

CMSIS RTOS

César Yutaka Ofuchi

ofuchi@utfpr.edu.br

(adaptado do prof. André Schneider de Oliveira)



Kernel

- Kernel (ou núcleo) é uma abstração do hardware para a programação em alto-nível das interfaces
- Promove a conexão entre hardware e software por
 - processos
 - comunicação entre processos
 - memória virtual
 - sistema de arquivos
- Contém um conjunto de "**device drivers**" para gerenciar a interação com os subsistemas de hardware

Ciclo de Vida dos Elementos do Kernel

Os elementos do Kernel ([threads](#), temporizadores, semáforos, etc.) seguem um ciclo de vida:

1. Definição

```
osThreadDef(job1, osPriorityAboveNormal, 1, 0);
```

2. Inicialização

```
job1_id = osThreadCreate(osThread(job1), NULL);
```

3. Operação

```
osStatus status = osThreadYield();  
osThreadId id = osThreadGetId();  
osPriority priority = osThreadGetPriority(id);  
osThreadSetPriority(id, osPriorityNormal);
```

4. Término

```
osThreadTerminate(job1_id);
```

Macros de Definição e Acesso

Definição

- osThreadDef
- osTimerDef
- osMutexDef
- osSemaphoreDef
- osPoolDef
- osMessageDef
- osMailDef

Acesso

- osThread
- osTimer
- osMutex
- osSemaphore
- osPool
- osMessage
- osMail

Macros de Definição e Acesso (Exemplo de Threads)

Arquivo cmsis_os.h → Macro de Definição da estrutura

```
#define osThreadDef(name, priority, instances, stacksz) \
const osThreadDef_t os_thread_def_##name = \
{ (name), (priority), (instances), (stacksz) }
```

Exemplo: osThreadDef(**job**,0,1,1)

- Macro que cria uma instância de uma estrutura do tipo **osThreadDef_t**, que contém:

- nome da tarefa: **job**
- Prioridade: 0
- número máximo de instâncias: 1
- tamanho da pilha em bytes: 1

```
typedef struct os_thread_def {
    os_pthread          pthread;
    osPriority           tpriority;
    uint32_t            instances;
    uint32_t            stacksize;
} osThreadDef_t;
```

- O nome desta instância é: os_thread_def_**job**

Macros de Definição e Acesso (Exemplo de Threads)

Arquivo cmsis_os.h → Macro de Acesso da estrutura

```
#define osThread(name)    &os_thread_def_##name)
```

- Macro que é expandida para: `&os_thread_def_job`, ou seja, um ponteiro para a estrutura criada com `osThreadDef`

Kernel

- `osKernelInitialize`
 - Inicializa o kernel
- `osKernelStart`
 - Ativa o kernel
- `osKernelRunning`
 - Consulta se o kernel está ativo
- `osKernelSysTick`
 - Obtém o valor do temporizador do kernel (usado no gantt por exemplo)
- `osDelay` (função de espera genérica)
 - Espera por um tempo específico

Kernel Estrutura Básica

```
#include "cmsis_os.h"

// definições de tarefas, temporizadores, etc.
// declarações das funções das tarefas e de callback

void main() {
    osKernelInitialize();

    // inicializações de hardware
    // ativação de tarefas, temporizadores, etc.

    osKernelStart();

    // laço de repetição ou encerramento da tarefa main()
    // exemplo: while(1){}; ou
    // osDelay(osWaitForever)
} // main
```


Controle de Kernel no RTOS

Macros (ou definições possíveis) - "cmsis_os.h"

#define **osFeature_MainThread** 1

- define se a função **Main** será uma thread **1=habilitada, 0=desabilitada**

#define **osFeature_SysTick** 1

- habilita as funções do osKernelSysTick **1=habilitada, 0=desabilitada**

#define **osCMSIS** 0x10002 - versão da API CMSIS

#define **osCMSIS_KERNEL** 0x10000 - Identificação e versão do RTOS

#define **osKernelSystemId** "KERNEL V1.00" - String de identificação do RTOS

#define **osKernelSysTickFrequency** 100000000 - Frequencia (Hz) do SysTick

#define **osKernelSysTickMicroSec(microsec)**

$(((\text{uint64_t})\text{microsec} * (\text{osKernelSysTickFrequency})) / 1000000)$

- Converte um tempo em microsegundos para a frequência do SysTick
- Comumente utilizado para pequenos atrasos em tarefas de "**pooling**"

Configurações do Kernel

Arquivo **RTX_Conf_CM.c**



Muito importante

Configurações das threads

- **OS_TASKCNT** = # de threads executando concorrentemente <padrão 6>
- **OS_STKSIZE** = define o tamanho da pilha threads com stackz=0 <padrão 200>
- **OS_MAINSTKSIZE** = define o tamanho da pilha para a Main thread <padrão 200>
- **OS_PRIVCNT** = # threads com pilhas especificadas pelo usuário <padrão 0>
- **OS_STKCHECK** = habilita ou desabilita o teste de "**overflow**" de pilha na preempção (essa opção atrasa a troca de threads)
- **OS_RUNPRIV** = define o modo de execução das threads (0=Unpriv, 1=Priv) <padrão 1>

