AVR FreeRTOS : Resource Management

1. 이 장의 개요

Multitasking System에서 한 Task가 어떤 Resource를 사용 하고 있는 도중에 Running State에서 벗어 나는 일이 생길 수 있고, 이 상태에서 다른 Task나 Interrupt가 동일한 Resource를 사용 하려고 시도 할 수 있다. 이 경우 Data 가 충돌 하거나 손상 될 수 있다.

다음은 이러한 경우가 발생 할 수 있는 예이다.

A. Accessing Peripherals

다음 시나리오는 두 장치가 하나의 LCD에 Data를 서로 쓰려고 시도 하는 경우의 예 이다.

i. Task A가 실행되고 있는 중에 문자열 “Hello world”를 LCD에 출력 하기 시작 하였다.

ii. Task A는 문자열을 “Hello w” 까지 출력한 상태에서 Task B에게 pre-empted(Task A의 Run 상태가 Task B에 의하여 대체 됨) 되었다.

iii. Task B가 Blocked State로 들어가기 전에 “Abort, Retry, Fail?” 메시지를 LCD에 출력 한다.

iv. 다시 Task A가 Run 상태가 되어 “orld”를 LCD에 출력 한다.

그 결과 LCD에는 출력 문자 열이 서로 충동하여 “Hello wAbort, Retry, Fail?orld” 이 출력 된다.

B. Read, Modify, Write Operation

아래의 프로그램 예는 C Code와 이 Code의 AVR128 프로세서 Assembly Language Output 이다. 이 예에서 먼저 PORTD의 값을 Rg24에 읽고, 이 Rg의 값을 Modify 하고, 그 결과를 다시 PORTD에 쓴다. 이러한 과정을 Read, Modify, Write Operation 이라 한다.

```
PORTD |= 0x03; // C code Program

// Assembly Language Output

in      r24, PORTD          ; PORTD 값을 읽고,
```

```
ori     r24, 0x03           ; Modify 하고,
out     PORTD, r24          ; Write 한다.
```

위 예는 명령이 하나 실행된 다음, 나머지 명령이 실행되기 전에 Interrupt가 발생 할 수 있기 때문에 'non-atomic Operation' 이다.

다음의 시나리오는 2개의 Task가 PORTD를 Update 하려고 시도 하는 경우 이다.

i. Task A가 PORTD의 값을 Load 한다. : Read Operation

ii. Task A가 Modify, Write Operation 부분 까지 완료 하기 전에 Task B에게 pre-empted(Task A의 Run 상태가 Task B에 의하여 대체 됨) 되었다.

iii. Task B가 PORTD를 Updates 하고, Blocked 상태가 된다.

iv. Task A가 다시 pre-empted 되어, Rg24의 값을 Modify 하고, PORTD를 Update 한다.

위의 시나리오에서 Task B 에서 수행한 Operation은 Task A에 의하여 변경 되었기 때문에 결과 적으로 PORTD의 값이 잘못된 결과일 수 있다.

위 예는 Peripheral Register(I/O Port)를 사용 한 예이지만 동일한 일이 Global Variables에도 발생 할 수 있다.

C. Variable의 Non-atomic Access

프로세서 Architecture 보다 큰 변수(프로세서가 메모리에 R/W 할 때 한번에 R/W 하는 Word 폭 보다 더 큰 폭(Bit 수)을 갖는 변수)를 Updating 하는 경우 도 또 다른 형태의 Non-atomic Operation 이다.

D. Function Reentrancy

만약 함수가 Task나 Interrupt 로부터 Call 되고, 아직 종료되지 않은 상태에서 다른 하나 이상의 Task나 Interrupt 로부터 안전하게 Call 될 수 있다면 이 함수는 Reentrant 한 함수 이다.

각 Task는 자신의 Stack과 자신의 Core Register Set의 값을 유지(보존) 하고 있다. 만약 함수가 Register에 저장된 Data와 Stack에 저장

된 Data(Local Variable) 만을 사용 한다면 이 함수는 Reentrant 한 함수 이다.

아래 예는 Reentrant Function의 예 이다.

```
/* Function에 Pass되는 Parameter는 CPU Register 또는 Stack를 이용 하기 때문에, 각 Task는 자신의 Stack과 자신의 Core Register Set의 값을 유지(보존) 하여야 한다. 는 조건을 충족 한다. */

Int AddInt( int var1, var2)

{

    /* 이 함수는 Register에 저장된 Data와 Stack을 이용 하는 Data(Local Variable) 만을 사용 한다. */

    Int sum;


    Sum = var1 + var2;

    /* Return 값은 Register와 Stack 만을 이용 하여 전달 한다.

    Return sum;

}
```

아래 예는 Not Reentrant Function의 예 이다.

```
/* sum은 Global Variable로 각 Task가 이 함수를 Call 할 때 변경 될 수 있다. */

Int sum;


Int AddTen( int var1)

{
```

```
    /* 변수 var2는 Static 변수로 Stack에 할당 되지 않는다. 각 Task
    가 이 함수를 Call 할 때 이 변수가 복사되어 반복 사용 된다. */

    static int var2 = 10;

    sum = var1 + var2;


    return sum;

}
```

Mutual Exclusion

Task 사이에 공유되는 자원과 Task와 Interrupt 사이에 공유되는 자원은 'Mutual Exclusion' 기술로 관리 되어 야 한다. 한 Task가 이용 하기 시작한 자원(Resource)은 이 Task의 사용이 종료 되어 Returned 될 때까지 배타적으로 사용 되어야 한다.

FreeRTOS는 Mutual Exclusion을 구현 하는 방법을 제공 한다. 그러나 최상의 방법은 하나의 자원은 하나의 Task만 사용 하도록 설계하는 것이 최선 이다.

이 장의 개요

i. 언제, 왜 Resource Management와 Control이 필요 한가?

ii. Critical Section 이란 무엇인가?

iii. Mutual Exclusion의 의미는 ?

iv. Scheduler를 Suspend 한다는 의미는?

v. Mutex를 어떻게 사용 하나?

vi. Gatekeeper Task의 생성과 사용에 대하여

vii. Priority Inversion에 대한 이해와 어떻게 Priority 상속이 Priority Inversion 때문에 발생 하는 문제를 해결(완전한 해결은 아님) 하는지에 대한 이해

2. Critical Sections과 Scheduler의 Suspending

A. Basic Critical Sections(Regions)

Basic Critical Sections은 아래의 예와 같이 taskENTYER_CRITICAL(), 과 taskEXIT_CRITICAL() Macros에 둘러 싸인 Code 영역 이다.

```
/* Critical Section 내부의 Code가 실행되는 동안에는 Interrupted 되지 않는다 */

taskENTER_CRITICAL();

/* taskENTER_CRITICAL()과 taskEXIT_CRITICAL() 사이에서는 다른 Task로 Switch 되지 않는다. Priority가 configMAX_SYSCALL_INTERRUPT_PRIORITY 상수 보다 높은 Interrupt는 Interrupt Nesting를 할 수 있다. 그러나 이들 Interrupt에서 FreeRTOS API 함수를 Call 하는 것은 허용 되지 않는다. */

PORTD |= 0x01;

taskEXIT_CRITICAL();
```

