ARM PROGRAMMING

Bùi Quốc Bảo

Resource management

- Nếu nhiều tác vụ cùng truy cập 1 tài nguyên (VD:UART), sẽ dẫn đến đụng độ (conflict).
- VD:
  - Accessing Peripherals
  - Read, Modify, Write Operations
  - Non-atomic Access to Variables
  - Function Reentrancy

Accessing Peripherals

- Task A executes and starts to write the string “Hello world” to the LCD.
- Task A is pre-empted by Task B after outputting just the beginning of the string – “Hello w”.
- Task B writes “Abort, Retry, Fail?” to the LCD before entering the Blocked state.
- Task A continues from the point at which it was pre-empted and completes outputting the remaining characters – “orld”.
- The LCD will now be displaying the corrupted string “Hello wAbort, Retry, Fail?orld”.

Read, Modify, Write Operations

```
/* The C code being compiled. */
   155:        PORTA |= 0x01;
   /* The assembly code produced. */
LDR  R0,[PC,#0x0070]  ; Obtain the address of PORTA
LDR  R1,[R0,#0x00]    ; Read the value of PORTA into R1
MOV  R2,#0x01         ; Move the absolute constant 1 into R2
ORR  R1,R2            ; OR R1 (PORTA) with R2 (constant 1)
STR  R1,[R0,#0x00]    ; Store the new value back to PORTA
```

Nếu trong quá trình này có 1 tác vụ khác có mức ưu tiên cao hơn nhảy vào ghi vào PORTA, dữ liệu sẽ bị sai

Non-atomic Access to Variables

- Khi truy cập vào các biến có độ rộng lớn hơn 32bit (VD: struct), CPU cần nhiều hơn 1 lệnh.
- Quá trình này gọi là Non-atomic Access
- Nếu có 1 tác vụ khác cắt ngang quá trình này, sẽ có thể dẫn đến sai trong dữ liệu.

Function Reentrancy

- Một hàm gọi là “reentrant” nếu nó có thể được gọi cùng lúc trong nhiều tác vụ hoặc ngắt.
- Mỗi tác vụ đều có stack riêng. Nếu hàm chỉ truy cập vào các biến lưu trong stack của tác vụ, hàm đó là “reentrant”

Reentrant function

```
long lAddOneHundered( long lVar1 )
{
/* This function scope variable will also be allocated to the stack
or a register, depending on compiler and optimization level.  Each
task or interrupt that calls this function will have its own copy
of lVar2. */
long lVar2;

   lVar2 = lVar1 + 100;

   /* Most likely the return value will be placed in a CPU register,
   although it too could be placed on the stack. */
   return lVar2;
}
```

Non-reentrant function

```
long lVar1;
long lNonsenseFunction( void )
{
static long lState = 0;
long lReturn;

   switch( lState )
   {
      case 0 : lReturn = lVar1 + 10;
               lState = 1;
               break;
           case 1 : lReturn = lVar1 + 20;
               lState = 0;
               break;
   }
}
```

Mutual Exclusion

- Khi một task truy cập vào 1 tài nguyên, nó sẽ có toàn quyền sử dụng tài nguyên cho đến khi xử lý xong.
- Các tác vụ thường được thiết kế sao cho các tài nguyên không được chia sẻ và chỉ được truy cập bởi đúng 1 tác vụ.

Basic Critical Sections

- Là section nằm giữa 2 macro
  - taskENTER_CRITICAL()
  - taskEXIT_CRITICAL()

