EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv2 과정

제 1기

2022. 03. 12

손표훈

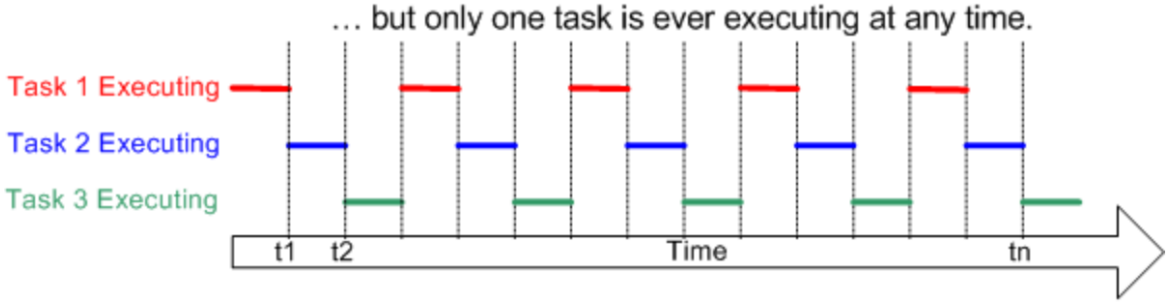
CONTENTS

EDDI
Electronic Design Development Institute

Multi-Tasking이란

EDDI
Electronic Design Development Institute

→ Multi-Tasking이란 여러 개의 태스크를 스케줄링 방식에 따라 실행시킨다
→ Multi-Tasking은 태스크의 수만큼 CPU의 수가 동일하지 않은 이상 동시에 여러 태스크를 "시간차 없이 동시에" 실행시키는 것은 아니다
사람이 반응할 수 있는 시간은 300ms로 300ms미만의 속도로 돌아가는 프로그램은 사람에게 마치 동시에 동작하는 것 으로 느껴진다
실제 multi-tasking은 하나의 태스크만 cpu자원을 사용 할 수 있기 때문에 정해진 스케줄링 시간(tick rate)에 따라 스케줄러가
태스크를 실행시키며, tick rate가 300ms미만이라면 동시에 동작하는 것 처럼 느껴진다



Multi-Tasking이란

EDDI
Electronic Design Development Institute

➢ Scheduler란?

→ Scheduler는 커널의 기능 중 하나로 스케줄링 방식에 따라 태스크의 상태를 보고 ready상태에 있는 태스크를 실행한다

→ 스케줄링 방식

(1) Preemptive : 제한된 스케줄링 시간에 의해 Task가 실행 중 스케줄러에 의해 다른 태스크로 전환(인터럽트를 허용한다)될 수 있다.
잦은 태스크 스위칭(context switching)으로 CPU자원이 소모된다.
시분할 방식, 우선순위 방식, 짧은 태스크 실행시간 우선순위 전환 방식이 있다.

(2) Non-preemptive : 먼저 실행된 태스크가 동작 종료 때 까지 CPU를 점유하고 있는 방식이다.
Task가 실행 중 다른 태스크로 전환될 수 없다(인터럽트를 허용하지 않는다). 태스크의 실행시간 보장된다
만약 task의 실행시간이 크면 다른 태스크들은 계속 기다리게 되며, 지연이 생긴다

→ freeRTOS는 두 가지 스케줄링 방식을 모두 지원한다. FreeRTOSConfig.h의 configUSE_PREEMPTION(1 : 선점형, 0 : 비선점형)
freeRTOS의 선점형 방식 중 태스크가 동일한 우선순위를 가지면 RoundRobin방식으로 설정된다.