다음 예는 여러 개의 Task에서 Call 되는 vPrintString() 함수의 예이다. 이 예에서는 한 Task에서 표준 출력 장치를 사용 중일 때 이 장치를 다른 Task에서 Access 하는 것을 Protect 하기 위하여 Critical Section를 사용 한다.

```
Void vPrintString( const char *pcString )

{

/* 문자열을 표준 장치에 출력 하는 동안 다른 Task가 이 장치를 사용하지 못하도록 하는데 Mutual Exclusion의 초보적인 방법으로 Critical Section를 사용하였다. */

 taskENTER_CRITICAL();
```

```
    {
        printf( "%s", pcString );
        fflush( stout );
    }
    taskEXIT_CRITICAL();
}
```

위 예와 같이 Critical Section를 사용하는 것은 매우 초보적인 방법으로 만약 Critical Section 내의 프로그램 Code가 길어지면 Interrupt 응답 이 지연 되는 등 문제가 발생 할 수 있기 때문에 Critical Section내의 Code 길이는 가능한 짧게 하여야 한다. 위 예와 같이 Critical Section 내에 터미널 장치에 출력 하는 Code가 있는 경우에 터미널 장치의 동작은 매우 느리기 때문에 Interrupt를 사용 하는 시스템에 오동작이 발생 할 가능성이 크다. 그러므로 이러한 방식으로 자원(프린터 등)을 사용 하는 것은 피 하여야 한다.

Critical Section은 Kernel이 Nesting Depth를 Count 하기 때문에 Nest 될 수 있다. Nesting Counter 값이 0 일 때 Critical Section이 종료 된다.

B. Scheduler의 Suspending(or Locking)

Critical Section은 Scheduler를 Suspending 하는 것에 의하여 생성 될 수 있다. Scheduler의 Suspending은 Scheduler을 'Locking' 하는 것과 같다.

Scheduler를 Suspending 하는 것에 의하여 구현된 Critical Section은 다른 Task에 의하여 해당 영역 Code를 Protect 하는 것만 가능 하다. 이 경우 Interrupt은 사용 가능 하기 때문에 Interrupt에 의한 Access 가 발생할 가능성이 있다.

C. vTaskSuspendAll() API 함수

vTaskSuspendAll() API Function의 Prototype

```
void vTaskSuspendAll ( void);
```

Scheduler는 vTaskSuspendAll() 함수를 Call 하는 것에 의하여 Suspend 된다.

Scheduler를 Suspending 하면 Context Switch은 발생 하지 않지만, Interrupt는 Enable 될 수 있다. 만약 Scheduler가 Suspending 된 동안 Interrupt에 의한 Context Switching 요구가 발생 하면, 그 요구는 Hold 되고, Scheduler가 Un-suspended 된 후에 실행 된다.

FreeRTOS API 함수는 Scheduler가 Suspending 된 동안 Call 될 수 없다.

D. xTaskResumeAll() API 함수

xTaskResumeAll() API Function의 Prototype

portBASE_TYPE xTaskResumeAll (void);

Scheduler는 xTaskResumeAll() 함수를 Call 하는 것에 의하여 Un-suspend 된다.

Returned value	Description
Returned value	Scheduler가 Suspending 된 동안 요구된 Context Switch는 Hold 되었다 가 Scheduler가 Un-suspended 된 후(그러나 xTaskResumeAll() 함수가 Return 되기 전)에 실행 된 경우 pdTRUE 가 Return 된다. 그 이외의 경우에는 pdFALSE 가 Return 된다.

vTaskSuspendAll()과 xTaskResumeAll()는 Kernel이 Nesting Depth를 Count 하기 때문에 Nest될 수 있다. Nesting Counter 값이 0 일 때 Scheduler는 Un-suspended 된다.

다음 예는 Scheduler를 Suspend 하는 방법에 의하여 터미널 출력을 Protect 하는 예 이다.

```
Void vPrintString( const char *pcString )
```

```
{
/* 문자열을 표준 장치에 출력 하는 동안 다른 Task가 이 장치를 사용
하지 못하도록 하기 위하여 Scheduler를 Suspend 하는 방법에 의하
여 Mutual Exclusion를 구현 하였다. */
 vTaskSuspendScheduler();
    {
            printf( "%s", pcString );
            fflush( stout );
    }
    xTaskResumeScheduler();
}
```

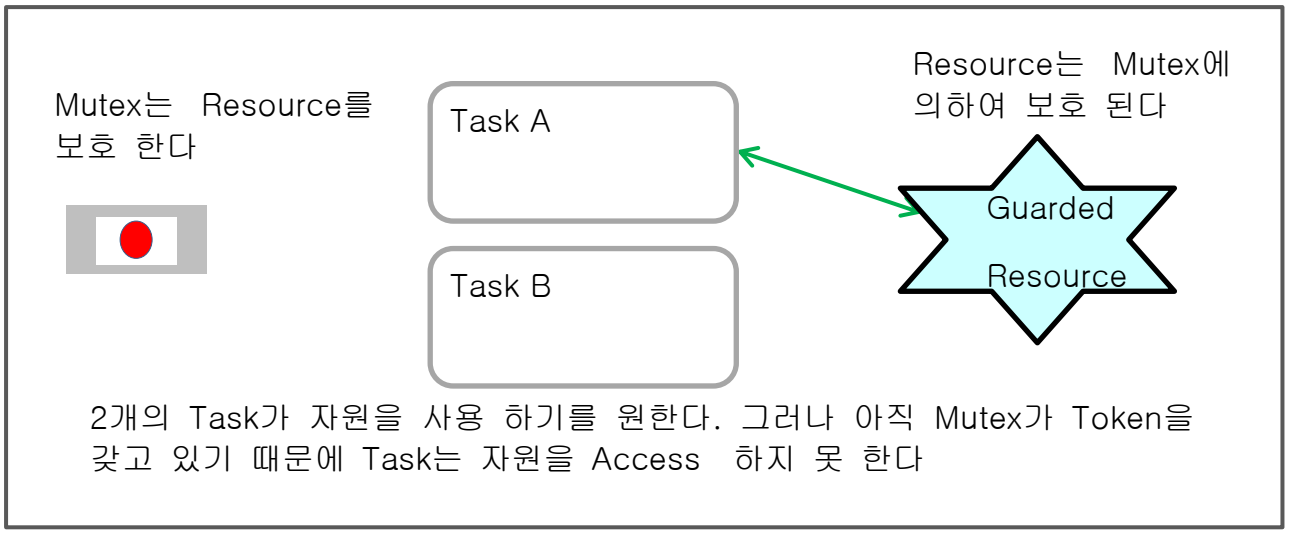
3. Mutexes (and Binary Semaphores)

Mutex는 Binary Semaphore의 특별한 형태로 2개 이상의 Task가 자원을 공유 할 때 사용 한다. MUTEX는 'MUTual Exclusion'의 의미 이다.