Basic Critical Sections

```
taskENTER_CRITICAL();
    /* A switch to another task cannot occur between the call to
    taskENTER_CRITICAL() and the call to
    taskEXIT_CRITICAL().  Interrupts  may still execute on
    FreeRTOS ports that allow interrupt nesting, but  only
    interrupts whose priority is above the value assigned to the
    configMAX_SYSCALL_INTERRUPT_PRIORITY constant –
    and those interrupts are  not permitted to call FreeRTOS
    API functions. */

   PORTA |= 0x01;
   /* We have finished accessing PORTA so can safely leave the
   critical section. */
   taskEXIT_CRITICAL();
```

```
void vPrintString( const portCHAR *pcString )
{
    /* Write the string to stdout, using a critical section as a
     crude method  of mutual exclusion. */
    taskENTER_CRITICAL();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    taskEXIT_CRITICAL();
if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

taskENTER_CRITICAL

- Khi gọi , taskENTER_CRITICAL các ngắt sẽ bị disable (trừ các ngắt có độ ưu tiên cao hơn *configMAX_SYSCALL_INTERRUPT_PRIORITY*).
- Khi sử dụng các macro trên, critical section phải được thiết kế thật ngắn.

Suspending the Scheduler

- Critical section có thể được thực thi bằng cách disable scheduler.
- Hàm sau đây được dùng để suspend scheduler:
  - void vTaskSuspendAll(void);

Resume the scheduler

- portBASE_TYPE xTaskResumeAll(void);