TIME SLICE = 4

PROCESS	ARRIVAL TIME	BURST TIME	
		TOTAL	REMAINING
P1	0	8	8
P2	1	6	6
P3	3	3	3
P4	5	2	2
P5	6	4	4

READY QUEUE:

P1	P2	P3	P1	P4	P5	P2

GANTT CHART:

Preemptive scheduling(RoundRobin)

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	10
P2	3	5
P3	5	2
P4	6	6
P5	8	4

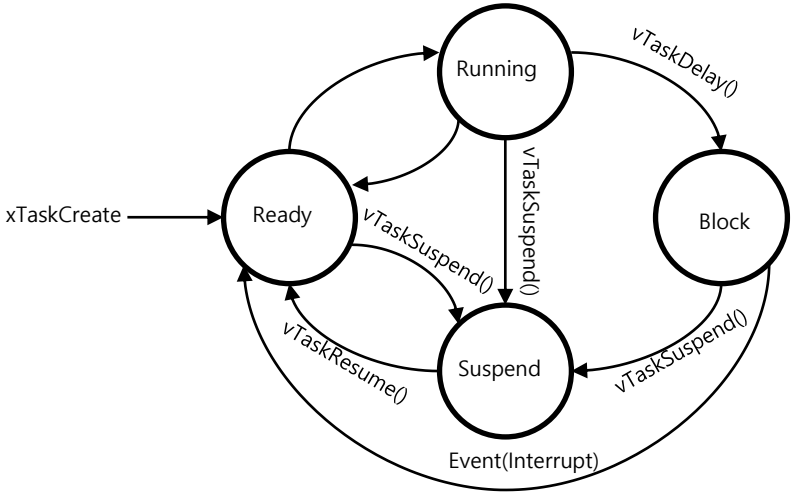
GANTT CHART

Non-preemptive scheduling

Multi-Tasking이란

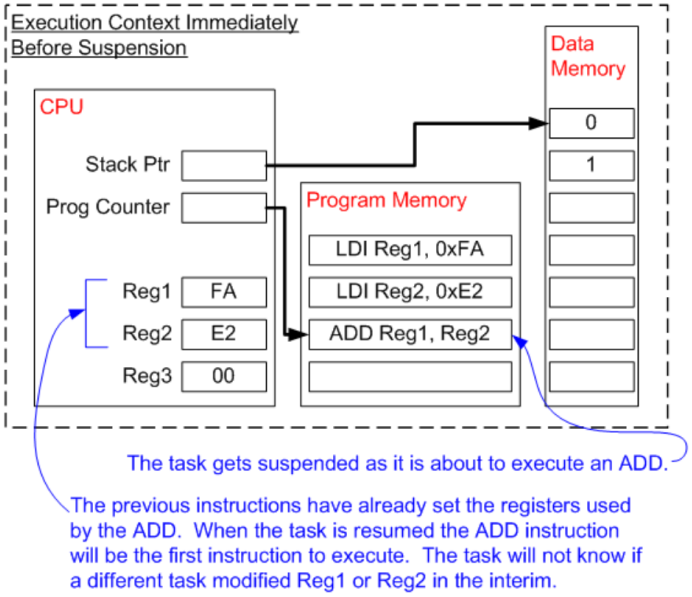
EDDI
Electronic Design Development Institute

- freeRTOS Task API에 따른 태스크 상태 전환



Context Switching이란

→ Context switching이란 task switching이라고도 하며 스케줄링 방식에 의해 태스크가 전환되는 것
→ Context switching이 발생하는 즉시 태스크의 TCB(Task Control Block)에 현재 레지스터 정보들이 저장된다