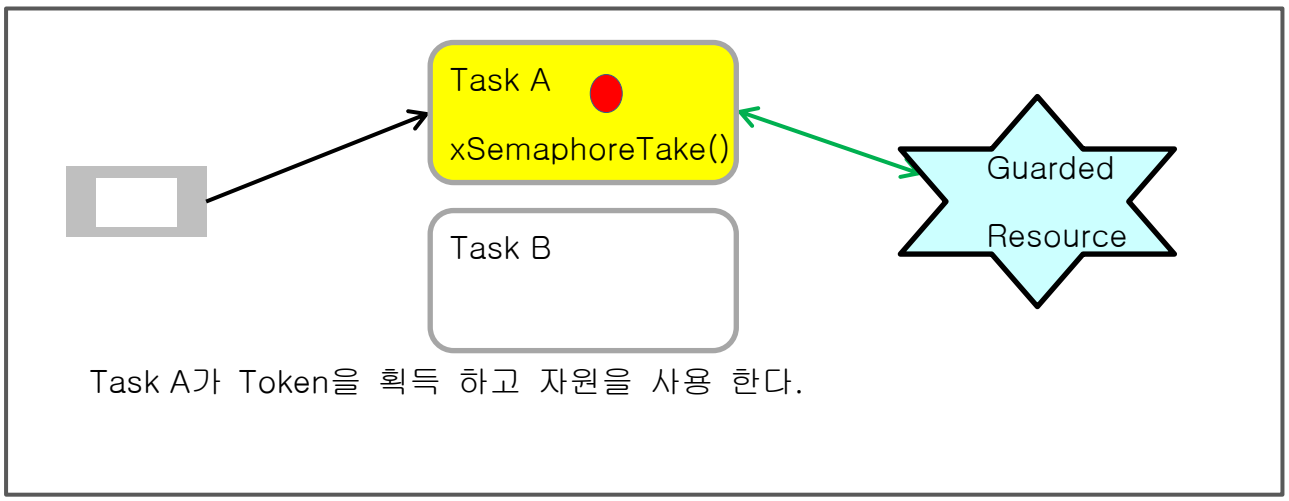
Mutex를 이용하여 자원을 공유하는 경우, Task가 자원을 이용하기 위하여는 먼저 Token을 'Take' 하여야 한다. Token을 갖은 Task는 자원의 사용이 종료되면 Token을 'Give' 하여 다른 Task가 자원을 사용 할 수 있도록 하여야 한다.

이러한 과정의 예는 아래 그림과 같다.

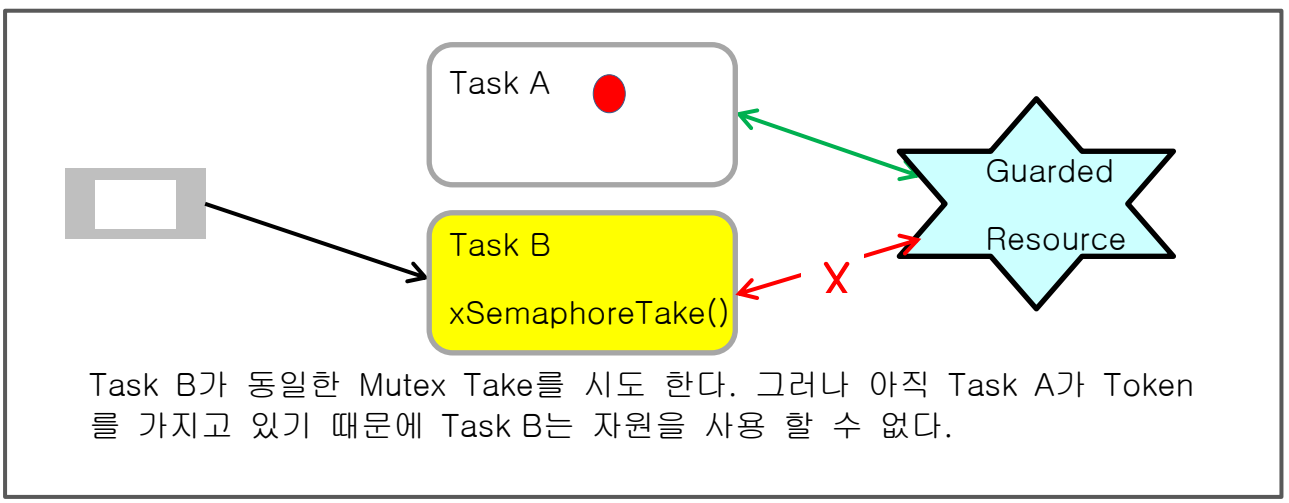
Mutex를 사용한 Mutual Exclusion의 구현



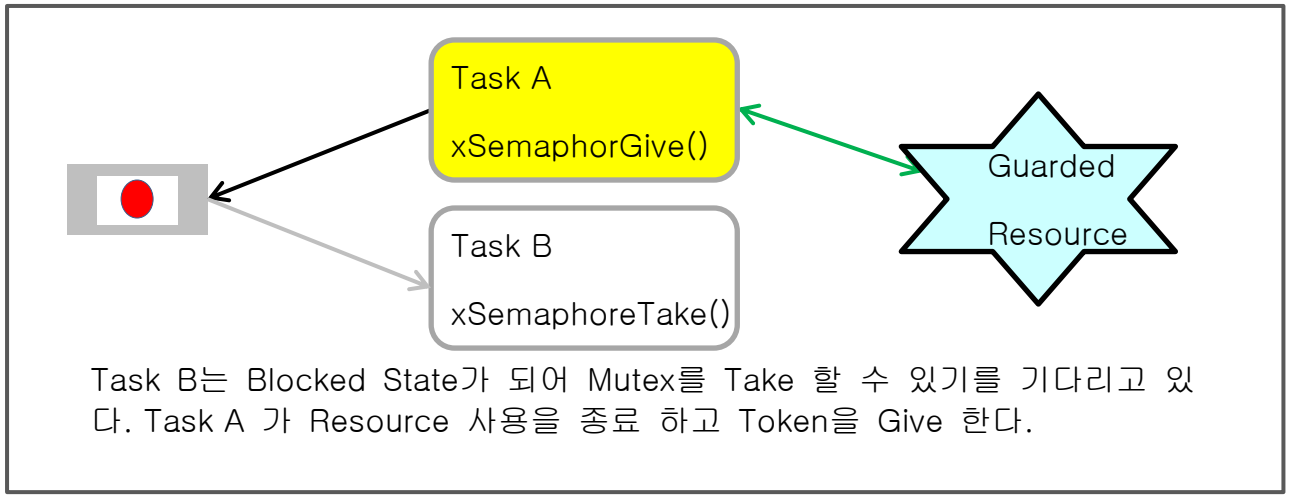
2개의 Task가 자원을 사용 하기를 원한다. 그러나 아직 Mutex가 Token을 갖고 있기 때문에 Task는 자원을 Access 하지 못 한다