Configurações do Kernel

Arquivo **RTX_Conf_CM.c**

Configurações do SysTick

- **OS_SYSTICK** = 1 para utilizar o SysTick timer como RTOS Kernel Timer
- **OS_CLOCK** = especifica a frequencia do RTOS Kernel timer [Hz], geralmente é idêntico ao **core clock**
- **OS_TICK** = intervalo do SysTick [μ s] <padrão 1000 = 1 μ s>

Função de idle

void **os_idle_demon** (void) {...}

função que é executada quando nenhuma thread está em estado **ready**

Gerenciamento de Threads

- **osThreadCreate** – Ativa a execução de uma tarefa

Thread Ativa

- **osThreadTerminate** – Desativa a execução de uma tarefa
- **osThreadYield** – Passa a execução à próxima tarefa que pronta
- **osThreadId** – Obtém o identificador que referencia a tarefa
- **osThreadSetPriority** – Altera a prioridade de uma tarefa
- **osThreadGetPriority** – Obtém a prioridade atual de uma tarefa

Funções de espera

1. **osDelay:** Suspende a execução de uma thread por um intervalo de tempo
2. **osWait:** Aguarda um evento não especificado por um período de tempo

Macros (ou definições possíveis) - "cmsis_os.h"

- #define **osFeature_Wait** 1
 - 1=habilitado, 0=não habilitado

osDelay

osStatus **osDelay** (uint32_t millisec)

- Retorna o status da solicitação

Status and Error Codes

- **osEventTimeout** = delay executado
- **osErrorISR** = não pode ser chamada de uma ISR

```
#include "cmsis_os.h"

void Thread_1 (void const *arg) {
    osStatus status;
    uint32_t delayTime;

    delayTime = 1000;
    :
    status = osDelay (delayTime);
    // handle error code
    :
}
```

// Thread function
// capture the return status
// delay time in milliseconds

// delay 1 second

// suspend thread execution

osWait

osEvent **osWait** (uint32_t millisec)

- Aguarda um evento não especificado
- millisec: timeout ou 0 para sem timeout
- Retorna o evento com um componente RTOS ou erro

Eventos

- Um **signal** enviado para uma thread explicitamente
- Um **mail** ou **message** registrado para uma thread

Status and Error Codes

- **osEventSignal** = ocorreu um evento de **signal**
- **osEventMessage** = ocorreu um evento de **message**
- **osEventMail** = ocorreu um evento de **mail**
- **osEventTimeout** = não ocorreu um evento dentro do tempo especificado (timeout)
- **osErrorISR** = não pode ser chamada de uma ISR

Exemplo osWait

osEvent **osWait** (uint32_t millisec)

```
#include "cmsis_os.h"

void Thread_1 (void const *arg) {
    osEvent Event;
    uint32_t waitTime;

    :
    waitTime = osWaitForever;
    Event = osWait (waitTime);
    switch (Event.status) {
        case osEventSignal:
            :
            break;

        case osEventMessage:
            :
            break;

        case osEventMail:
            :
            break;

        case osEventTimeout:
            break;

        default:
            break;
    }
    :
}
```

// Thread function
// capture the event
// wait time in milliseconds

// special "wait" value
// wait forever and until an event occurred

// Signal arrived
// Event.value.signals contains the signal flags

// Message arrived
// Event.value.p contains the message pointer
// Event.def.message_id contains the message Id

// Mail arrived
// Event.value.p contains the mail pointer
// Event.def.mail_id contains the mail Id