```
void vPrintString( const portCHAR *pcString )
{
vTaskSuspendScheduler();
   {
      printf( "%s", pcString );
      fflush( stdout );
   }
   xTaskResumeScheduler();
if( kbhit() )
   {
      vTaskEndScheduler();
   }
}
```

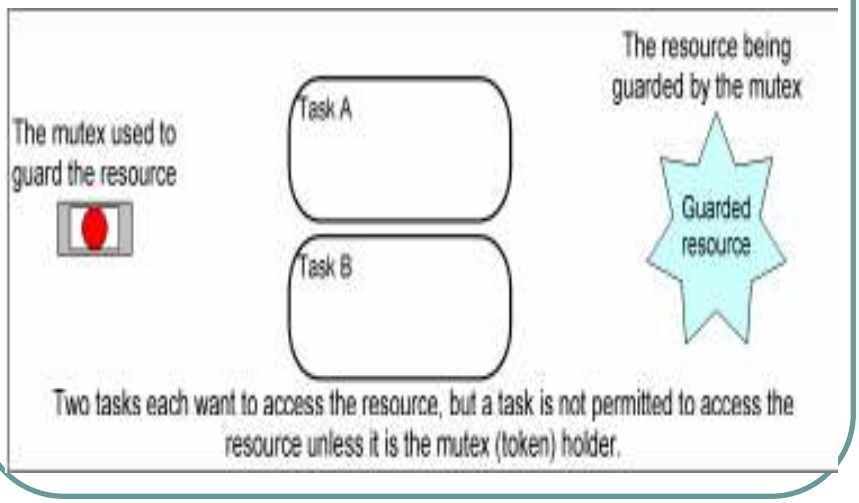
MUTEXES (MUTual EXclusion)

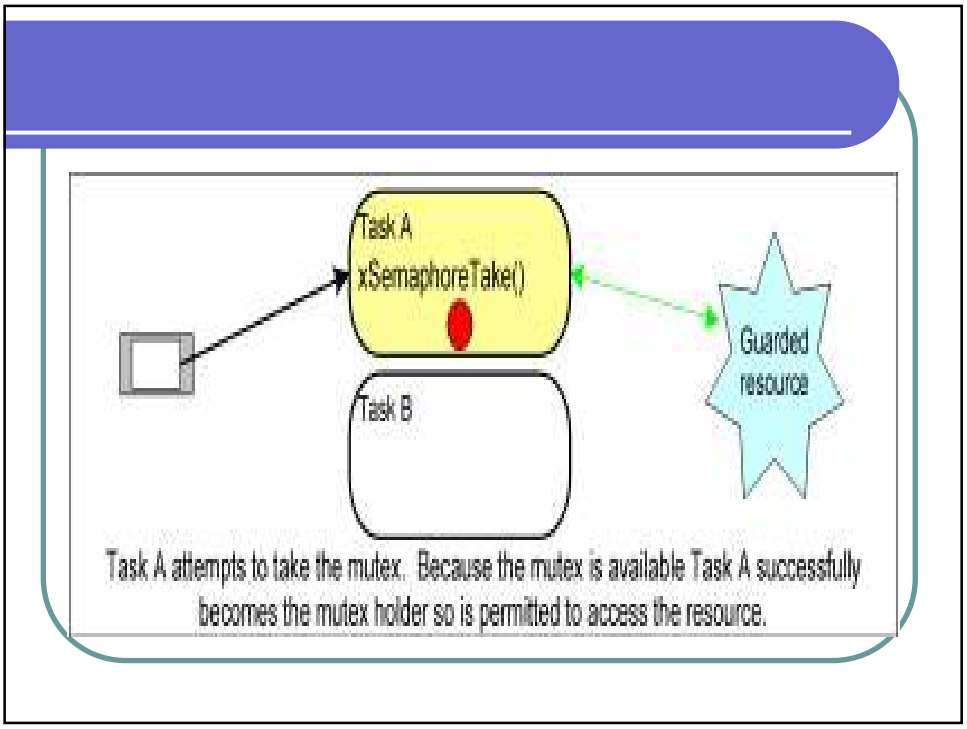
- Mutex là 1 binary semaphore dùng để quản lý việc truy cập tài nguyên.
- Tác vụ muốn truy cập tài nguyên phải lấy 1 “token” hay “key”
- Sau khi sử dụng xong tài nguyên, tác vụ sẽ trả lại token