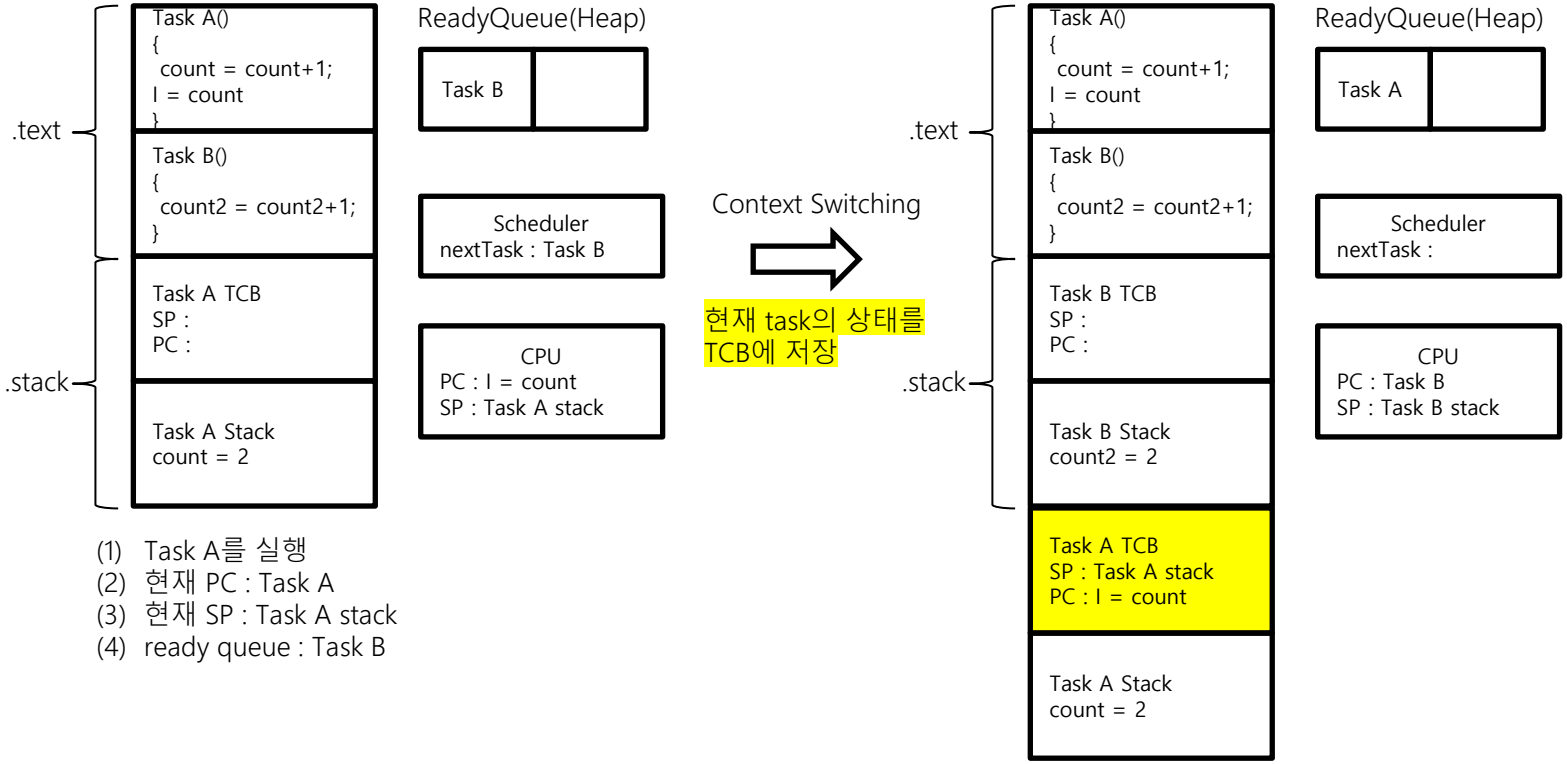
(1) Task1 : ADD 명령실행 중간 context switching이 발생
(2) Task2에 의해 Reg1, Reg2 값이 바뀜
(3) Task2를 종료하고 Task1을 재개 했을 때 바뀐 레지스터 값으로 ADD 명령을 실행
(4) 의도한 결과를 얻지 못한다

→ 위와 같은 상황을 방지하기 위해 커널은 중단 직전
현재 task의 context(레지스터, 스택 주소..)를 TCB에 저장해야 한다.

→ TCB에 저장되는 task의 주요 context는 다음과 같다
(1) task ID : 태스크 ID
(2) Stack Pointer : 현재 태스크의 스택 주소
(3) Program counter : 현재 PC 값
(4) task state : 현재 태스크의 상태(실행, 준비, 대기)
(5) register value : 프로세서 레지스터의 값

