(uart_mutex_id);
```

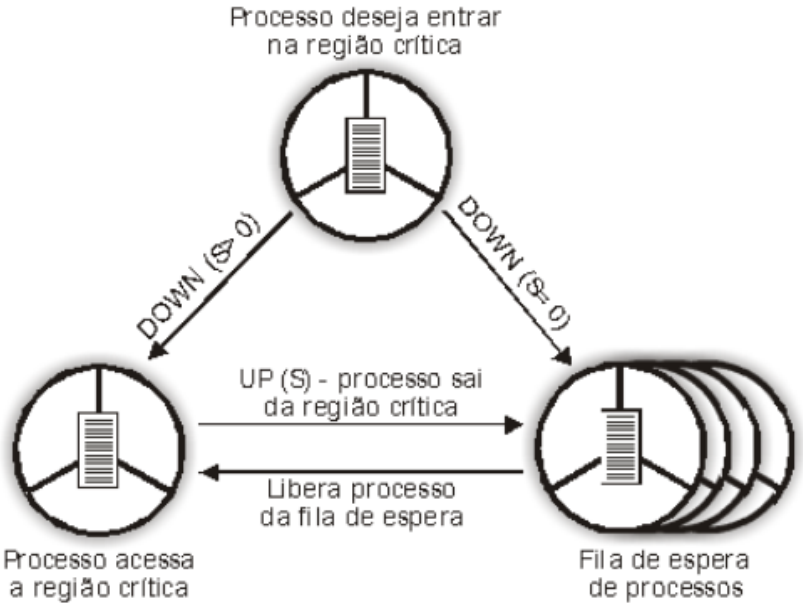
Exemplo do uso de Mutexes

```
osMutexId stdio_mutex;
osMutexDef(stdio_mutex);

void notify(const char* name, int state) {
    osMutexWait(stdio_mutex, osWaitForever);
    printf("%s: %d\n\r", name, state);
    osMutexRelease(stdio_mutex);
}

void test_thread(void const *args) {
    while (1) {
        notify((const char*)args, 0);
        osDelay(1000);
        notify((const char*)args, 1);
        osDelay(1000);
    }
}

void t2(void const *argument) {
      test_thread("Thread 2");
}
osThreadDef(t2, osPriorityNormal, 1, 0);
```

```
void t3(void const *argument) {
      test_thread("Thread 3");
}
osThreadDef(t3, osPriorityNormal, 1, 0);

int main() {
    stdio_mutex = osMutexCreate(osMutex(stdio_mutex));

    osThreadCreate(osThread(t2), NULL);
    osThreadCreate(osThread(t3), NULL);

    test_thread((void *)"Thread 1");

    osDelay(osWaitForever);
}
```

Semáforos

- O Semáforo realiza o controle de acesso a recursos compartilhados pela passagem de permissão de acesso (**tokens**)
- O gerenciamento é feito por uma variável inteira não negativa que pode ser manipulada por duas instruções *Down* e *Up*
- As modificações feitas no valor do semáforo usando *Down* e *Up* são atômicas



Gerenciamento de Semáforos

- **osSemaphoreCreate** - define e inicializa um semáforo
- **osSemaphoreWait** - Obtém um semáforo ou aguarda até que este esteja disponível
- **osSemaphoreRelease** - Libera um token de um semáforo
- **osSemaphoreDelete** - Deleta um semáforo

Gerenciamento de Semáforos

osSemaphoreId **osSemaphoreCreate** (const osSemaphoreDef_t *
semaphore_def, int32_t *count*)

- Cria o objeto de Semafóro
- Retorna o status da solicitação

Parameters

- semaphore_def = definição do objeto semáforo
- count = número de recursos disponíveis

Returns

retorna o identificador do Semáforo ou NULL no caso de erro

Gerenciamento de Semáforos

```
#include "cmsis_os.h"

osThreadId tid_thread1;                         // ID for thread 1
osThreadId tid_thread2;                         // ID for thread 2

osSemaphoreId semaphore;                        // Semaphore ID
osSemaphoreDef(semaphore);                      // Semaphore definition

//
//   Thread 1 - High Priority - Active every 3ms
//
void thread1 (void const *argument) {
  int32_t value;
  while (1) {
    osDelay(3);                                 // Pass control to other tasks for 3ms
    val = osSemaphoreWait (semaphore, 1);       // Wait 1ms for the free semaphore
    if (val > 0) {
                                                // If there was no time-out the semaphore was acquired
      :                                         // OK, the interface is free now, use it.
      osSemaphoreRelease (semaphore);           // Return a token back to a semaphore
    }
  }
}