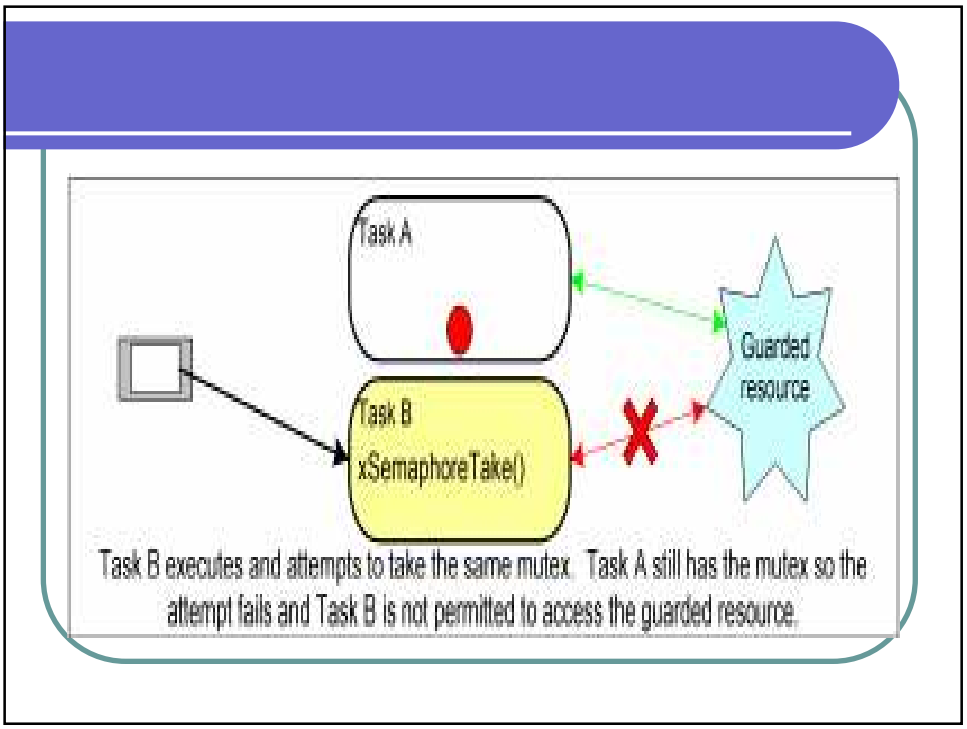
Mutex and synchronization

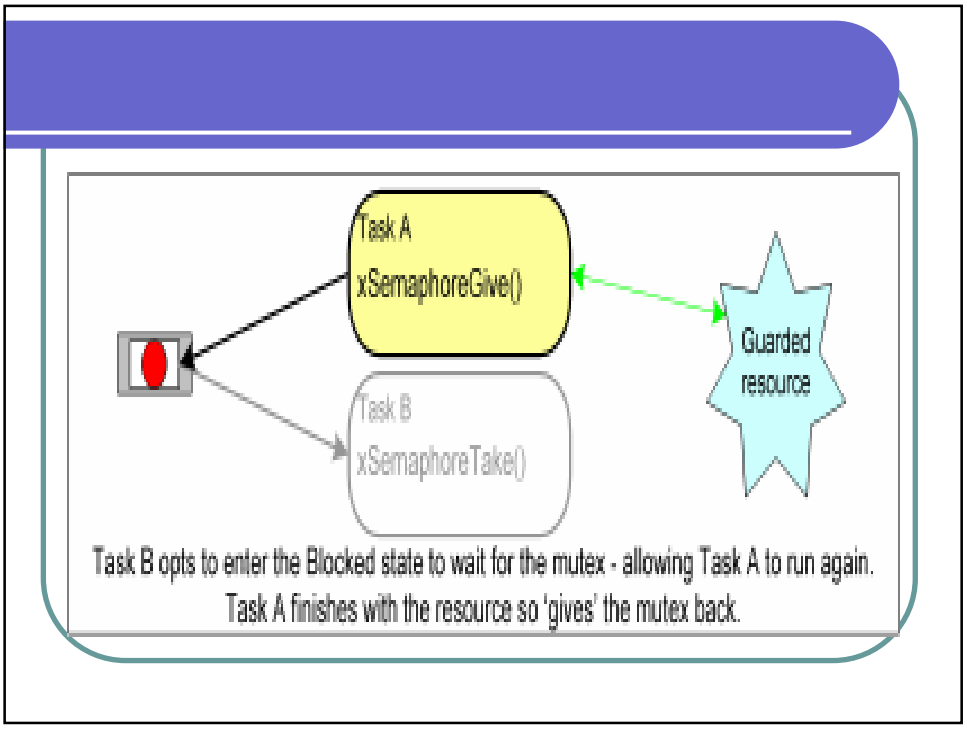
Critical section:
Initial count = 1
Wait_sem to enter critical
Signal_wait to exit critical

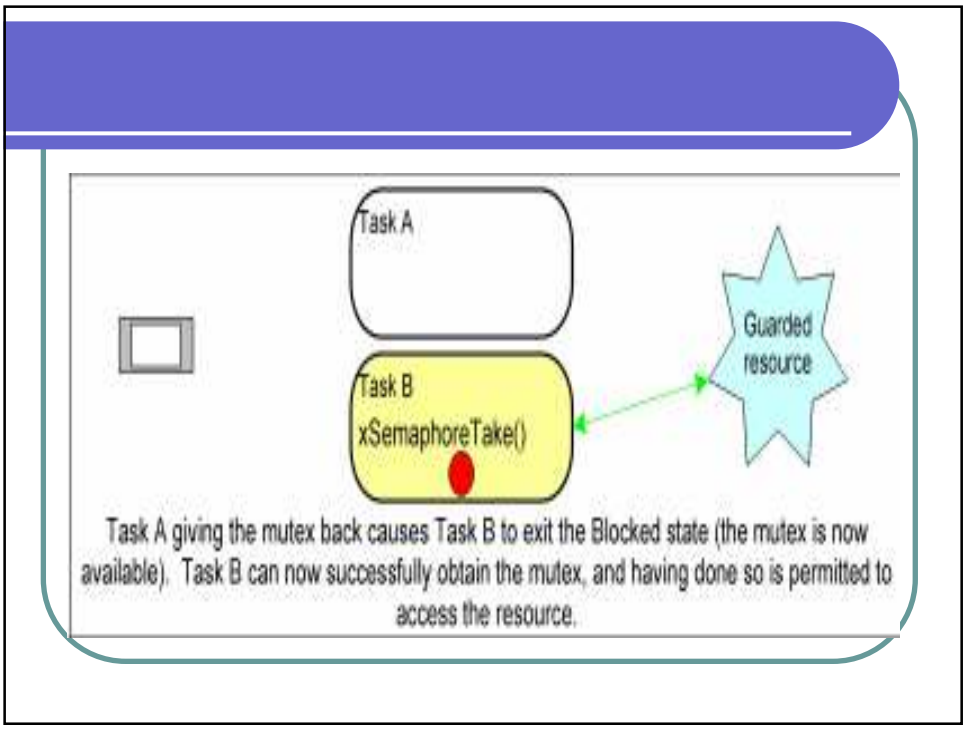
Synchronization:
Initial count = 0
Wait_sem to wait
Signal_wait to signal

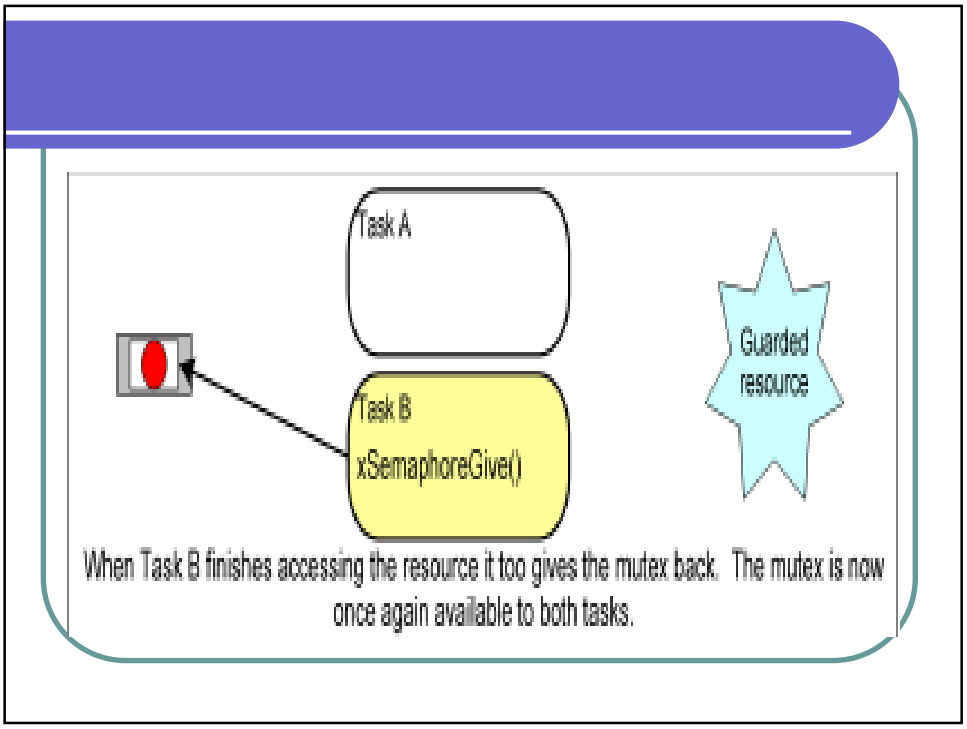



Task A
xSemaphoreTake()
Task B
Guarded resource
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.


Task A
Task B
xSemaphoreTake()
