Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

Roteiro 07

Mutex e Semáforos

Leia com atenção - Informações iniciais:

1. No início de cada tópico/assunto é apresentado um **exercício de revisão** em que basta copiar o código na ferramenta, realizar a compilação e a execução e, então, interpretar o resultado. Este tipo de exercício tem como objetivo auxiliar o aluno a relembrar alguns conceitos e a validar as ferramentas que estão sendo utilizadas. Este código sempre estará correto e funcionando.
2. Os exercícios estão apresentados em **ordem crescente de dificuldade**.
3. Os exercícios abordam todos os conceitos relacionados ao conteúdo da aula em questão. Deste modo, caso o aluno não consiga resolver alguns dos exercícios, recomenda-se que o aluno participe dos **plantões de dúvidas/monitorias** e que busque aprender os conceitos envolvidos na atividade.
4. A **próxima atividade** de laboratório admitirá que os conceitos aqui apresentados já foram plenamente compreendidos.

Este tutorial foi traduzido e adaptado a partir do trabalho disponível em "The Designers Guide to the Cortex-M Processor Family" de Trevor Martin (Pág. 43 a 55).

Assim como os sinais, os semáforos são um método de sincronizar atividades entre duas ou mais threads. Simplificando, um semáforo é um contêiner que contém um número de tokens. À medida que uma thread executa, ela chegará a uma chamada RTOS para adquirir um token do semáforo. Se o semáforo contiver um ou mais tokens, a thread continuará executando e o número de tokens no semáforo será decrementado em um. Se atualmente não houver tokens no semáforo, a thread será colocada em um estado de espera até que um token esteja disponível. A qualquer momento durante sua execução, uma thread pode adicionar um token ao semáforo, fazendo com que seu número de tokens incremente em um.

A Fig. 01 ilustra o uso de um semáforo para sincronizar duas threads. Primeiro, o semáforo deve ser criado e inicializado com um número inicial de tokens. Neste caso, o semáforo é inicializado com um único token. Ambas as threads serão executadas e chegarão a um ponto em seu código onde tentarão adquirir um token do semáforo. A primeira thread a chegar a este ponto adquirirá o token do semáforo e continuará a execução. A segunda thread também tentará adquirir um token,

mas como o semáforo está vazio, ela interromperá a execução e será colocada em um estado de espera até que um token de semáforo esteja disponível.

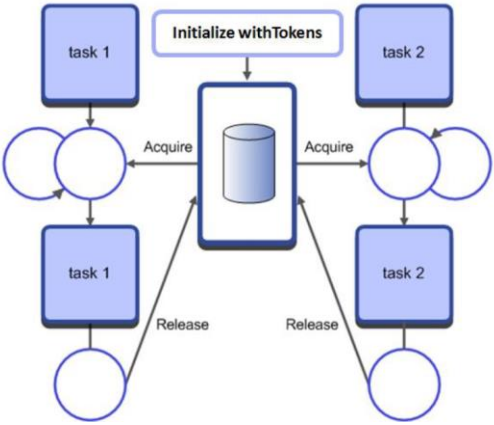


Figura 01 – Semáforo para sincronizar threads

Enquanto isso, a thread em execução pode liberar um token de volta ao semáforo. Quando isso acontecer, a thread em espera adquirirá o token e sairá do estado de espera para o estado pronto. Uma vez no estado pronto, o escalonador colocará a thread no estado de execução para que a execução da thread possa continuar. Embora os semáforos tenham um conjunto simples de chamadas do sistema operacional, eles podem ser um dos objetos do sistema operacional mais difíceis de entender completamente.

Nesta seção, primeiro veremos como adicionar semáforos a um programa RTOS e, em seguida, analisaremos as aplicações mais úteis de semáforos.

1) Adicionando Semáforos a um Programa RTOS

Para usar um semáforo no CMSIS-RTOS, você deve primeiro declarar um contêiner de semáforo:

```
osSemaphoreId sem1;
osSemaphoreDef(sem1);
```

Então, dentro de uma thread, o contêiner de semáforo pode ser inicializado com um número de tokens.

```
sem1 = osSemaphoreCreate(osSemaphore(sem1), SIX_TOKENS);
```

É importante entender que tokens de semáforo também podem ser criados e destruídos conforme as threads são executadas. Por exemplo, você pode inicializar um semáforo com zero tokens e, em seguida, usar uma thread para criar tokens no semáforo enquanto outra thread os remove. Isso permite projetar threads como threads produtoras e consumidoras.