Context Switching이란

➢ Context Switching 처리 과정

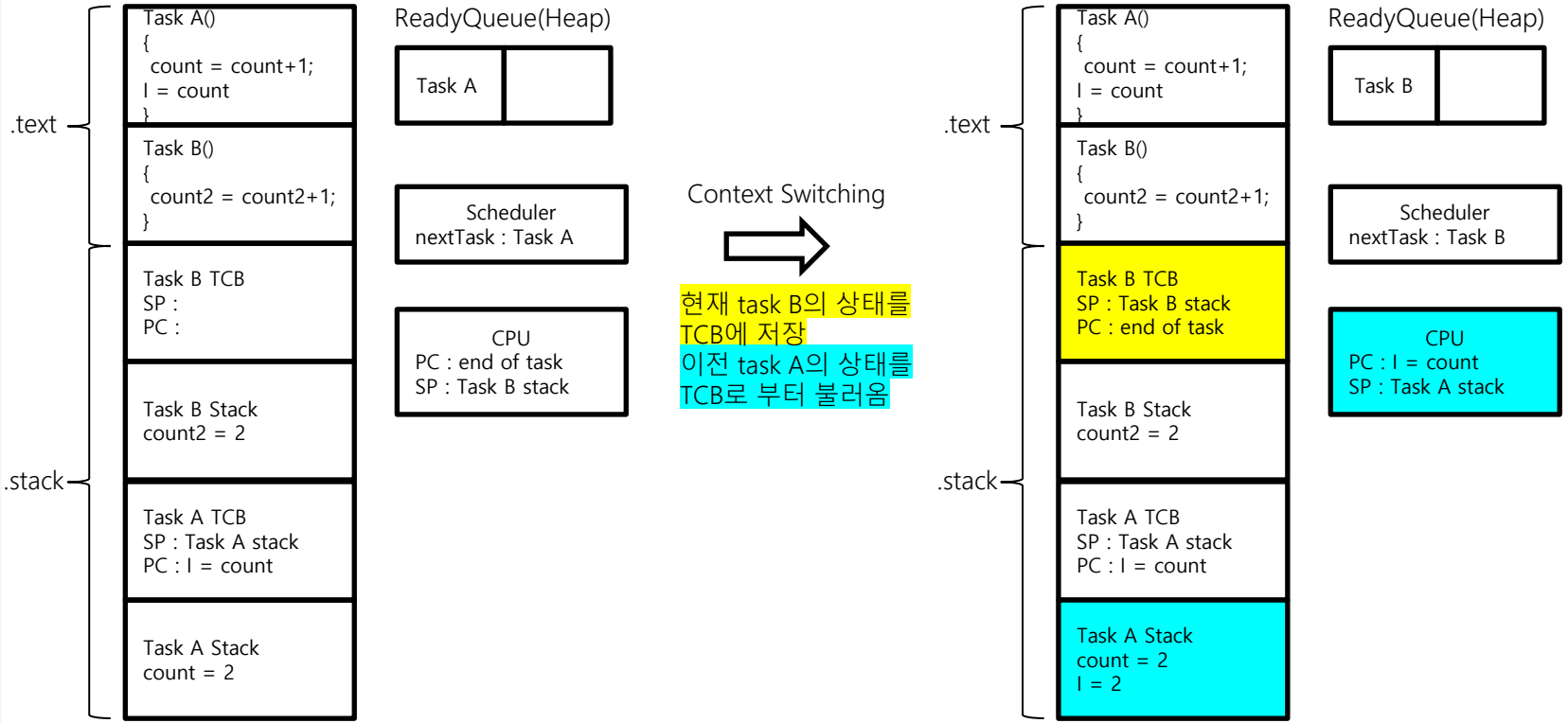


(1) Task A를 실행
(2) 현재 PC : Task A
(3) 현재 SP : Task A stack
(4) ready queue : Task B

Context Switching이란

EDDI
Electronic Design Development Institute

➢ Context Switching 처리 과정



Critical Section이란

EDDI
Electronic Design Development Institute

→ 서로 다른 두 Task가 자원을 공유하게 되면 해당 task들을 critical section이라 한다

→ Race Condition에 의한 오동작 문제(Race Condition은 하나의 공유자원에 여러 태스크들이 접근을 하게 되는 경우이다)

* 예시

```
unsigned count = 5;

void task1_produce(void)
{
  product[i] = produce();
  in = (in+1)%total_product;
------critical section------
  count++;
---------------------------
}
void task2_packing(void)
{
  packing(product[out]);
  out = (out+1)%total_product;
  ------critical section------
  count--;
---------------------------
}
```

