uC/OS-Ⅱ 뛰어넘기(Ⅲ)

Task 생성 및 제어

이전 회까지 우리는 RTOS와 uC/OS에 대한 일반적인 내용들을 배웠다. 이번 회에서는 코드 중심으로 Task를 생성하고 제어하는 방법을 살펴보고, 이 예제들을 통해서 Multi-Tasking 프로그램의 동작방식을 공부하고 Preemptive kernel에 대한 개념을 이해해 보도록 하자.

글: 김대홍/CyberLab 실장, 삼성 첨단기술연수소 RTOS강사
redizi@armkorea.com

Task 만들기

리스트 1은 3개의 Task가 동작하는 예제이다.

```
#include "includes.h"

#define TASK_STK_SIZE              512
#define  N_TASKS                    2

OS_STK TaskStk[N_TASKS][TASK_STK_SIZE];    ----- (1)
OS_STK TaskStartStk[TASK_STK_SIZE];        ----- (2)

void   Task1(void *data);
void   Task2(void *data);
void   TaskStart(void *data);

void main (void)
{
        PC_DispClrScr(DISP_FGND_WHITE
+DISP_BGND_BLACK);
        OSInit();
        PC_DOSSaveReturn();
        PC_VectSet(uCOS, OSCtxSw);
        OSTaskCreate(TaskStart, (void *)0, (void
*)&TaskStartStk[TASK_STK_SIZE - 1], 0);
        OSStart();
}

void TaskStart (void *data)
{
        UBYTE  i;
        char   s[100];
        WORD   key;

        data = data

        OS_ENTER_CRITICAL();
        PC_VectSet(0x08, OSTickISR);
        PC_SetTickRate(OS_TICKS_PER_SEC);
        OS_EXIT_CRITICAL();
```

```
        OSTaskCreate(Task1, (void *)&TaskData[0], (void
*)&TaskStk[0][TASK_STK_SIZE - 1], 1);      -- (3)
        OSTaskCreate(Task2, (void *)&TaskData[1], (void
*)&TaskStk[1][TASK_STK_SIZE - 1], 2);                -- (4)

        for (;;)
        {
                if (PC_GetKey(&key))
                {
                        if (key == 0x1B)
                        {
                          PC_DOSReturn();
                        }
                }
                OSTimeDlyHMSM(0, 0, 1, 0)    --- (5)
        }
}

void Task1(void *data)
{
        for (;;)
        {
                printf("Task1₩n");
                OSTimeDly(100);                --- (6)
        }
}

void Task2(void *data)
{
        for (;;)
        {
                printf("Task2₩n");
                OSTimeDly(200);                --- (7)
        }
}
```

리스트 1. Task가 동작하는 예제

main 함수와 StartTask의 앞쪽 코드까지는 이미 지난 시간에 설명을 했기 때문에 이 부분은 더 이상 설명을 하지 않도록 하겠다.

Task는 일반적으로 아래와 같은 특징을 갖고 있다.

- 문제의 한 부분을 책임지는 독립적 수행 단위
- Scheduling과 자원 할당의 단위
- 각 Task는 서로에게 독립적 존재
- Task간의 의사소통은 변수나 Event를 사용
- Priority, Stack을 소유
- 무한 루프로 구성
- 함수 형태로 존재

```
void Task(void * para)
{
        while(1)
        {
        }
}
```

〈기본적인 Task의 형태〉

보기에는 C의 일반 함수로 보이지만, 자세히 보면 약간 다른 점을 발견할 수 있다. 즉, 함수의 내부가 무한 루프로 구성이 된다는 것과 이것을 호출하는 곳이 존재하지 않는다는 것이다.

하지만, 리스트 1을 실행시키면 아래와 같은 결과가 나온다.

```
Task1
Task1
Task2
Task1
Task1
Task2
Task1
Task1
Task2
```

〈리스트 1의 출력 결과〉

이 결과로 uC/OS 커널에 의해 두 개의 함수가 Task로써 동작을 하고 있다는 것을 알 수 있게 된다.