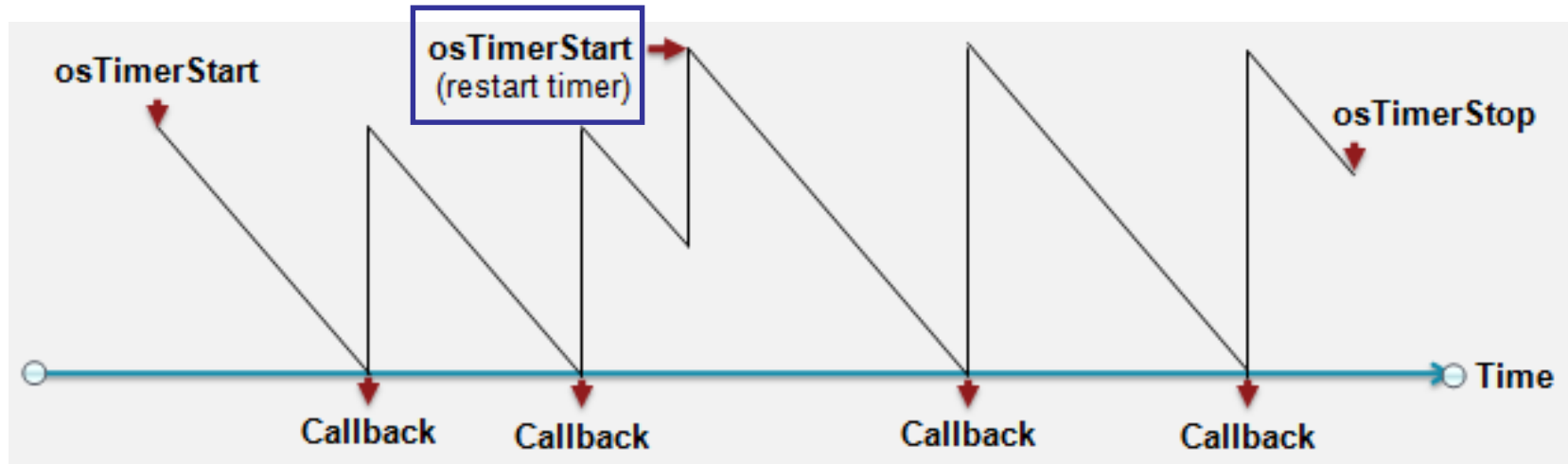
// Timeout occurred

// Error occurred

Temporizadores

- O CMSIS RTOS permite a criação de temporizadores (**timers**) para a geração de eventos periódicos ou atrasos
- É gerada uma interrupção (**callback**) de estouro do temporizador
- Existem dois tipos de temporizadores
 - `osTimerOnce` = temporizador sem auto-reload (**one-shot**)
 - `osTimerPeriodic` = temporizador com auto-reload

Temporizador Periódico

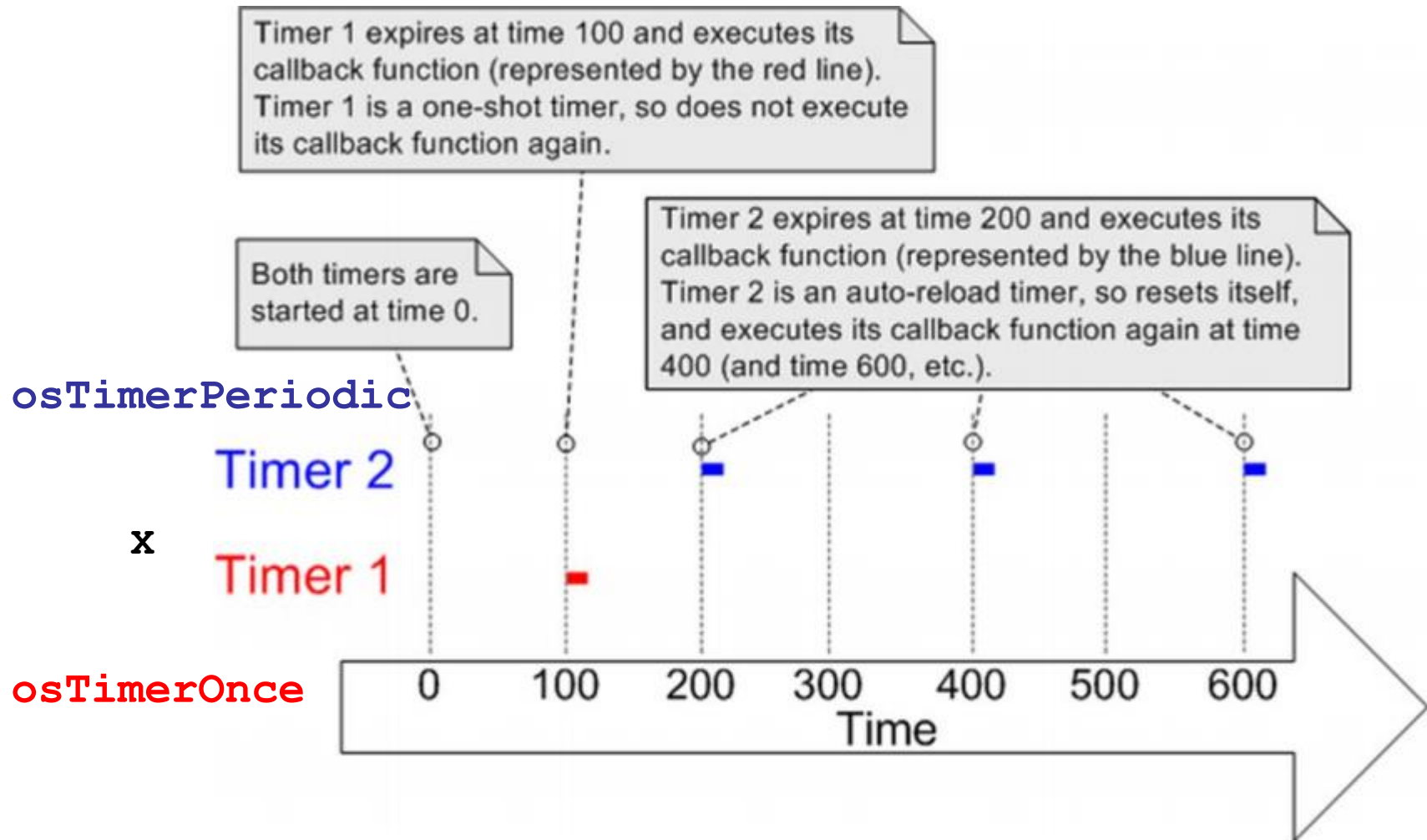


Behavior of a Periodic Timer

- **osTimerStart** define o valor inicial e inicia contagem regressiva
- **osTimerStop** para a contagem regressiva
- Quando a contagem atinge zero, é chamada a função de **callback**

Obs: para redefinir o valor inicial e reiniciar a contagem, é necessário executar **osTimerStop** antes de executar **osTimerStart**!