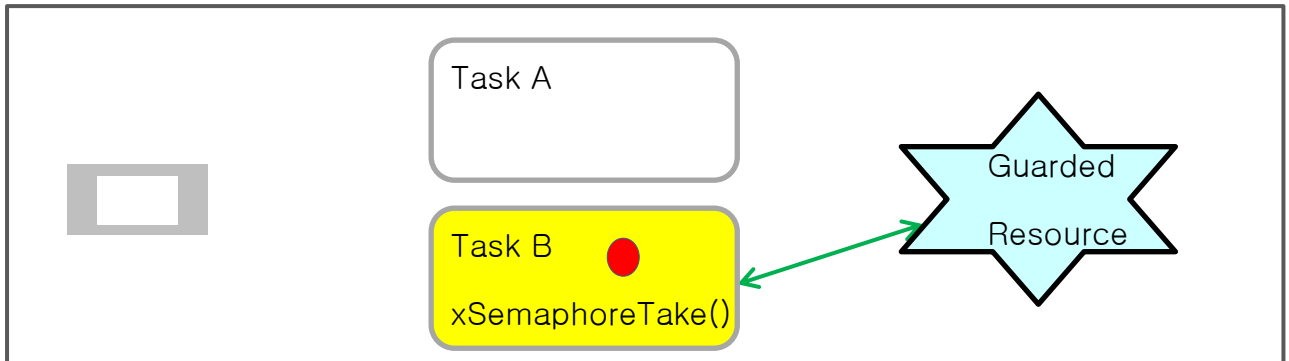
Task A가 Token을 획득 하고 자원을 사용 한다.



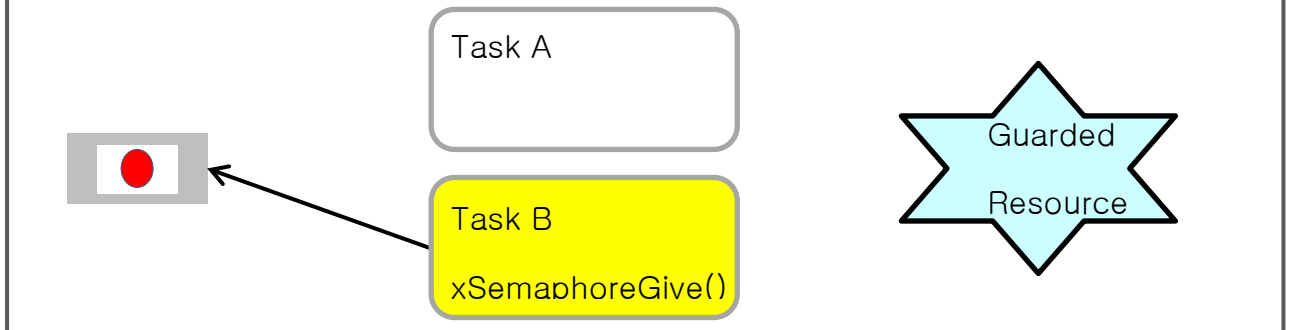
Task B가 동일한 Mutex Take를 시도 한다. 그러나 아직 Task A가 Token를 가지고 있기 때문에 Task B는 자원을 사용 할 수 없다.



Task B는 Blocked State가 되어 Mutex를 Take 할 수 있기를 기다리고 있다. Task A 가 Resource 사용을 종료 하고 Token을 Give 한다.



Task B가 Un-Blocked 되고, Mutex를 획득 하여 Resource를 사용 한다.



Task B가 Resource 사용을 종료 하고 Token을 Give 하여 Mutex는 다시 두 Task가 사용 가능 한 상태가 된다.

A. xSemaphoreCreateMutex() API 함수

Mutex는 Semaphore의 한 형태이다. FreeRTOS Semaphore의 여러 형태의 Handle은 모두 xSemaphoreHandle Type 변수에 저장 된다. Mutex는 사용되기 전에 생성 되어야 한다. Mutex Type Semaphore는 xSemaphoreCreateMutex() API 함수에 의하여 생성 된다.

xSemaphoreCreateMutex() API 함수의 Prototype

xSemaphoreHandle xSemaphoreCreateMutex(void)

Returned value	Description
Returned value	NULL : Mutex data를 저장 하기 위한 Memory 부족 등의 이유로 Mutex 생성에 실패 한 경우 Non-NULL : Mutex가 성공적으로 생성되고 Mutex의 Handle이 Return 된다.

B. MUTEX를 이용한 자원 관리 프로그램 예

1) RT_mutex_LED

```
// 실험 목표
//   Mutex Semaphore를 이용 한 Resource Management
//   vTaskSuspend, vTaskResume를 이용한 Task State Management
//
// 실험 방법
//  처음 프로그램이 실행되면 LED0, LED1, LED2가 랜덤한 속도로 점멸 한다. ( vLED1Task, vLED2Task, vLED3Task 가 모두 Running 하고 있는 상태)
//  SW0 를 누르면 vLED1Task이 Suspending 되고 vLED2Task, vLED3Task 만 Running.
//   SW1 를 누르면 vLED1Task이 다시 Running 상태가 되어 LED0, LED1, LED2가 모두 점멸 한다.
//
// LED
//  LED 0Bit : vLED1Task의 출력( vLED1Task이 Mutex  Semaphore를 획득 하였을 때 Turn On 된다.
//  LED 1Bit : vLED2Task의 출력( vLED2Task이 Mutex  Semaphore를 획득 하였을 때 Turn On 된다.
//  LED 2Bit : vLED3Task의 출력( vLED3Task이 Mutex  Semaphore를 획득 하였을 때 Turn On 된다.
//  LED 6Bit : vLED1Task이 Suspending 상태일 때 Turn On 된다.
```

```
//   LED 7Bit : vLED1Task이 Running 상태일 때 Turn On 된다.

#include <stdio.h>
#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <compat/deprecated.h>

//FreeRTOS include files
#include "FreeRTOS.h"
#include "task.h"
#include "croutine.h"
#include "semphr.h"


//User include files

#define SW_DEBOUNCE_TIME   20

// xSemaphoreHandle type의 변수로서 mutex type semaphore를 참조
하는데 이용 한다.
// LED를 배타적으로 사용 할 수 있도록 하는데 사용 한다.
xSemaphoreHandle xMutex;

// xSemaphoreHandle type의 변수로서 Binary type semaphore를 참조
하는데 이용 한다.
// Switch Interrupt(Interrupt0,1)과 Switch Handling
Task(vtaskControlTask)을 동기 시키는데 사용 한다.
xSemaphoreHandle xKeyInSemaphore = NULL;

xTaskHandle xHandleControl, xHandleLED1, xHandleLED2, xHandleLED3;
unsigned portBASE_TYPE taskPriorityNo;

void vLedControl(char ledNo)
{
        // Semaphore 획득에 실패한 Task는 Semaphore를 획득 할 때
        까지 Blocking 상태가 된다.
        // Semaphore를 획득 하면 자원 사용을 완료 할 때까지 자원을
        사용 할 수 있다.
      xSemaphoreTake( xMutex, portMAX_DELAY );
      {
            // 만약 Semaphore를 성공적으로 획득 한 경우 아래
            Code가 실행 된다.
```

```
                PORTF &= ~0x0f;
                PORTF |= (1 << ledNo);

                // LCD 또는 프린터에 출력 하는 경우 처럼 한번 자원을
                점유하면 일정 시간 사용 하는 효과를 주기 위함.
                vTaskDelay( 100 );
        }
        // 사용이 완료된 자원(Resource)을 되돌려 준다.
        xSemaphoreGive( xMutex );


}

// vLEDTask의 Instance 3개(vLED1Task, vLED2Task, vLED3Task)가
생성되어 실행 되고,
// 각각의 Instance는 "0", "1", "2"를 인수로 전달 받 는다.
// 전달 받은 인수를 수로 변환 하여 해당 위치의 LED를 Turn On 한다.
void vLEDTask( void *pvParameters )
{
        char LEDTaskNo, *pcChar;
        pcChar = ( char * ) pvParameters;

        // 문자열 Pointer(*pvParameters)에 의하여 전달 받은 문자를
        LEDTaskNo에 저장
        LEDTaskNo = *pcChar;
        // LEDTaskNo에 저장돤 숫자 문자를 정수로 변환 한다.
        LEDTaskNo -= '0';

        for( ;; )
        {
                // Task에서 직접 자원을 사용하는 함수를 Call 한다.
                // Mutex를 획득에 성공 하면 자원을 사용 할 수 있고,
                // 실패 하면 이 Task 보다 우선도가 낮은 Task가 자원을
                사용 하고 있는 경우에도 먼저 Mutex를 획득 한 Task가
                Give 할 때까지 Blocked 상태가 지속 된다.
                // 점멸 할 LED 번호를 vLedControl 함수에 전달 한다.
                vLedControl( LEDTaskNo );

                // 이 Task가 자원을 사용 한 후, 랜덤하게 발생된 지연
                시간 동안 Blocked 상태에 있도록 하여, 낮은 우선도의
                Task가 자원을 사용 할 수 있는 기회를 준다.
                vTaskDelay( ( rand() & 0x03FF ) );
        }
```