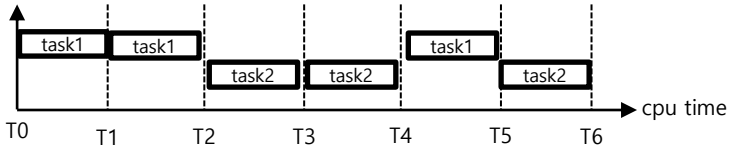
(1) 공장에서 제품을 생산하고 포장을 한다
(2) 생산태스크를 실행하고 포장태스크를 실행하여 포장을 진행한다

* task1_produce 코드의 기계어 동작은 다음과 같은 절차로 이루어진다
(1) register1 = count : 프로세서 레지스터에 메모리에 있는 count값을 로드 : T0
(2) register1 = register1+1 : 레지스터의 값을 증가연산 실행(decode과정도 있다) : T1
(3) account = register1 : 계산 완료된 register1의 값을 메모리에 있는 account로 저장 : T4

* task2_packing 코드의 기계어 동작은 다음과 같은 절차로 이루어진다
(1) register2 = count : 프로세서 레지스터에 메모리에 있는 count값을 로드 : T2
(2) register2 = register2-1 : 레지스터의 값을 감소연산 실행(decode과정도 있다) : T3
(3) price = register2 : 계산 완료된 register2의 값을 메모리에 있는 price로 저장 : T5

여기서 count에 대한 연산은 atomic operation이 아니다
- atomic operation이란? 명령어가 분리되지 않고, 실행되는 연산을 말한다
- count연산의 경우 읽기, 변경, 쓰기의 3단계 명령으로 연산이 진행된다
- 연산도중 다른 태스크에 의해 중단(race condition)될 수 있다
- 이 atomic operation이 보장되지 않으면 예시와 같이 mutual exclusion이 발생한다

→ 아래와 같이 task1 실행 중 count값을 메모리에 저장하기 직전 task2가 실행될 때 레지스터와 카운트 변수의 상황을 보면



T0 : register1 = count = 5
T1 : register1 = register1+1 = 6
T2 : register2 = count = 5
T3 : register2 = register2(5) – 1 = 4
T4 : count = register1 = 6
T5 : count = register2(0) = 4

→ 제품을 1개 생산하여 6개가 되었지만 카운트 값을 저장하기 전 task2가 실행되어 초기 카운트 값으로 포장 개수를 카운트하게 된다

→ 원하는 결과는 6개 제품 중 1개 포장으로 5였으나 4개가 된다

→ 위 코드의 count연산을 해주는 부분이 바로 critical section이 된다

Critical Section이란

→ 서로 다른 두 Task가 자원을 공유하게 되는 critical section의 문제를 해결하기 위해 현재 실행중인 태스크 외에 다른 태스크가 공유자원에 접근하지 못하도록 막는다. 이 때 사용되는 것이 semaphore, mutex이다

→ critical section문제의 고전적인 SW처리 방법은 Peterson's Solution이다.

Task A

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Task B

```
do {
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);

        critical section

    flag[j] = FALSE;

        remainder section

} while (TRUE);
```

(1) Task A 실행 시 Task A에 대한 Flag[i]를 set한다
(2) Task A에서 Task B로 전환 되면 Task B에서 flag[j]를 set하고, turn변수에 i를 저장한다.
(3) flag[i]와 turn이 i이므로 while 무한루프를 실행하게 되며 critical section을 실행하지 못한다
(4) 이 때 다시 Task A로 복귀했을 때 flag[j]가 set이 되었어도 Task B에서 turn을 i로 변경했기 때문에 while문을 실행하지 않고 critical section에 진입한다.
(5) critical section 실행을 마치고 flag[i]를 reset한다
(6) Task B는 critical section에 진입할 수 있는 조건이 된다

Critical Section이란

EDDI
Electronic Design Development Institute

➢ Mutex vs Semaphore?