One-shot & Auto-reload



Gerenciamento de Temporizadores

- **osTimerCreate**
 - Define os atributos da função de callback de um temporizador
- **osTimerStart**
 - Ativa um temporizador com um valor de tempo
- **osTimerStop**
 - Desativa um temporizador

Exemplo do uso de timer LPC1343

```
void blink(void const *n) {  
    pca_toggle((int)n);  
}  
  
osTimerDef(blink_0, blink);  
osTimerDef(blink_1, blink);  
osTimerDef(blink_2, blink);  
osTimerDef(blink_3, blink);  
  
int main(void) {  
  
    osKernelInitialize();  
  
    SystemInit();  
    I2CInit( (uint32_t)I2CMASTER, 0 );  
  
    osTimerId timer_0 = osTimerCreate(osTimer(blink_0), osTimerPeriodic, (void *)0);  
    osTimerId timer_1 = osTimerCreate(osTimer(blink_1), osTimerPeriodic, (void *)1);  
    osTimerId timer_2 = osTimerCreate(osTimer(blink_2), osTimerPeriodic, (void *)2);  
    osTimerId timer_3 = osTimerCreate(osTimer(blink_3), osTimerPeriodic, (void *)3);  
  
    osTimerStart(timer_0, 2000);  
    osTimerStart(timer_1, 1000);  
    osTimerStart(timer_2, 500);  
    osTimerStart(timer_3, 250);  
  
    osKernelStart();  
    osDelay(osWaitForever);  
}
```

callback

Prática – exemplo timer

1. Abrir o exemplo RTOS_2_Timer do workspace e verifique o funcionamento
2. Troque o modo periódico para one-shot
3. Troque todos os leds que piscam
4. Após o osTimerStart, redefina outro período de tempo para os temporizadores

Consulte o site do CMSIS RTOS para mais detalhes da API

https://www.keil.com/pack/doc/CMSIS/RTOS/html/group__CMSIS__RTOS_TimerMgmt.html

Comunicação entre threads

- Comunicação entre processos - **Interprocess communication (IPC)** consiste de um conjunto de mecanismos para troca de informações entre os processos

Existem dois tipos de comunicação

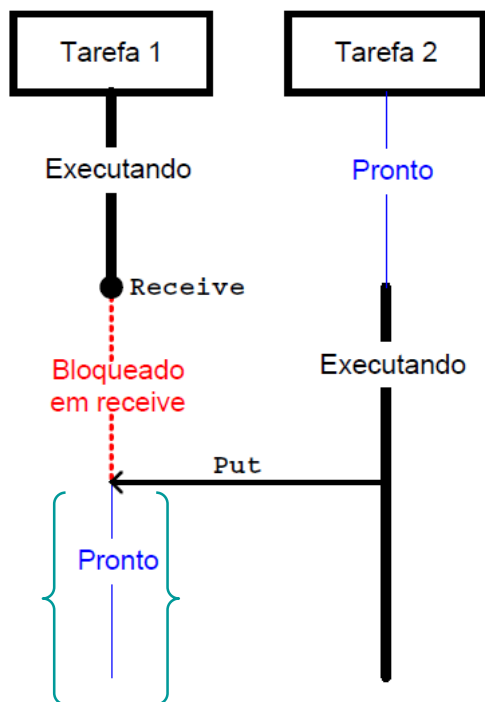
- **Bloqueante** = a informação é enviada e o processo fica aguardado resposta.
 - Ocorre a preempção para outro processo pronto (**ready**) e o processo atual vai para o estado de espera (**waiting**)
- **Não bloqueante** = a informação é enviada sem a necessidade de resposta
 - O processo continua em execução (**running**)

Comunicação entre threads

- Principais métodos para a comunicação entre processos
 - **eventos entre processos**
 - existem flags sinalizadoras de eventos entre os processos
 - um processo fica aguardando que seja realizado um evento de alteração de flag
 - **memória compartilhada**
 - os processos contém um espaço de memória em comum
 - devem haver políticas para evitar a perda/destruição das informações
 - **troca de mensagem**
 - os processos enviam mensagem em um canal de comunicação (real ou virtual)
 - não existe um endereçamento comum