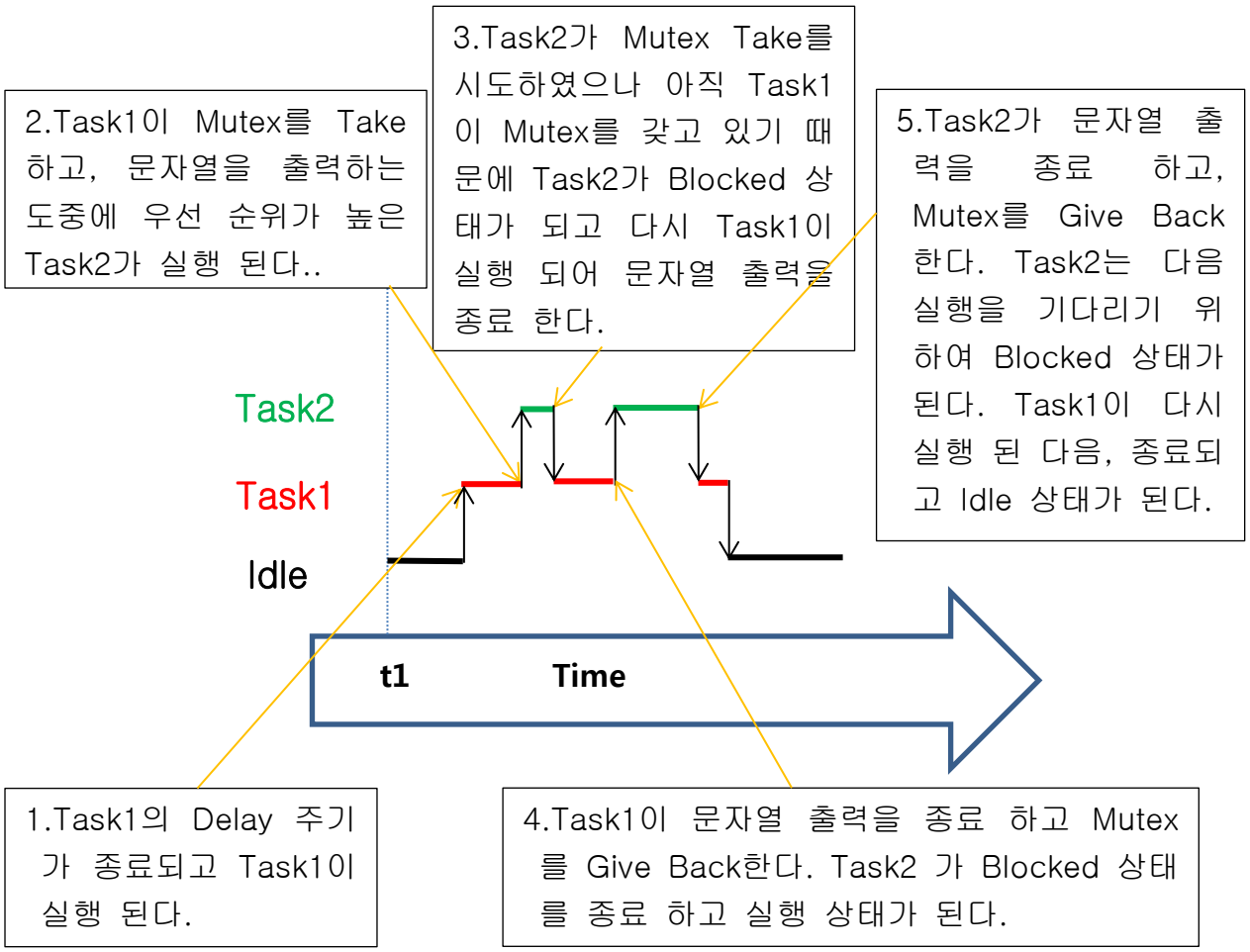
```
}

// SW0, SW1의 상태에 따라 LED1Task을 Suspending 또는 Running
상태 되도록 제어 한다.
void vtaskControlTask( void *pvParameters )
{
        PORTF = 0x80;   // vLED1Task : Running State

        for( ;; )
        {

                if(xSemaphoreTake( xKeyInSemaphore, portMAX_DELAY )
                  == pdTRUE)
                {
                        vTaskDelay(SW_DEBOUNCE_TIME);


                         // SW0가 Push 되면 vLED1Task이 Suspending
                         되고 vLED2Task, vLED3Task 만 Running.
                        if((PIND & 0x01) == 0x00){   // SW0 가 Push 되면
                        vLED1Task를 Suspending 상태로 천이 시킨다.

                          vTaskSuspend(xHandleLED1);
                          PORTF = 0x40; // vLED1Task이 Suspended State
                          인 것을 LED에 표시 한다.
                        }
                        // SW1가 Push 되면 vLED1Task이 다시 Running
                        상태가 되어 LED0, LED1, LED2가 모두 점멸 한다.
                        if((PIND & 0x02) == 0x00){ // SW1 가 Push 되면
                        vLED1Task를 Running 상태로 천이 시킨다.
                            vTaskResume(xHandleLED1);
                            PORTF = 0x80;   // vLED1Task이 Running State
                            인 것을 LED에 표시 한다.
                        }
                EIMSK |= 0x03;   // External Interrupt 0, 1 Enable
                }
        }
}

static void init_device(void);

portSHORT main(void) {
        init_device();
```

```
        // Semaphore는 사용 하기 전에 생성 되어야 한다..
        // Mutex type semaphore 를 create 한다.
        xMutex = xSemaphoreCreateMutex();
        vSemaphoreCreateBinary( xKeyInSemaphore );

        // LED 제어 Task는 각각 Pseudo Random 하게 Delay 된다.
        // Pseudo random delay 를 발생 시키기 위한 Seed
        Number(random)를 설정 한다.
        srand( 567 );

        // Semaphore가 성공적으로 생성 되었는지 확인 한다.
        if( (xMutex != NULL) & (xKeyInSemaphore != NULL) )
        {
            xTaskCreate(vtaskControlTask, (signed portCHAR *)
            "vtaskControlTask", configMINIMAL_STACK_SIZE, NULL,
            tskIDLE_PRIORITY + 4, &xHandleControl   );
            // vLEDTask의 Instance 3개(vLED1Task, vLED2Task,
            vLED3Task)가 생성된다.
            // 각각의 Instance는 "0", "1", "2"를 인수로 전달 한다.
            xTaskCreate(vLEDTask, (signed portCHAR *)"vLED1Task",
            configMINIMAL_STACK_SIZE, "0", tskIDLE_PRIORITY + 1,
            &xHandleLED1 );
            xTaskCreate(vLEDTask, (signed portCHAR *)"vLED2Task",
            configMINIMAL_STACK_SIZE, "1", tskIDLE_PRIORITY + 1,
            &xHandleLED2 );
            // vPrint3Task의 Priority를 가장 높게 한다.
            // 그러나 vPrint1Task, vPrint2Task가 자원을 먼저 선점 하면
            자원 사용이 완료 될 때까지 Blocked 상태에 있게 된다.
            xTaskCreate(vLEDTask, (signed portCHAR *)"vLED3Task",
            configMINIMAL_STACK_SIZE, "2", tskIDLE_PRIORITY + 2,
            &xHandleLED3 );

            // RunSchedular
            vTaskStartScheduler();
        }
        for(;;) {}
}

static void init_device(void){
    cli();              //disable all interrupts
    // output 1, input 0
    outp(0x00,DDRD);
```

```
	outp(0xff,PORTD); //let pull up resistor work, so pin x will be one all
	 the time.
	outp(0xff,DDRF);
	outp(0x00,PORTF);//clear LED.

	EICRA = 0x0a;	// External Interrupt 0, Falling Edge
	   Asynchronously Interrupt
	EIMSK |= 0x03;	// External Interrupt 0, 1 enable
	sei();			//re-enable interrupts
}

// SW0, SW1의 Interrupt 처리를 위한 Interrupt Service Routine
SIGNAL (INT0_vect)
{
	static portBASE_TYPE xHigherPriorityTaskWoken;
	EIMSK &= ~0x03;  // External Interrupt 0, 1 Disable
	xHigherPriorityTaskWoken = pdFALSE;

	/* Semaphore Give 하여 Semaphore 가 준비 되기를
	기다리는(SW가 Push 되기를 기다리는) Blocking 상태의 Task중
	Priority 가 높은 Task가 Unblocking 상태가 되게 한다. */
	xSemaphoreGiveFromISR( xKeyInSemaphore,
	  &xHigherPriorityTaskWoken );

	if( xHigherPriorityTaskWoken == pdTRUE )
	{
		/* Semaphore 'Give'에 의하여 Unblock되는 Task의
		Priority가 현재 실행 중인 Task 보다 높은 경우 Interrupt는
		직접 Unblock된 Task로 Return 된다. */
		taskYIELD();
	}
}
ISR(INT1_vect, ISR_ALIASOF(INT0_vect));
```