Uma vez que o semáforo é inicializado, tokens podem ser adquiridos e enviados ao semáforo de maneira semelhante às flags de eventos. A chamada `os_sem_wait` é usada para bloquear uma thread até que um token de semáforo esteja disponível, semelhante à chamada `os_evnt_wait`. Um período de timeout também pode ser especificado, com 0xFFFF sendo uma espera infinita.

```
osStatus osSemaphoreWait(osSemaphoreId semaphore_id, uint32_t millisec);
```

Uma vez que a thread tenha terminado de usar o recurso do semáforo, ela pode enviar um token ao contêiner de semáforo.

```
osStatus osSemaphoreRelease(osSemaphoreId semaphore_id);
```

2) Exercício de Sinalização de Semáforo

Neste exercício, vamos configurar um semáforo e usá-lo para sinalizar entre duas tarefas. No *Pack Installer*, selecione "**Ex 11 Interrupt Signals**" e copie-o para o seu diretório de tutoriais.

Primeiro, o código cria um semáforo chamado `sem1` e o inicializa com zero tokens.

```
osSemaphoreId sem1;
osSemaphoreDef(sem1);
int main (void) {
  sem1 = osSemaphoreCreate(osSemaphore(sem1), 0);
```

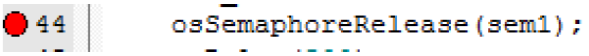
A primeira tarefa espera que um token seja enviado ao semáforo.

```
void led_Thread1 (void const *argument) {
  for (;;) {
    osSemaphoreWait(sem1, osWaitForever);
    LED_On(1);
    osDelay(500);
    LED_Off(1);
  }
}
```

Enquanto a segunda tarefa periodicamente envia um token ao semáforo.

```
void led_Thread2 (void const *argument) {
  for (;;) {
    LED_On(2);
    osSemaphoreRelease(sem1);
    osDelay(500);
    LED_Off(2);
    osDelay(500);
  }
}
```

Compile o projeto e inicie o depurador.
Coloque um ponto de interrupção na tarefa `led_Thread2`.



Execute o código e observe o estado das threads quando o ponto de interrupção for atingido.

ID	Name	Priority	State
255	os_idle_demon	0	Ready
3	led_Thread1	5	Wait_SEM
2	led_Thread2	4	Running
1	main	4	Ready

Agora `**led_Thread1**` está bloqueada esperando para adquirir um token do semáforo. `**led_Thread1**` foi criada com uma prioridade mais alta que `**led_Thread2**`, então assim que um token for colocado no semáforo, ela passará para o estado pronto, preemptando a tarefa de menor prioridade e começando a executar. Quando atingir a chamada `**osSemaphoreWait**()`, ela novamente será bloqueada.
Agora, avance passo a passo o código (F10) e observe a ação das threads e do semáforo.

3) Usando Semáforos

Embora os semáforos tenham um conjunto simples de chamadas do sistema operacional, eles têm uma ampla gama de aplicações de sincronização. Isso os torna talvez o objeto de sistema operacional mais desafiador de entender. Nesta seção, veremos os usos mais comuns dos semáforos. Estes são retirados do "*The Little Book Of Semaphores*" de Allen B. Downey.

4) Sinalização

Sincronizar a execução de duas threads é o uso mais simples de um semáforo:

```
osSemaphoreId sem1;
osSemaphoreDef(sem1);
void thread1 (void) {
```

```
  sem1 = osSemaphoreCreate(osSemaphore(sem1), 0);
  while(1) {
    FuncA();
    osSemaphoreRelease(sem1);
  }
}
void task2 (void) {
  while(1) {
    osSemaphoreWait(sem1, osWaitForever);
    FuncB();
  }
}
```

Neste caso, o semáforo é usado para garantir que o código em `FuncA()` seja executado antes do código em `FuncB()`.

5) Multiplexação

Um multiplex é usado para limitar o número de threads que podem acessar uma seção crítica de código. Por exemplo, isso pode ser uma rotina que acessa recursos de memória e só pode suportar um número limitado de chamadas.

```
osSemaphoreId multiplex;
osSemaphoreDef(multiplex);
void thread1 (void) {
  multiplex = osSemaphoreCreate(osSemaphore(multiplex), FIVE_TOKENS);
  while(1) {
    osSemaphoreWait(multiplex, osWaitForever);
    ProcessBuffer();
    osSemaphoreRelease(multiplex);
  }
}
```

Neste exemplo, inicializamos o semáforo multiplex com 5 tokens. Antes que uma thread possa chamar a função **`ProcessBuffer()`**, ela deve adquirir um token de semáforo.