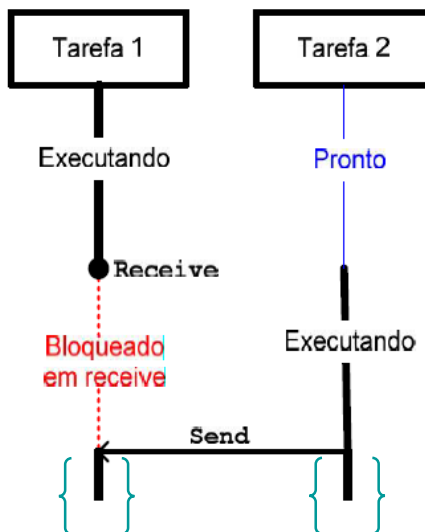
Comunicação entre threads

Assíncrono



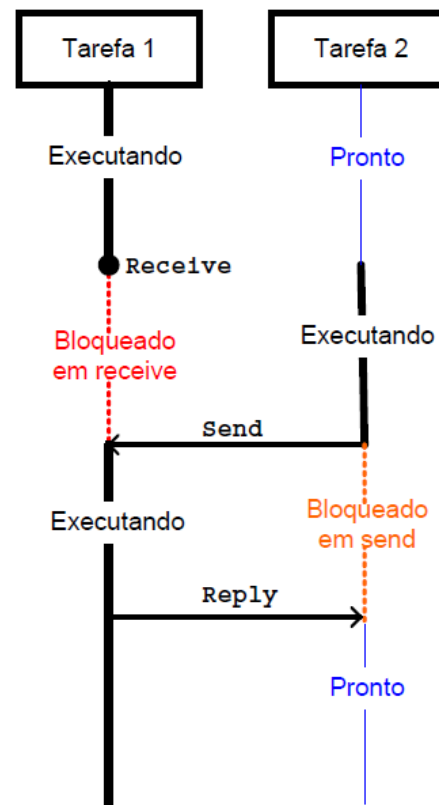
Ex: Fila de Mensagens de tamanho grande

Síncrono



Ex: Thread enviando evento

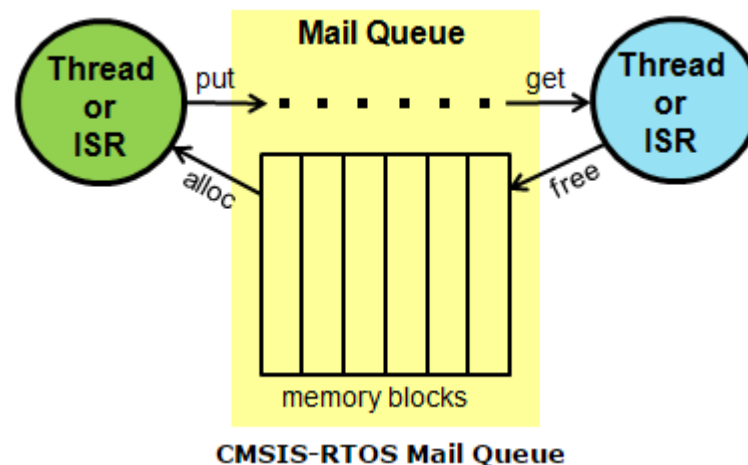
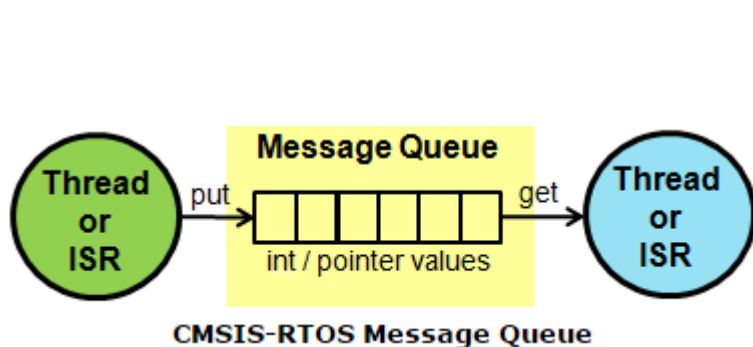
Síncrono com resposta



Ex: Thread aguardando recurso

Comunicação entre threads RTOS

- O CMSIS RTOS contém diversos mecanismos para a comunicação entre threads
- Esses componentes são geradores de eventos (**osWait**)
 - sinais (flags sinalizadores)
 - mensagens (inteiro de 32 bits ou ponteiros)
 - correspondências (blocos de memória)



Signal

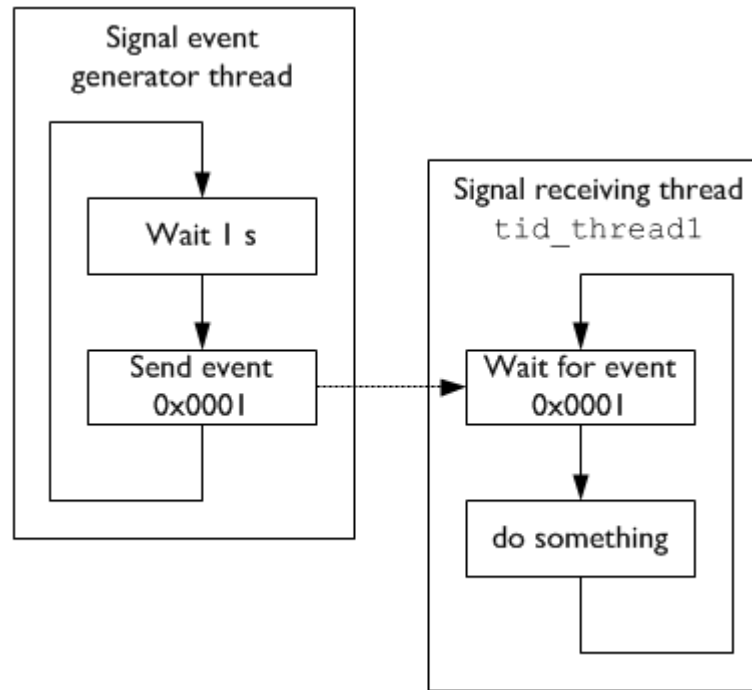
- O **signal** (sinal) é uma flag compartilhada entre duas threads que gera eventos como uma interrupção de software
- O evento de **signal** é análogo à uma interrupção e força a execução de uma determinada área de um processo
- Essa modalidade de comunicação não tem o intercâmbio de dados mas de eventos (do tipo do **signal**)
- Os **signals** são comumente utilizados para a sincronização de threads
- Cada thread CMSIS RTOS pode possuir até 32 **signal flags**