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

```
signal(S) {
    S++;
}
```

→ 세마포어는 변수로 n개의 태스크가 n개의 공유자원을 사용하는 것을 동기화 한다

→ 세마포어는 signal, wait 매커니즘을 사용한다

→ 세마포어에는 두 가지 방법이 있다. 하나는 counting semaphore, 다른 하나는 binary semaphore이다
binary semaphore는 1,0 두 가지 상태로 한번에 하나의 태스크만 공유자원에 접근할 수 있게 한다.

→ counting semaphore는 공유자원 사용에 대한 태스크 수를 count변수로 제한한다
여기서 공유자원인 count변수는 atomic operation으로 연산 된다.
(race condition의 문제를 방지하는 목적인 변수가 race condition문제가 발생되면 안되므로..)

Acquire:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

Release:

```
release() {
   available = true;
}
```

→ Mutex는 한번에 1개의 태스크만이 공유자원에 접근할 수 있도록 한다.

→ 상태는 1, 0이며 이진 세마포어(binary semaphore)라고도 한다. 세마포어와 차이점은 우선순위 상속이다
우선순위의 상태가 뮤택스를 가지고 있는 태스크1 < 대기중인 태스크2라면, 태스크2가 우선순위가 높아 실행 빈도가 잦아 태스크1이 뮤택스 give를 하지 못하는 상황이 발생하여 프로세서가 대기상태에 들어간다. 이를 해결하기 위해 태스크1의 우선순위를 critical section 진입 시 태스크2와 동일하게 변경하고 give시 원래 우선순위로 돌려 놓는다.

→ 뮤텍스는 take(critical section진입 시), give(critical section 종료 시)인 locking 매커니즘을 사용하여
태스크간 공유자원에 대해 상호배제를 보장하여 동기화를 보장한다

→ spin lock 문제 : 먼저 mutex를 take한 태스크에 의해 다른 태스크가 while루프에 의해 block상태(spin lock)가 되며
mutex를 take한 태스크가 give를 할 때 까지 유지된다. 이 block상태는 cpu cycle을 낭비한다

→ 대신 이 spin lock방식으로 인해 context switching의 overhead(태스크가 사용하는 메모리, 실행시간)가 발생하지 않는다
mutex를 take하지 못한 태스크들은 while루프에 빠져 다른 동작을 못하기 때문이다

Critical Section이란

EDDI
Electronic Design Development Institute

➢ Mutex vs Semaphore?