Uma vez que a função tenha sido concluída, o token é enviado de volta ao semáforo. Se mais de cinco threads estiverem tentando chamar `**ProcessBuffer()**`, a sexta deve esperar até que uma thread tenha terminado com `**ProcessBuffer()**` e devolva seu token. Assim, o semáforo multiplex garante que um máximo de cinco threads possam chamar a função `**ProcessBuffer()**` "simultaneamente".

6) Exercício de Multiplexação

Neste exercício, vamos usar um semáforo para controlar o acesso a uma função criando um multiplex.
No *Pack Installer*, selecione "**Ex 12 Multiplex**" e copie-o para o seu diretório de tutoriais.
O projeto cria um semáforo chamado `**semMultiplex**`, que contém um token. Em seguida, são criadas seis instâncias de uma thread contendo um semáforo multiplex.
Compile o código e inicie o depurador.
Abra a janela Peripherals -> General Purpose IO -> GPIOB.
Execute o código e observe como as tarefas definem os pinos da porta. À medida que o código é executado, apenas uma thread por vez pode acessar as funções de LED, de modo que apenas um pino de porta é definido.
Saia do depurador e aumente o número de tokens alocados ao semáforo quando ele for criado:

```
semMultiplex = osSemaphoreCreate(osSemaphore(semMultiplex), 3);
```

Compile o código e inicie o depurador. Execute o código e observe os pinos GPIOB.
Agora, três threads podem acessar as funções de LED "concomitantemente".

7) Rendezvous (Barreira simples)

Uma forma mais generalizada de sinalização de semáforo é um rendez-vous (barreira). Um rendezvous garante que duas threads alcancem um certo ponto de execução. Nenhuma pode continuar até que ambas tenham atingido o ponto de rendezvous.

```
osSemaphoreId arrived1, arrived2;
osSemaphoreDef(arrived1);
osSemaphoreDef(arrived2);
```

```
void thread1 (void) {
  arrived1 = osSemaphoreCreate(osSemaphore(arrived1), ZERO_TOKENS);
  arrived2 = osSemaphoreCreate(osSemaphore(arrived2), ZERO_TOKENS);
  while (1) {
    FuncA1();
    osSemaphoreRelease(arrived1);
    osSemaphoreWait(arrived2, osWaitForever);
    FuncA2();
  }
}

void thread2 (void) {
  while (1) {
    FuncB1();
    osSemaphoreRelease(arrived2);
    osSemaphoreWait(arrived1, osWaitForever);
    FuncB2();
  }
}
```

Neste caso, os dois semáforos garantirão que ambas as threads se encontrarão na barreira e, em seguida, procederão para executar FuncA2() e FuncB2().

8) Exercício: Barreira simples

Neste projeto, criaremos duas tarefas e garantiremos que elas tenham alcançado um rendezvous de semáforo antes de executar as funções de LED.

- No *Pack Installer,* selecione **"Ex 13 Rendezvous**" e copie para seu diretório de tutorial.
- Compile o projeto e inicie o depurador.
- Abra a janela Peripherals -> General Purpose IO -> GPIOB.

- Execute o código: Inicialmente, o código do semáforo em cada uma das tarefas de LED está comentado. Como as threads não estão sincronizadas, os pinos do GPIO vão alternar aleatoriamente.
- Saia do depurador.
- Descomente o código do semáforo nas tarefas de LED.
- Compile o projeto e inicie o depurador.
- Execute o código e observe a atividade dos pinos na janela do GPIOB.

Agora, as tarefas estão sincronizadas pelo semáforo e executam as funções de LED "concorrentemente".