Guarded resource
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.


Task A
xSemaphoreGive()
Guarded resource
Task B
xSemaphoreTake()
Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again.
Task A finishes with the resource so 'gives' the mutex back.


Task A
Guarded resource
Task B
xSemaphoreTake()
Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.



vSemaphoreCreateBinary
vSemaphoreCreateCounting
xSemaphoreCreateMutex
xSemaphoreCreateRecursiveMutex

xSemaphoreGiveFromISR

xSemaphoreTake
xSemaphoreTakeRecursive
xSemaphoreGive
xSemaphoreTakeRecursive

xSemaphoreAltTake
xSemaphoreAltGive

Create mutex xSemaphoreCreateMutex

- xSemaphoreHandle xSemaphoreCreateMutex(void);
- The initial value is 1
  - Returned value:
    - NULL: can not create mutex
    - Non-NUL: handle for mutex

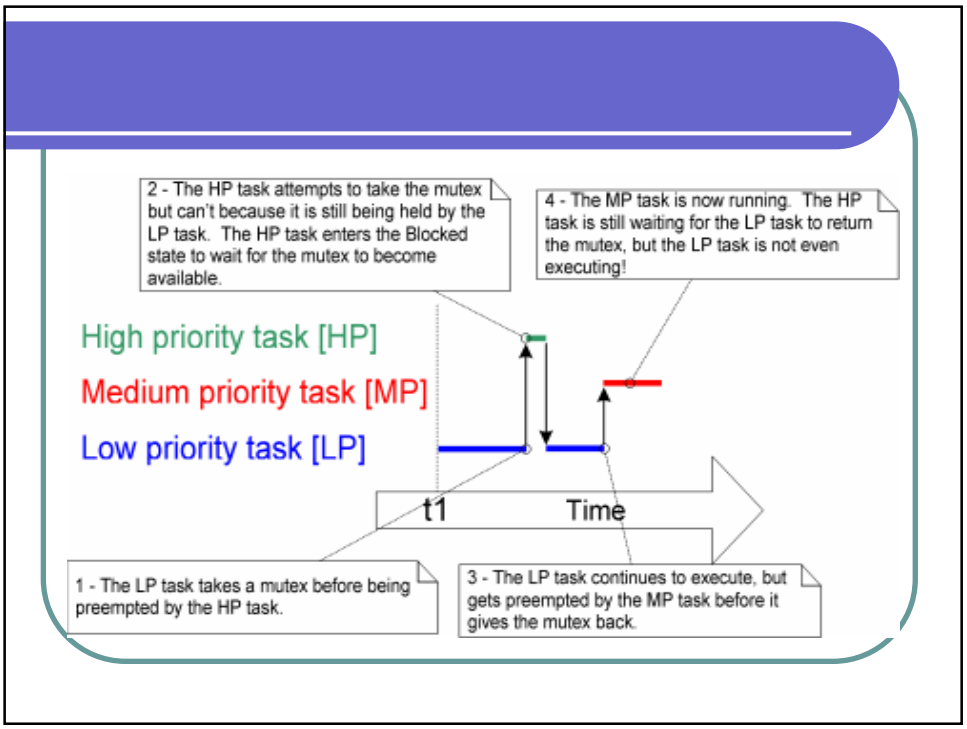
```
static void prvNewPrintString( const portCHAR *pcString )
{
xSemaphoreTake( xMutex, portMAX_DELAY );
   {
          printf( "%s", pcString );
          fflush( stdout );