Macros (ou definições possíveis) - "cmsis_os.h"
`#define osFeature_Signals 8`
numero máximo de SF por thread

Signal

- `osSignalSet`
 - Ativa flags sinalizadores para uma tarefa
- `osSignalClear`
 - Desativa flags sinalizadores para uma tarefa
- `osSignalWait`
 - Suspende execução até que flags sinalizadores específicos sejam ativados

Comunicação por Eventos (Signal)



Simple signal event communication

1. In the thread (for example thread ID `tid_thread1`) that is supposed to wait for a signal, call the wait function:

```
osSignalWait (0x0001, osWaitForever); // wait forever for the signal 0x0001
```

2. In another thread (or threads) that are supposed to wake the waiting thread up call:

```
osSignalSet (tid_thread1, 0x0001); // set the signal 0x0001 for thread tid_thread1  
osDelay (1000); // wait for 1 second
```

Exemplo de comunicação por SF no LPC1343

```
void led_thread(void const *args) {
    while (1) {
        // Signal flags that are reported as event are automatically cleared.
        osSignalWait(0x1, osWaitForever);
        pca_toggle(0);
    }
}

osThreadDef(led_thread, osPriorityNormal, 1,0);

int main (void) {

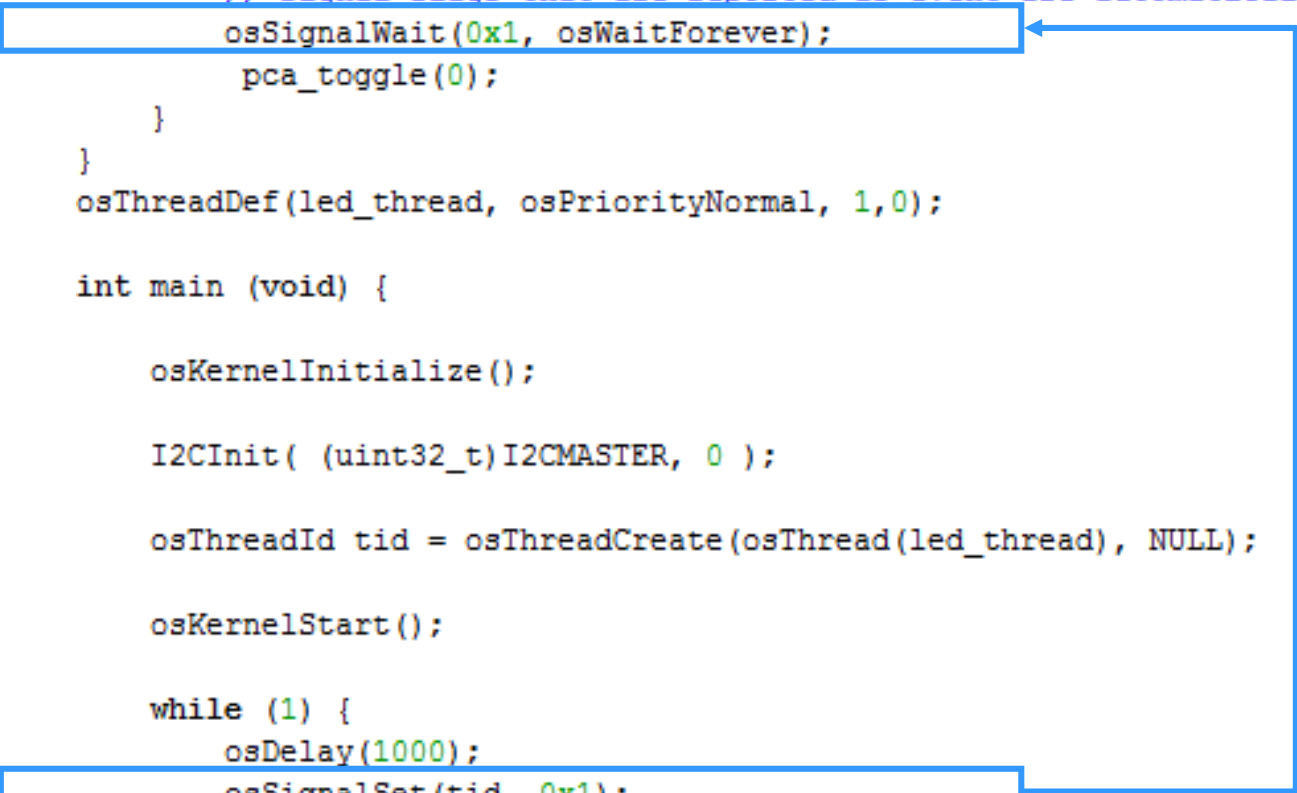
    osKernelInitialize();

    I2CInit( (uint32_t)I2CMaster, 0 );

    osThreadId tid = osThreadCreate(osThread(led_thread), NULL);

    osKernelStart();

    while (1) {
        osDelay(1000);
        osSignalSet(tid, 0x1);
    }
}
```



Chamada de ISR

- Apenas algumas funções específicas do CMSIS RTOS podem ser chamadas de Interrupções (ISR - Interrupt Service Routines)
 - `osKernelRunning`
 - `osSignalSet`
 - `osSemaphoreRelease`
 - `osPoolAlloc`, `osPoolCAlloc`, `osPoolFree`
 - `osMessagePut`, `osMessageGet`
 - `osMailAlloc`, `osMailCAlloc`, `osMailGet`, `osMailPut`, `osMailFree`
- As funções que não podem ser chamadas de ISR, irão verificar o status de interrupção e caso estejam em uma interrupção, vão gerar um código `osErrorISR`.