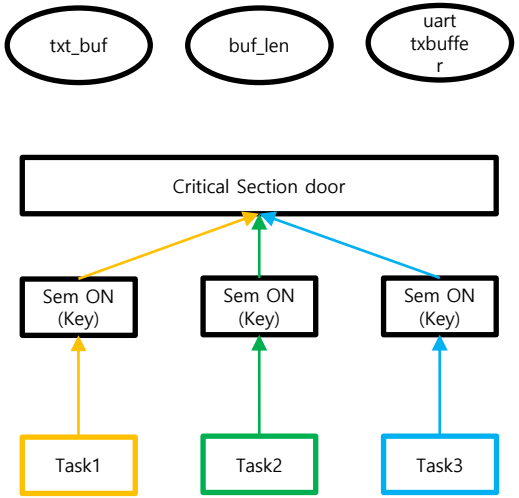
→ 카운팅 세마포어의 예

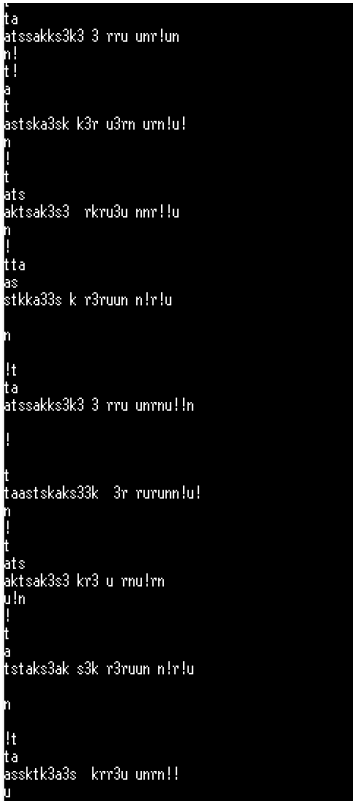
```
//task
//sem = xSemaphoreCreateBinary();
sem = xSemaphoreCreateCounting(3,0);

xSemaphoreGive(sem);
xSemaphoreGive(sem);
xSemaphoreGive(sem);
 void run_task3(void *pvParameters)
{
    char i = 0;
    for(;;)
    {
        //sprintf(txt_buf, "Enter task3!\n\r\0");
        //buf_len = strlen(txt_buf);
        //sci_display_text(UART, (uint8 *)txt_buf, buf_len);

        if(xSemaphoreTake(sem, (TickType_t)0x01) == pdTRUE)
        {
            sprintf(txt_buf, "Task3!\n\r\0");
            buf_len = strlen(txt_buf);
            sci_display_text(UART, (uint8 *)txt_buf, buf_len);
            xSemaphoreGive(sem);

            vTaskDelay(1000);
        }
    }
}
```





→ 위 와 같은 태스크가 3개 있고, 카운트 값을 3으로 설정했다
→ 세마포어가 3개가 생성되었으므로 각각의 태스크는 임계영역에 들어간다
→ 문제는 각각의 태스크가 공유자원에 접근 가능하게 되면서 race condition 해결이 안된다
→ 바이너리 세마포어, 뮤텍스와 달리 문제가 발생한다
→ 뮤텍스나 바이너리 세마포어는 공유자원에 1개의 태스크만 접근 가능하도록 제한하기 때문에 문제가 발생하지 않는다

Critical Section이란

EDDI
Electronic Design Development Institute

➢ Mutex vs Semaphore?

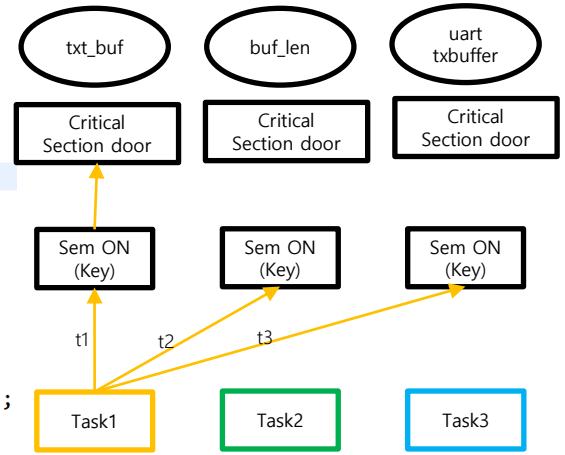
→ 카운팅 세마포어의 예

```
void run_task1(void *pvParameters)
{
    for(;;)
    {
        xSemaphoreTake(sem, (TickType_t )0x01);
        sprintf(txt_buf, "Task1!\n\r\0");


        xSemaphoreTake(sem, (TickType_t )0x01);
        buf_len = strlen(txt_buf);


        xSemaphoreTake(sem, (TickType_t )0x01);
        sci_display_text(UART, (uint8*) txt_buf, buf_len);


            vTaskDelay(4001);
    }
}
```



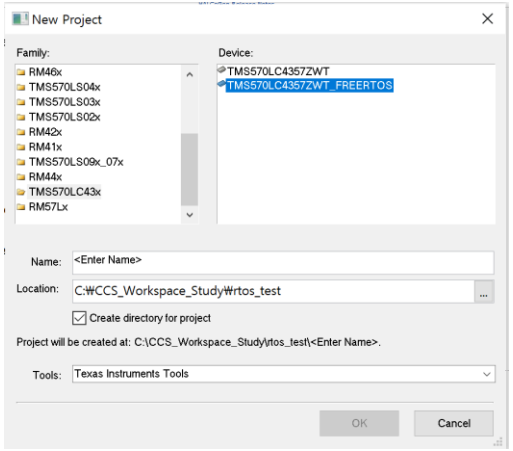
→ 위 와 같이 코드를 변경하여 각각의 공유자원에 세마포어 take, give를 해준다
→ 예를 들어 t1, t2, t3시간에 세마포어 3개를 task1이 다 가져가면 Task2, Task3는 임계영역에 들어가지 못한다
→ task1이 공유자원을 다 사용한 뒤 세마포어를 give하면 다음 Task2, Task3가 공유자원을 사용 할 수 있게 된다