      /* The mutex MUST be given back! */
   }
   xSemaphoreGive( xMutex );

if( kbhit() )
   {
      vTaskEndScheduler();
   }
}
```

```
static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;
pcStringToPrint = ( char * ) pvParameters;
  for( ;; )
  {
    /* Print out the string using the newly defined
    function. */
    prvNewPrintString( pcStringToPrint );
    vTaskDelay( ( rand() & 0x1FF ) );
  }
}
```

```
int main( void )
{
xMutex = xSemaphoreCreateMutex();
srand( 567 );
if( xMutex != NULL )
   {
xTaskCreate( prvPrintTask, "Print1", 1000,
                "Task 1
    ****************************************\r\n", 1, NULL );
      xTaskCreate( prvPrintTask, "Print2", 1000,
                "Task 2 ----------------------------------------\r\n", 2,
    NULL );
      /* Start the scheduler so the created tasks start executing. */
      vTaskStartScheduler();
   }
   for( ;; );
}
```

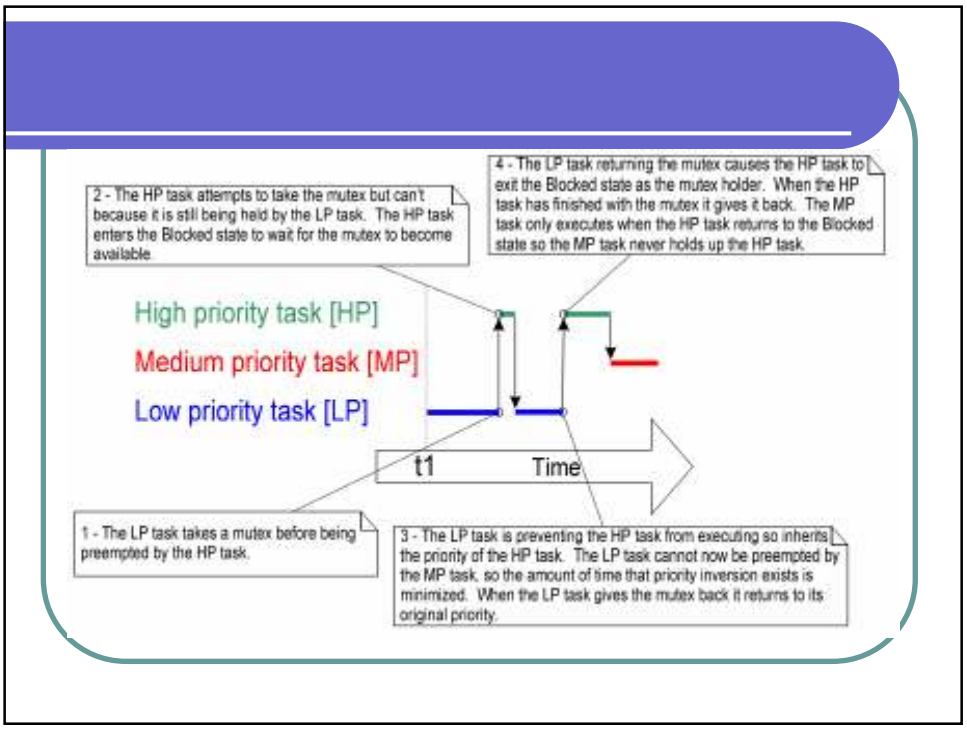
Priority Inversion

- Priority Inversion là khi 1 tác vụ ưu tiên cao phải chờ tác vụ ưu tiên thấp.



Priority Inheritance

- FreeRTOS tạm thời tăng mức ưu tiên của tác vụ đang giữ mutex lên bằng mức ưu tiên cao nhất của các tác vụ đang đòi mutex
- Khả năng này gọi là Priority Inheritance



Deadlock

- Deadlock là trường hợp khi 2 tác vụ chờ resource đang được giữ bởi một trong hai.

- 1. Task A executes and successfully takes mutex X.
- 2. Task A is pre-empted by Task B.
- 3. Task B successfully takes mutex Y before attempting to also take mutex X – but mutex X is held by Task A so is not available to Task B. Task B opts to enter the Blocked state to wait for mutex X to be released.
- 4. Task A continues executing. It attempts to take mutex Y – but mutex Y is held by Task B so is not available to Task A. Task A opts to enter the Blocked state to wait for mutex Y to be released.

Gate keeper task

- Gate keeper là tác vụ duy nhất quản lý tài nguyên.
- Các tác vụ khác muốn sử dụng tài nguyên phải thông qua gatekeeper

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;

for( ;; )
   {

      xQueueReceive( xPrintQueue, &pcMessageToPrint,
    portMAX_DELAY );

      /* Output the received string. */
      printf( "%s", pcMessageToPrint );
      fflush( stdout );
}
}
```

```
static void prvPrintTask( void *pvParameters )
{
int iIndexToString;

    iIndexToString = ( int ) pvParameters;

    for( ;; )
    {
     xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[
     iIndexToString ] ), 0 );
     vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```