2) RT_mutex_printf 프로그램 참고 요

Mutex를 이용한 자원 관리 예(Task2의 Priority 가 Task1보다 높은 경우)

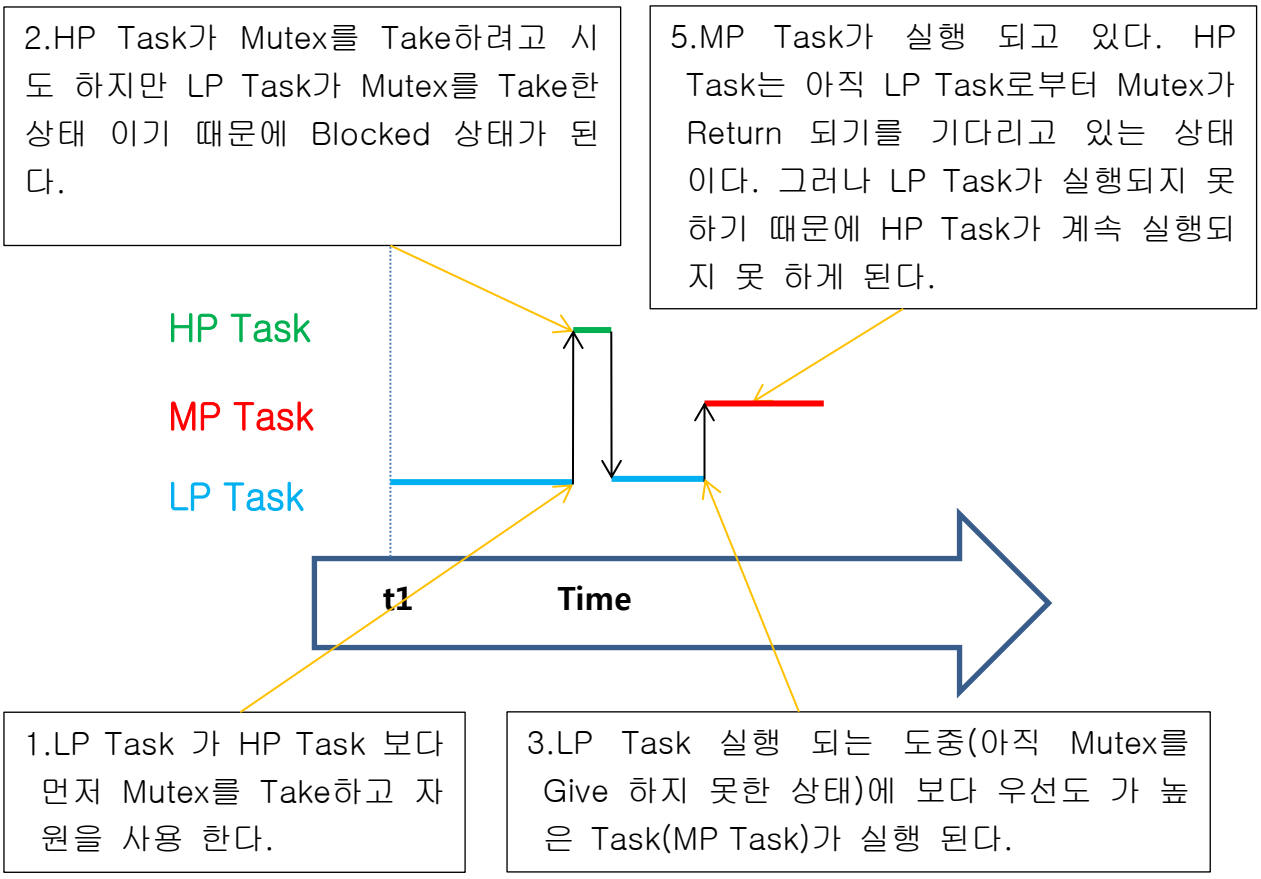


C. Priority Inversion

위 예에서 낮은 Priority Task(LP Task) 가 먼저 Mutex을 Take 하여 높은 Priority Task(HP Task) 가 LP Task 가 Mutex를 Give 할 때 까지 기다리는 경우가 발생 하는 것이 가능 한 것을 알 수 있다. 이와 같이 HP Task가 LP Task에 의하여 Delay 되는 것을 Priority Inversion이라고 한다. 만약의 경우, HP Task가 Semaphore를 기다리는 동안 중간 Priority 을 갖는 Task(MP Task)가 실행되는 경우 LP Task가 실행 기회를 갖지 못하게 되고, LP Task가 Mutex를 Give 하지 못 하기 때문에 HP Task가 계속 실행 되지 못하는 문제가 발생 한다.