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

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 uma região crítica quando outro processo está utilizando essa região

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

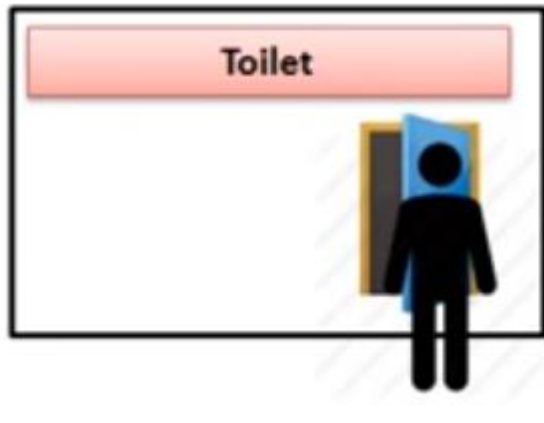
Mutex



Banheiro possui uma chave e apenas a pessoa com a chave pode entrar no banheiro



Mutex



Pessoa pega a chave, destranca a porta, entra no banheiro e depois fecha



Mutex



Mutex

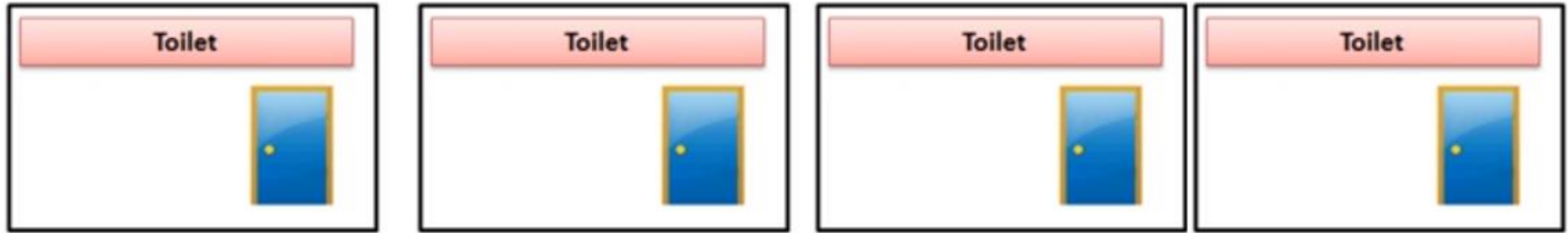


Pessoa passa a chave para próxima pessoa entrar no banheiro



- Pessoa = Thread
- Chave = Mutex
- Banheiro = Seção Crítica
- Pessoa tem a posse da chave (Thread possui o mutex)

