LABORATÓRIO RTOS

Prática 06: Gerenciamento de Recursos

Prof. Francisco Helder

17 de agosto de 2024

1 Controlando acesso com mutex

Há o potencial para um conflito surgir em um sistema multitarefa se uma tarefa começar a acessar um recurso, mas não concluir seu acesso antes de ser transferida para fora do estado Running. Se a tarefa deixou o recurso em um estado inconsistente, o acesso ao mesmo recurso por qualquer outra tarefa ou interrupção pode resultar em corrupção de dados ou outro erro semelhante.

1.1 Exclusão Mútua

O acesso a um recurso compartilhado entre tarefas ou entre tarefas e interrupções deve ser gerenciado usando uma técnica de "exclusão mútua" para garantir que a consistência dos dados seja mantida o tempo todo. O objetivo é garantir que, uma vez que uma tarefa comece a acessar um recurso compartilhado, a mesma tarefa tenha acesso exclusivo até que o recurso tenha retornado a um estado consistente.

O FreeRTOS fornece vários recursos que podem ser usados para implementar a exclusão mútua, mas o melhor método de exclusão mútua é (sempre que possível) projetar o aplicativo de forma que os recursos não sejam compartilhados e cada recurso seja acessado apenas de uma única tarefa.

1.2 Mutex (um tipo de Semáforo Binário)

Um Mutex é um tipo de semáforo binário usado para controlar o acesso a um recurso compartilhado entre duas ou mais tarefas, no qual a palavra MUTEX se origina de "MUTual EXclusion".

Quando usado em cenários de exclusão mútua, o mutex pode ser conceitualmente pensado como um token que está associado ao recurso que está sendo compartilhado. Para acessar o recurso, uma tarefa deve primeiro "pegar" o token com sucesso (ser o detentor do token). Quando o detentor do token tiver terminado com o recurso, ele deve "devolver" o token. Somente quando o token tiver sido devolvido, outra tarefa pode pegar o token com sucesso e então acessar com segurança o mesmo recurso compartilhado. Uma tarefa não pode acessar o recurso compartilhado a menos que tenha o token, como visto na Figura 1.

Embora mutexes e semáforos binários compartilhem muitas características, o cenário mostrado na Figura 1 (onde um mutex é usado para exclusão mútua) é completamente diferente daquele mostrado no semáforo da prática anterior (onde um semáforo binário é usado para sincronização). A principal diferença é o que acontece com o semáforo depois de obtido:

- Um semáforo que é usado para exclusão mútua deve sempre ser retornado;
- Um semáforo que é usado para sincronização é normalmente descartado e não retornado.

O mecanismo funciona puramente por meio da disciplina do escritor do aplicativo. Não há razão para que uma tarefa não possa acessar o recurso a qualquer momento, mas cada tarefa "concorda" em não fazê-lo, a menos que primeiro consiga se tornar o detentor do token.

1.3 Gerenciando Recursos

Nesta prática iremos aprender a proteger e controlar acesso concorrente para um caso específico de uma porta serial.