이러한 경우의 시나리오 예는 아래 와 같다.

Priority Inversion 경우 중 가장 나쁜 경우의 시나리오

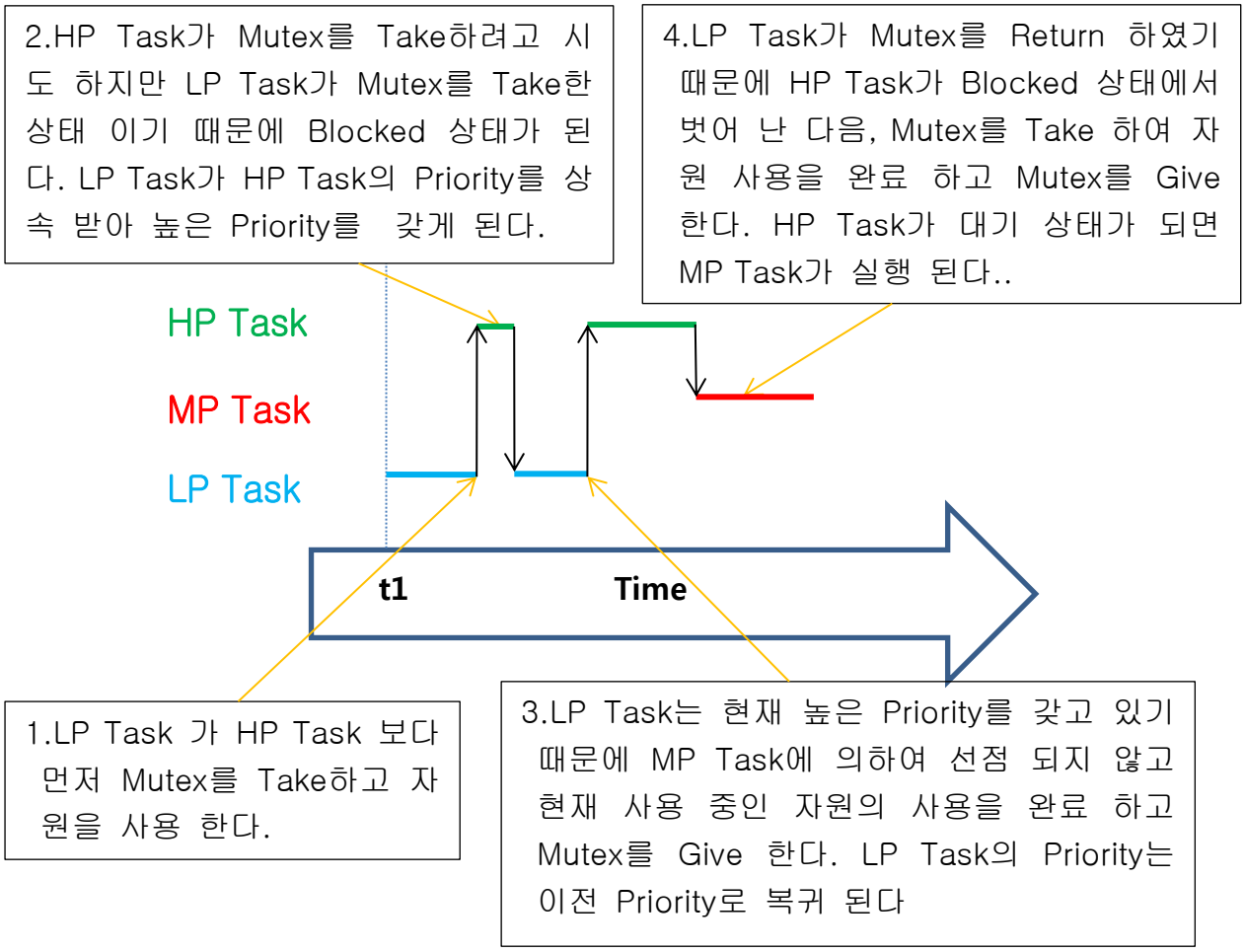


Priority Inversion은 심각한 문제 이지만 소규모 시스템에서는 시스템 설계 시 이러한 문제를 피할 수 있다.

D. Priority 상속

FreeRTOS 에서 Mutex와 Binary Semaphore는 매우 유사 하다.
Mutex와 Binary Semaphore의 차이는 Mutex는 Priority 상속 기능을 갖고 있다는 것 이다. Priority 상속 개념은 Priority Inversion 문제를 최소 화 한다. Priority 상속에 관련된 System Time 해석은 매우 복잡 하고, 또한 Priority Inversion 문제를 완전히 해결 하지는 못 한다.
Priority 상속은 현재 Mutex를 Take 하고 있는 Task 보다 더 높은 Priority를 갖는 Task가 동일한 Mutex를 Take 하려고 시도 하는 경우에 현재 실행 중인 Task의 Priority를 임시적으로 Mutex를 Take 하려고 시도 하는 Task 중에서 가장 높은 Priority를 갖는 Task와 동일 하게 높은 Priority를 부여 하는 것 이다. 현재 Mutex을 갖고 있는 Task가 Mutex를 Give 하는 경우 이 Task는 원래 자신이 갖고 있던 Priority를 갖게 된다. 한 Task는 어느 한 순간에 하나의 Mutex만 갖는 다는 가정 하에 Mutex에 의한 Priority 상속 구조는 구현 되었다.

Priority 상속 기능을 갖는 Mutex를 이용 한 자원 공유 예



E. Deadlock

Mutex를 이용 하여 자원을 배타적으로 사용 하는 경우 Deadlock이l 발생 할 수 있다.

Deadlock은 2개의 Task가 서로 상대가 갖고 있는 자원 사용 권한을 기다리는 상태가 되어 두 Task 모두가 실행 될 수 없도록 되는 경우 이다.