freeRTOS 사용법

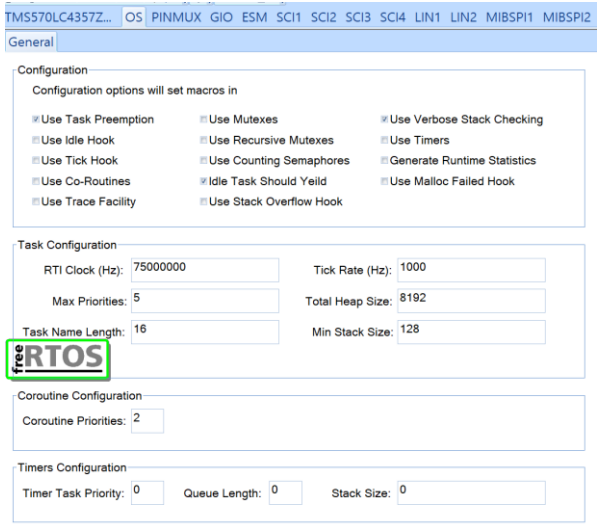
EDDI
Electronic Design Development Institute

➢ Halcogen 설정

#1 : Device 선택에서 freeRTOS가 붙은 것으로 선택한다



#2 : OS 탭에서 freeRTOS를 포팅을 위해 원하는 설정을 선택할 수 있다



freeRTOS 사용법

EDDI
Electronic Design Development Institute

➢ Halcogen 설정 및 SW 설정

순번	내용		헤더파일명	소스파일
1	세마포어 사용을 위한SemaphoreHandle_t 변수 선언		os_semphr.h	
2	Task관련 자료구조 사용을 위한 TaskHandle_t 변수 선언(void형 포인터)		os_task.h	
3	Task 사용자함수 정의		os_semphr.h	os_semphr.c
4	SemaphoreHandle_t변수에 세마포어 생성	xSemaphoreCreateBinary()		
		xSemaphoreCreateCounting(세마포어 수, 초기값)		
5	세마포어 초기 활성화, 태스크가 critical section에 접근할 권한 활성화	xSemaphoreGive()		
6	태스크 생성	xTaskCreate(태스크 함수, "함수 이름", 태스크 스택 사이즈, , 태스크 우선순위, &태스크 핸들러 변수)	os_task.h os_mpu_wrappers.h	os_task.c
7	다른 태스크가 공유자원에 접근하는 것을 막기위해 태스크 함수내 critical section 진입 시 세마포어 설정	xSemaphoreTake(세마포어 핸들러변수, 태스크 block 수행시간)	os_semphr.h	os_semphr.c

freeRTOS 사용법

EDDI
Electronic Design Development Institute

➢ Halcogen 설정 및 SW 설정

순번	내용		헤더파일명	소스파일
1	세마포어 사용을 위한SemaphoreHandle_t 변수 선언		os_semphr.h	
2	Task관련 자료구조 사용을 위한 TaskHandle_t 변수 선언(void형 포인터)		os_task.h	
3	Task 사용자함수 정의		os_semphr.h	os_semphr.c
4	SemaphoreHandle_t변수에 세마포어 생성	xSemaphoreCreateBinary()		
		xSemaphoreCreateCounting(세마포어 수, 초기값)		
5	세마포어 초기 활성화, 태스크가 critical section에 접근할 권한 활성화	xSemaphoreGive()		
6	태스크 생성	xTaskCreate(태스크 함수, "함수 이름", 태스크 스택 사이즈, , 태스크 우선순위, &태스크 핸들러 변수)	os_task.h os_mpu_wrappers.h	os_task.c
7	다른 태스크가 공유자원에 접근하는 것을 막기위해 태스크 함수내 critical section 진입 시 세마포어 설정	xSemaphoreTake(세마포어 핸들러변수, 태스크 block 수행시간)	os_semphr.h	os_semphr.c