Semáforos



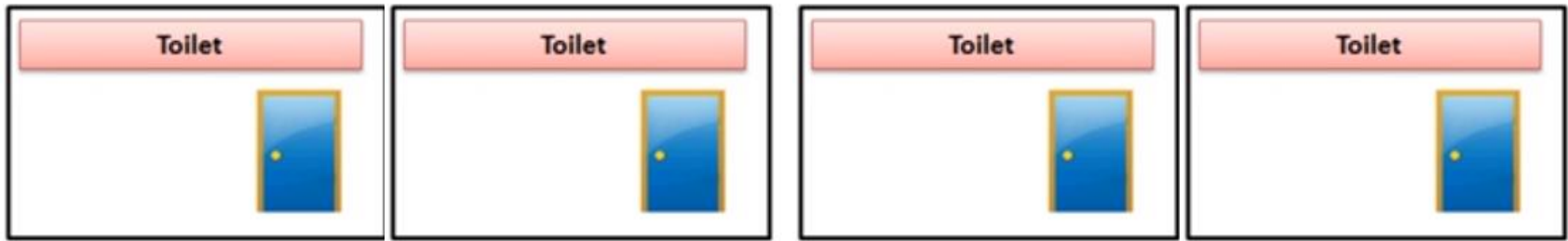
4 chaves idênticas

Chaves=4

5 pessoas



Semáforos



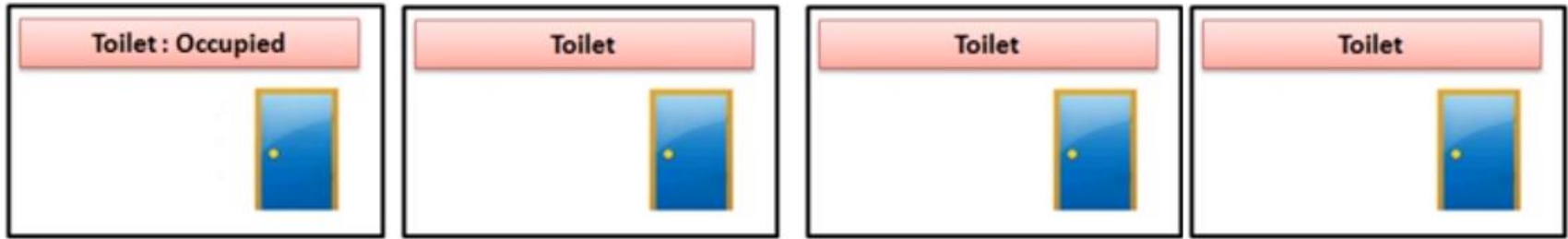
4 chaves idênticas

Chaves=3

4 pessoas



Semáforos



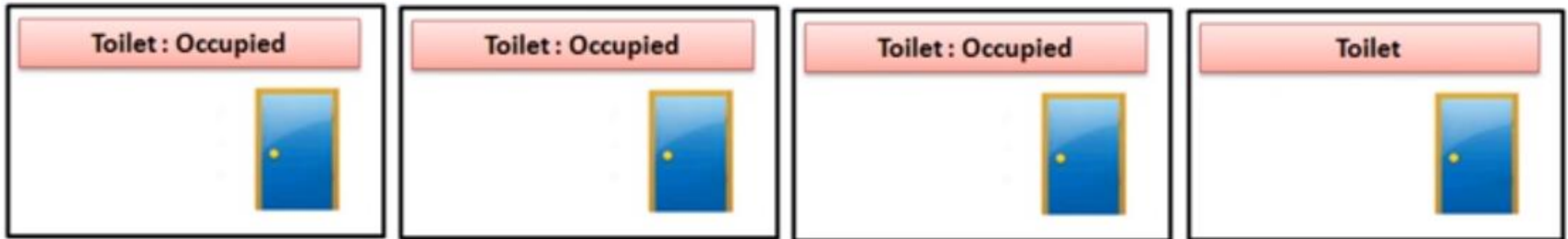
4 chaves idênticas

Chaves=1

2 pessoas



Semáforos



4 chaves idênticas



Chaves=0

1 pessoa



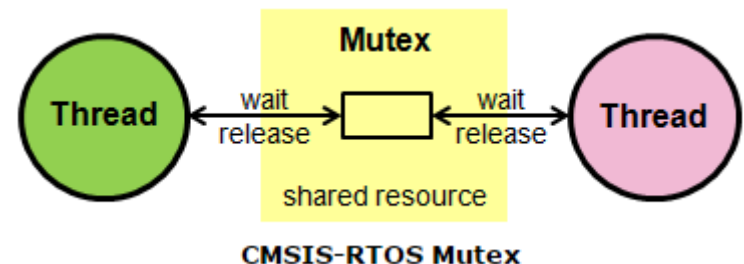
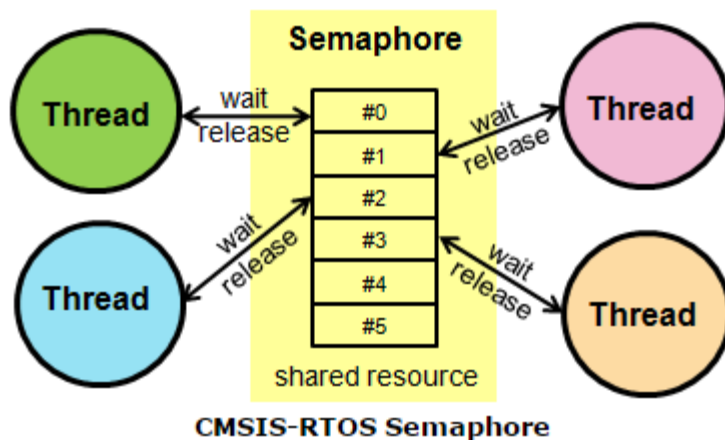
Deve Esperar!

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 no RTOS

- 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**



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

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);`

Obs: `osMutex()` é uma macro que retorna um ponteiro para a estrutura

Prática – exemplo blinky

1. Abrir o exemplo **Blinky** do workspace e verificar o funcionamento
2. Adicione as tags para o Gantt nas funções **phaseA—phaseD**, após o **signalWait**. Verifique a tela de Gantt.

```
time = osKernelSysTick()/ticks_factor;           //tempo1
.... Código a ser monitorado
fprintf(file," MSG : %i, %i\n", (int)time, //string
(int)osKernelSysTick()/ticks_factor);           //tempo 2
```

3. Na função **lcd** comente as funções de Mutex e delay. O que acontece no display OLED? Porque?
4. Na função **lcd** aumento o tamanho da variável char s[30] para s[300]. O que acontece? Se tiver um problema como resolve-lo?
5. Mude o número de threads no RTX_Conf_CM.c para 6. O que acontece? Porque?

Prática – exemplo blinky

6. No handler da interrupção onde estão os printf's, e adicione as tags para Gantt no Handler. Gere interrupções no botão. Plote o diagrama de Gantt. O que houve?
7. Adicione um `osDelay(500)` dentro do handler da interrupção O que acontece? Verifique o retorno da função `osDelay`.
8. Prática de signals:
 - 8.1 Troque a ordem das threads de `A->B->C->D->A...` para `B->A->C->B ...` (retirar thread D do ciclo).
 - 8.2 Mude a thread D para que ela não sinaliza mais para thread A. e faça a thread D ser acionada por interrupção (utilize signals para isso).