Task A와 Task B가 모두 Mutex X와 Mutex Y 가 필요 한 경우

1) Task A 가 실행 중 Mutex X를 Take 한다.
2) Task A 가 Task B에 의하여 선점(Pre-empted) 된다.
3) Task B 가 Mutex Y를 Take 한다. 또한 Mutex X를 Take 하기 위한 시도를 한다. 그러나 Mutex X는 Task A가 Take 하고 있기 때문에 Task B 는 Blocked 상태가 되어 Mutex X가 Release 되기를 기다리고 있다.
4) Task A가 실행 되고 있는 상태에서 Mutex Y를 Take 하기 위한 시도를 한다. Mutex Y는 Task B가 Hold 하고 있기 때문에 Task

A는 Blocked 상태가 되어 Muntex Y가 가 Release 되기를 기다리고 있다.

위의 시나리오 결과 Task A는 Task B가 Hold 하고 있는 Mutex를 기다리고, Task B는 Task A가 Hold 하고 있는 Mutex를 Blocked 상태에서 기다리고 있기 때문에 두 task 모두 실행 될 수 없게 되는 Deadlock 이 발생 한다.

Priority Inversion과 마찬 가지로 시스템 설계 시 Deadlock를 피하기 위한 노력을 하여야 한다.

4. Gatekeeper Tasks

Priority Inversion과 Deadlock를 피하기 위하여 Gatekeeper Tasks를 사용 한다.

Gatekeeper Task만 자원에 대한 소유권(자원을 사용 할 수 있는 권한)을 갖는다. Gatekeeper Task 만이 유일 하게 자원을 직접 사용 할 수 있는 권한을 갖고, 다른 Task는 Gatekeeper 서비스를 이용 하여 간접적으로만 자원을 사용 할 수 있다.

F. Gatekeeper Task를 이용 하는 예

다음은 **RT_gatekeeper_printf** 프로그램에서 Gatekeeper Task를 이해 하는데 도움이 되는 부분 만 발췌한 내용 이다.

```
// Tasks 와 Interrupt이 Gatekeeper를 이용 하여 출력할 문자열을 정의 한다.
static char *pcStringsToPrint[] =
{
        "Task 1 ****************₩r₩n",
        "Task 2 ----------------₩r₩n",
        "Tick hook interrupt ###₩r₩n"
};

// xQueueHandle 형의 변수를 선언 한다.
// 이 Queue는 Print Task가 Gatekeeper Task를 이용 하여 문자열을 출력 하는데 이용 한다.
xQueueHandle xPrintQueue;

void vAltStartMutexPrintTasks( void ){

    // Queue는 사용 하기 전에 Create 되어야 한다.
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

        // prvPrintTask Task는 각각 Pseudo Random 하게 Delay 된다.
```

```
        // Pseudo random delay 를 발생 시키기 위한 Seed
        Number(random)를 설정 한다.
        srand( 567 );


        // Queue가 성공적으로 create 되었는 지 확인 한다.
        if( xPrintQueue != NULL )
        {
             // prvPrintTask의 Instance 2개(vPrint1Task, vPrint2Task)가
            생성된다.
            // 각 Instance에서 출력 할 문자열의 Index가 Task
            Parameter로 Pass 된다.
            // Task는 서로 다른 Priority로 생성 되어 낮은 Priority를 갖은
            Task도 문자열을 안전 하게 출력 하는 예를 보여 준다.
            xTaskCreate( prvPrintTask, "vPrint1Task",
                configMINIMAL_STACK_SIZE * 2 , ( void * ) 0 ,
                tskIDLE_PRIORITY + 1, NULL );
                xTaskCreate( prvPrintTask, "vPrint2Task",
                configMINIMAL_STACK_SIZE * 2 , ( void * ) 1 ,
                tskIDLE_PRIORITY + 2, NULL );
            // Gatekeeper Task의 우선 순위를 가장 낮게 설정 하고
            Task를 생성 한다. 이 Task 만 Standard Out Device를 Access
            할 수 있다.
            xTaskCreate(   prvStdioGatekeeperTask, "Gatekeeper",
                configMINIMAL_STACK_SIZE * 2 , NULL,
                tskIDLE_PRIORITY, NULL );
        }
}

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

        // 이 Task 만 오직 출력 장치에 직접 출력 할 수 있다.
        // 다른 Task는 출력 장치에 직접 출력 할 수 없기 때문에
        // 다른 Task는 이 Task를 이용 하여 출력 하여야 한다.
        // 그 결과 Gatekeeper Task를 이용 하는 경우에는 No Mutual
        // exclusion 나 Serialization 문제가 발생 하지 않는다.
        for( ;; )
        {
            // Queue에 Message 가 도착 하기를 기다린다.
            xQueueReceive( xPrintQueue, &pcMessageToPrint,
                portMAX_DELAY );
```

```
            // 메세지를 표준 출력 장치에 출력 한다.
            printf( "%s", pcMessageToPrint );
            fflush( stdout );
        }
}

// Tick hook(or tick callback) function의 사용 예
// Tick hook(or tick callback) function은 매 Tick interrupt 주기 매다
// Kernel에 의하여 Call 되기 때문에(우선 순위가 높은 Task에서 출력 예)
// 아주 간결 하게 작성 되어야 하고,
// FreeRTOS API 함수에서 Call 하여서는 않된다.
void vApplicationTickHook( void )
{
        static int iCount = 0;
        portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

        // 매 600 ticks 마다 Tick Hook Interrupt Routine에서 출력 하는
        // 메세지가 Gatekeeper Task를 이용 하여 출력 된다.
        iCount++;
        if( iCount >= 600 )
        {
            // 이 예에서  xHigherPriorityTaskWoken 은 사용 되지 않지만
            함수에 Parameter로 포함되어야 한다.
            // 매 600 Tick 마다 3번째 메세지를 출력 하기 위하여
            Queue의 앞(Front)에 메세지 포인터를 출력 한다.
            xQueueSendToFrontFromISR( xPrintQueue,
                &( pcStringsToPrint[ 2 ] ), &xHigherPriorityTaskWoken );

            // 600 ticks를 만들기 위하여 iCount를 Reset 한다.
            iCount = 0;
        }
}

// prvPrintTask Instance 2개(vPrint1Task, vPrint2Task)가 생성되어 실행
된다.
// Instance는 Instance에서 출력 하고자 하는 문자열의 Pointer를 인수로
전달 받는다.
void prvPrintTask( void *pvParameters )
{
    int iIndexToString;

    // 이 Task의 Instances 를 2개 Create 한다. Create 된 Task 가 실행
```

```
        되면 출력 할 메시지의 Index Pointer를 Queue를 이용 하여
        prvStdioGatekeeperTask에 Pass 하여 표준 출력 장치에 출력 한다.
        iIndexToString = ( int ) pvParameters;

        for( ;; )
          {
            // 이 Task에서는 문자열을 직접 출력 하지 않고, Queue를 이용
            하여 Gatekeeper Task에 출력 정보(문자열의 Pointer 등)를 전달
            하여 실제 출력은 Gatekeeper Task에서 실행 된다.
           xQueueSendToBack( xPrintQueue,
              &( pcStringsToPrint[ iIndexToString ] ), 0 );

              // 이 Task가 자원을 사용 한 후, 랜덤하게 발생된 지연 시간
              동안 Blocked 상태에 있도록 하여 낮은 우선도의 Task가
              자원을 사용 할 수 있는 기회를 준다.
              vTaskDelay( ( rand() & 0x03FF ) );
           }
}
```