//
//   Thread 2 - Normal Priority - looks for a free semaphore and uses
//                                the resource whenever it is available
//
void thread2 (void const *argument) {
  while (1) {
    osSemaphoreWait (semaphore, osWaitForever); // Wait indefinitely for a free semaphore
                                                // OK, the interface is free now, use it.
    :
    osSemaphoreRelease (semaphore);             // Return a token back to a semaphore.
  }
}

// Thread definitions
osThreadDef(thread1, osPriorityHigh,   1, 0);
osThreadDef(thread2, osPriorityNormal, 1, 0);

void StartApplication (void) {
  semaphore = osSemaphoreCreate(osSemaphore(semaphore), 1);
  tid_thread1 = osThreadCreate(osThread(thread1), NULL);
  tid_thread2 = osThreadCreate(osThread(thread2), NULL);
  :
}
```

Gerenciamento de Semáforos

osSemaphoreId **osSemaphoreWait** (osSemaphoreId semaphore_id
uint32_t millisec)

- Espera até que um token do Semáforo fique disponível ou que ocorra o estouro do temporizador
- Retorna o número de tokens disponíveis ou -1 no caso de parâmetros incorretos

Parameters

- semaphore_id = identificador do objeto semáforo
- millisec = tempo máximo em millisegundos ou 0 para desabilitar o time-out

Gerenciamento de Semáforos

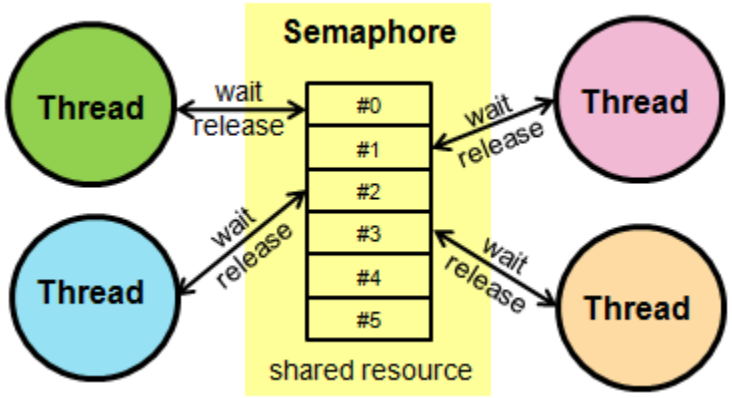
osStatus **osSemaphoreDelete** (osSemaphoreId semaphore_id)

- Deleta o objeto Semáforo
- Retorna o status da solicitação

Status and Error Codes

- osOK = Semáforo foi deletado com sucesso
- osErrorParameter = o parâmetro semaphore_id está incorreto
- osErrorResource = o objeto não pode ser deletado
- osErrorISR = não pode ser chamada de uma ISR

Trabalhando com Semáforos




```
osSemaphoreDef(multiplex);
osSemaphoreId (multiplex_id);

void thread_n (void)
  {
    multiplex_id = osSemaphoreCreate(osSemaphore(multiplex), 3);
    while(1)
      {
        osSemaphoreWait(multiplex_id, osWaitForever);
        // do something
        osSemaphoreRelease(multiplex_id);
      }
  }
```

Exemplo de uso dos Semáforos

```
osSemaphoreId two_slots;
osSemaphoreDef(two_slots);

void test_thread(void const *name) {
    while (1) {
        osSemaphoreWait(two_slots, osWaitForever);
        printf("%s\n\r", (const char*)name);
        osDelay(1000);
        osSemaphoreRelease(two_slots);
    }
}

void t2(void const *argument) {
      test_thread("Thread 2");
}
osThreadDef(t2, osPriorityNormal, 1, 0);

void t3(void const *argument) {
      test_thread("Thread 3");
}
osThreadDef(t3, osPriorityNormal, 1, 0);
```

```
int main () {

    osKernelInitialize();


    two_slots = osSemaphoreCreate(osSemaphore(two_slots), 2);

    osThreadCreate(osThread(t2), NULL);
    osThreadCreate(osThread(t3), NULL);

    osKernelStart();

    test_thread((void *)"Thread 1");

    osDelay(osWaitForever);

}
```