9) Turnstile de Barreiras

Embora um rendezvous seja muito útil para sincronizar a execução de código, ele funciona apenas para duas funções. Uma barreira é uma forma mais generalizada de rendezvous que funciona para sincronizar múltiplas threads:

```
osSemaphoreId count, barrier;
osSemaphoreDef(counter);
osSemaphoreDef(barrier);

unsigned int count;

void thread1 (void) {
 count = osSemaphoreCreate(osSemaphore(count), ONE_TOKEN);
 barrier = osSemaphoreCreate(osSemaphore(barrier), ZERO_TOKENS);
 while (1) {
  // Permitir apenas uma tarefa por vez a executar este código
  osSemaphoreWait(counter, osWaitForever);
  count++;
  if (count == 5) osSemaphoreRelease(barrier);
  osSemaphoreRelease(counter);
```

```
    // Quando todas as cinco tarefas tiverem chegado, a barreira é aberta
    osSemaphoreWait(barrier, osWaitForever);
    osSemaphoreRelease(barrier);
    critical_Function();
  }
}
```

Neste código, usamos uma variável global para contar o número de threads que chegaram à barreira. À medida que cada função chega à barreira, ela espera até que possa adquirir um token do semáforo contador. Uma vez adquirido, a variável count será incrementada em um. Uma vez que tenhamos incrementado a variável count, um token é enviado de volta ao semáforo contador para que outras threads esperando possam proceder. Em seguida, o código da barreira lê a variável count. Se for igual ao número de threads que estão esperando para chegar à barreira, enviamos um token para o semáforo da barreira.

Neste exemplo, estamos sincronizando cinco threads. As primeiras quatro threads incrementarão a variável count e, em seguida, esperarão no semáforo da barreira. A quinta e última thread a chegar incrementará a variável count e enviará um token para o semáforo da barreira. Isso permitirá que ela adquira imediatamente um token do semáforo da barreira e continue a execução.

Após passar pela barreira, ela envia imediatamente outro token para o semáforo da barreira. Isso permite que uma das outras threads em espera retome a execução. Essa thread coloca outro token no semáforo da barreira, que aciona outra thread em espera e assim por diante. Esta seção final do código da barreira é chamada de catraca porque permite que uma thread por vez passe pela barreira. Em nosso modelo de execução concorrente, isso significa que cada thread espera na barreira até que a última chegue, então todas retomam simultaneamente.

No exercício a seguir, criaremos cinco instâncias de uma thread contendo o código da barreira. No entanto, a barreira pode ser usada para sincronizar cinco threads únicas.

10) Exercício: Barreira de Semáforo

Neste exercício, usaremos semáforos para criar uma barreira para sincronizar múltiplas tarefas.

- No *Pack Installer*, selecione "**Ex 14 Barrier**" e copie para seu diretório de tutorial.
- Compile o projeto e inicie o depurador.
- Abra a janela Peripherals -> General Purpose IO -> GPIOB.
- Execute o código: Inicialmente, o código do semáforo em cada uma das threads está comentado. Como as threads não estão sincronizadas, os pinos do GPIO vão alternar aleatoriamente como no exemplo do rendezvous.
- Saia do depurador.
- Remova os comentários nas linhas 34, 45, 53 e 64 para habilitar o código da barreira.
- Compile o projeto e inicie o depurador.
- Execute o código e observe a atividade dos pinos na janela do GPIOB.

Agora, as tarefas estão sincronizadas pelo semáforo e executam as funções de LED "concorrentemente".

11) Advertências sobre Semáforos

Os semáforos são um recurso extremamente útil de qualquer RTOS. No entanto, os semáforos podem ser mal utilizados. Você deve sempre lembrar que o número de tokens em um semáforo não é fixo. Durante a execução de um programa, os tokens do semáforo podem ser criados e destruídos. Às vezes isso é útil, mas se seu código depende de ter um número fixo de tokens disponíveis em um semáforo, você deve ter muito cuidado para sempre devolver os tokens a ele. Você também deve descartar a possibilidade de criar acidentalmente novos tokens adicionais.

12) MutEx

Mutex significa "Exclusão Mútua". Na realidade, um mutex é uma versão especializada de semáforo. Assim como um semáforo, um mutex é um contêiner para tokens. A diferença é que um mutex só pode conter um token que não pode ser criado ou destruído. O uso principal de um mutex é controlar o acesso a um recurso do chip, como um periférico. Por esta razão, um token de mutex é binário e delimitado. Além disso, ele realmente funciona da mesma maneira que um semáforo. Primeiro, devemos declarar o contêiner de mutex e inicializar o mutex:

```
osMutexId uart_mutex;
osMutexDef(uart_mutex);
```

Uma vez declarado, o mutex deve ser criado em uma thread:

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

Então, qualquer thread que precise acessar o periférico deve primeiro adquirir o token do mutex:

```
osMutexWait(osMutexId mutex_id, uint32_t millisec);
```

Finalmente, quando terminamos de usar o periférico, o mutex deve ser liberado:

```
osMutexRelease(osMutexId mutex_id);
```

O uso de mutex é muito mais rígido do que o uso de semáforo, mas é um mecanismo muito mais seguro ao controlar o acesso absoluto aos registradores do chip subjacentes.