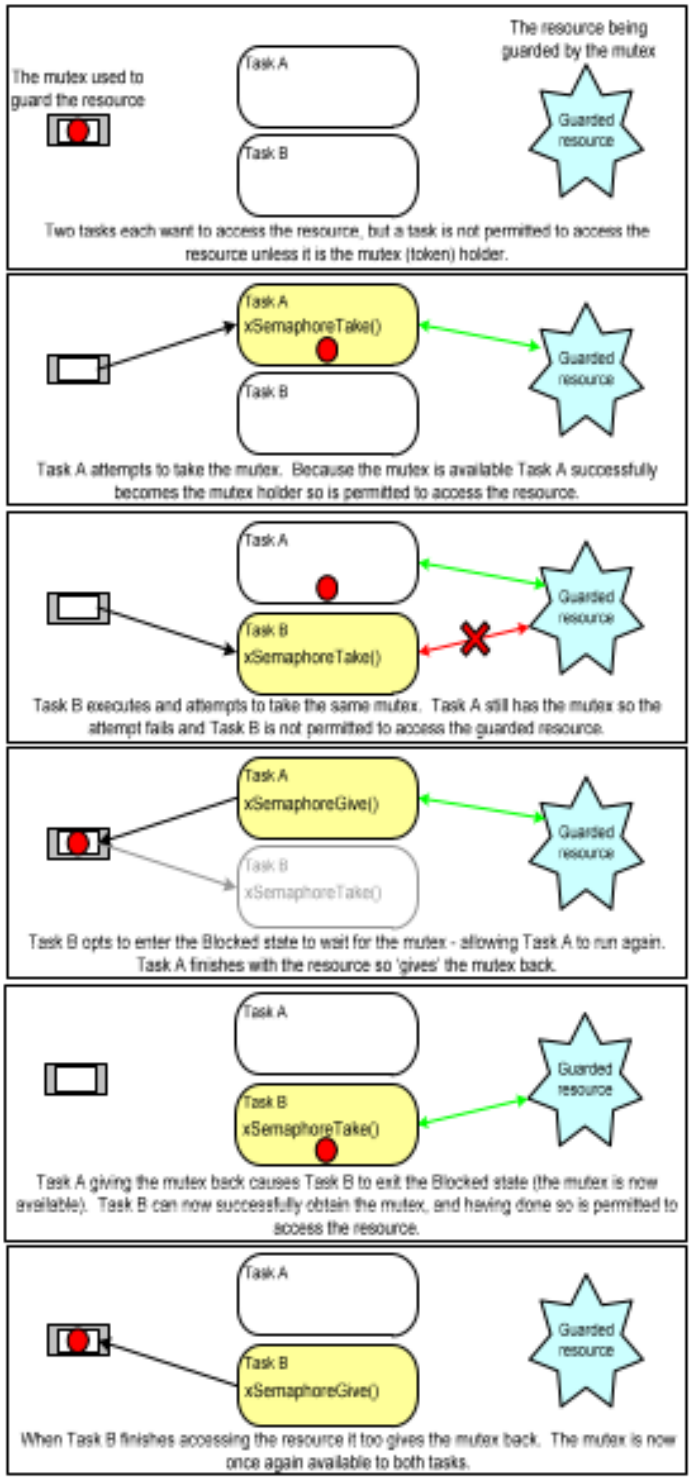

Figura 1: Exclusão mútua implementada usando mutex.

Normalmente trabalhamos com a porta serial através da função printf(), sem se preocupar com o compartilhamento desse recurso. Porém, o acesso concorrente à porta serial não esta protegido, significando que podemos ter problemas se mais de uma tarefa tentar acessar a porta serial ao mesmo tempo, então vamos usar a funcionalidade mutex para resolver esse problema. Vamos criar um projeto para exemplificar o problema de compartilhamento de recursos. No arquivo main.c, remova as tarefas existentes e crie uma nova tarefa:

```
void taskMsg(void *pvParameters){
  char *msg = (char *) pvParameters;
  int i;
  for (;;) {
    printf("%s\n", msg);
    for (i = 0; i < 2000000; i++);
  }
}
```

Então crie duas tarefas na função main():

```
/* create msg task 1 */
xTaskCreate(taskMsg, (signed char *)"TaskMsg1",
configMINIMAL_STACK_SIZE * 4, (void *)"--- 0123456789 ---", 1, NULL);

/* create msg task 2 */
xTaskCreate(taskMsg, (signed char *)"TaskMsg2",
configMINIMAL_STACK_SIZE * 4, (void *)"*** abcdefghij ***", 1, NULL);
```

compile e teste, então anilse o resultado.

Perceba que as mensagens ficaram misturadas, porque ambas tarefas estão acessando um mesmo recurso (porta serial) ao mesmo tempo. Vamos agora proteger o acesso à função printf() com um mutex. No arquivo main.c, defina o mutex que usaremos para proteger o acesso à função printf():

```
/* mutex to synchronise access to printf function */
xSemaphoreHandle printfMutex;
```

Crie o mutex na função main(), após a inicialização do hardware:

```
/* create mutex */
printfMutex = xSemaphoreCreateMutex();
```

E proteja a tarefa que escreve na porta serial com a função printf() usando o mutex criado:

```
void taskMsg(void *pvParameters){
  char *msg = (char *) pvParameters;
  int i;
  for (;;) {
    xSemaphoreTake(printfMutex, portMAX_DELAY);
    printf("%s\n", msg);
    xSemaphoreGive(printfMutex);
    for (i = 0; i < 2000000; i++);
  }
}
```

Compile e teste. Perceba que agora a rotina que envia uma mensagem à porta serial esta protegida de acesso concorrente. Uma outra forma de resolver este problema é controlar o acesso de dentro da função printf().

2 Atividades Práticas

pratica 1:

Crie uma aplicação com as seguintes características:

1. Crie uma estrutura global à aplicação que armazene os dados do acelerômetro.
2. Crie uma tarefa periódica de 50ms para ler o acelerômetro e salvar a leitura na estrutura criada no item 1.
3. Crie uma tarefa periódica de 1 segundo para ler os dados da estrutura, formatar e enviar à porta serial.
4. As tarefas de leitura deverão ter prioridade maior que as tarefas de exibição.
5. Proteja o acesso à estrutura com mutex.