13) Exercício: Mutex

Neste exercício, nosso programa escreve fluxos de caracteres na UART do microcontrolador a partir de diferentes threads. Vamos declarar e usar um mutex para garantir que cada thread tenha acesso exclusivo à UART até que tenha terminado de escrever seu bloco de caracteres.

- No *Pack Installer*, selecione "**Ex 15 Mutex**" e copie para seu diretório de tutorial.

Este projeto declara duas threads que escrevem blocos de caracteres na UART. Inicialmente, o mutex está comentado:

```
void uart_Thread1 (void const *argument) {
  uint32_t i;
  for (;;) {
    //osMutexWait(uart_mutex, osWaitForever);
    for (i = 0; i < 10; i++) SendChar('1');
    SendChar('\n');
    SendChar('\r');
    //osMutexRelease(uart_mutex);
  }
}
```

Em cada thread, o código imprime o número da thread. No final de cada bloco de caracteres, ele então imprime os caracteres de retorno de carro e nova linha.

- Compile o código e inicie o depurador.
- Abra a janela do console UART1 com View -> Serial Windows -> UART #1.
- Inicie o código e observe a saída na janela do console: Aqui podemos ver que o fluxo de dados de saída está corrompido por cada thread escrevendo na UART sem qualquer controle de acesso.
- Saia do depurador.
- Descomente as chamadas de mutex em cada thread.
- Compile o código e inicie o depurador.
- Observe a saída de cada tarefa na janela do console.

Agora o mutex garante que cada tarefa tenha acesso exclusivo à UART enquanto escreve cada bloco de caracteres.

14) Advertências sobre Mutex

Claramente, você deve ter cuidado para devolver o token do mutex quando terminar de usar o recurso do chip, ou você terá efetivamente impedido que qualquer outra thread acesse-o. Você também deve ser extremamente cuidadoso ao usar a chamada osThreadTerminate() em funções que controlam um token de mutex. Keil RTX é projetado para ser um RTOS de pequeno porte para que possa rodar até mesmo nos microcontroladores Cortex-M muito pequenos. Consequentemente, não há segurança de exclusão de threads. Isso significa que se você deletar uma thread que está controlando um token de mutex, você destruirá o token de mutex e impedirá qualquer acesso futuro ao periférico protegido.

15) Aplicação prática:

15.1) Sincronização de Sensores e Atuadores:

Considere um sistema de automação industrial que coleta dados de sensores e controla atuadores. O desafio é sincronizar a leitura de sensores e a acionamento de atuadores para garantir a precisão e confiabilidade do sistema.

Tarefas:

- Crie duas threads: uma para ler dados de sensores e outra para controlar atuadores.

- Utilize semáforos para garantir que a thread de leitura de sensores termine de coletar os dados antes que a thread de controle dos atuadores seja acionada.
- Implemente um mecanismo para lidar com situações de erro, como falhas na leitura de sensores ou na comunicação com atuadores.

15.2) Gerenciamento de Fila de Impressão:

Considere um sistema de impressão recebe diversos arquivos de diferentes usuários e os imprime em uma única impressora. O desafio é gerenciar a fila de impressão de forma eficiente, garantindo que os arquivos sejam impressos na ordem correta e evitando conflitos de acesso à impressora.

Tarefas:

- Crie uma thread para gerenciar a fila de impressão, recebendo novos arquivos e adicionando-os à fila.
- Utilize semáforos para controlar o acesso à impressora, garantindo que apenas um arquivo seja impresso por vez.
- Implemente um mecanismo de priorização para definir a ordem de impressão dos arquivos, considerando fatores como tamanho, urgência ou tipo de arquivo.

15.3) Comunicação Inter-thread com Semáforos:

Considere que duas threads precisam trocar informações entre si, mas não podem acessar diretamente a memória compartilhada por motivos de segurança. O desafio é estabelecer um canal de comunicação seguro e eficiente entre as threads utilizando semáforos.

Tarefas:

- Crie duas threads: uma para enviar dados e outra para recebê-los.
- Utilize semáforos para sincronizar o envio e recebimento de dados, evitando perda de informações ou corrupção de dados.
- Implemente um mecanismo de buffer para armazenar os dados temporariamente enquanto a thread receptora não está pronta para recebê-los.