[함수원형]
INT8U OSTaskCreate(void (*task)(void*pd), void *pdata, OSSTK *ptos, INT8U prio)

[인자]
task – Task로 만들기 위한 함수의 시작주소
pdata – Task에게 넘길 인자
ptos – Task가 사용할 스택의 시작주소
prio – Task의 우선순위

OSTaskCreate() 함수는 주어진 함수를 Task로 만들고 수행을 시작하게 만드는 함수이다.

이 함수는 주어진 코드와 스택을 연결하여 독립적인 실행 개체로 만들고 우선순위에 따라서 이 Task들을 제어하게 된다(그림 1 참조).

uC/OS에서는 Task가 사용하는 스택은 유저가 직접 할당받아 공급해주어야 한다. 그래서 리스트 1의 (1)과 (2)는 task들을 위한 스택 영역을 전역 배열로 할당을 받는 과정이다. 그리고 일반적으로 스택은 상위번지에서 하위번지로 자라게 되므로 스택의 시작번지를 할당받은 배열의 맨 마지막 주소를 알려주게 된다.

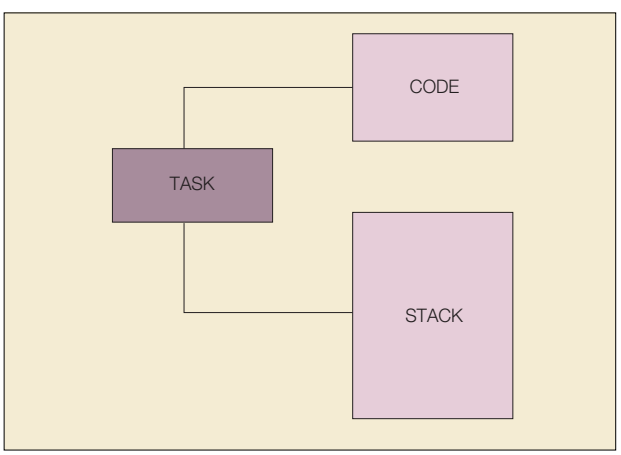
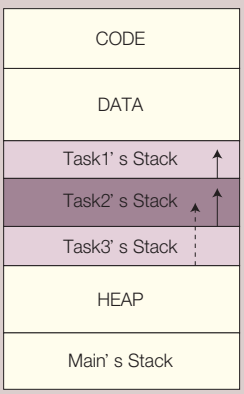


그림 1. Task의 생성

* 스택 생성 시 주의사항
만약 Task가 할당된 스택보다 더 많은 공간을 쓰게 되면 다른 Task의 스택이나 프로그램의 공간을 침범하게 되어 치명적인 손상을 입어 OS가 정상동작을 할 수 없게 된다. 그러므로 처음사용 시 충분한 공간을 주도록 하자.



〈Task2의 Stack이 깨지는 경우〉

uC/OS에서는 우선순위 값이 작을수록 우선순위가 높아진다. 그러므로 리스트 2의 코드에서 TaskStart 〉 Task1 〉 Task2 순의 우선순위를 갖게 된다.

Task 상태

그림 2의 Task 상태의 정의와 상태의 변화를 이해하는 것은 Multitasking 환경을 이해하는데 있어서 아주 중요한 요소가 된다(화살표 방향을 잘 이해하는 것이 중요하다).

그림 3은 그림 2의 상태에서 좀더 더 범위를 넓혀서 살펴본 것이다.

• Ready
현재 수행할 수 있는 준비는 되어있지만, 제어권을 갖지 못해 대기 중인 상태.

• Running
코드를 수행중인 상태로 단 1개의 Task만이 이 상태에 있을 수 있다. 그리고 running 상태의 Task가 반드시 존재해야 한다.

• Wait

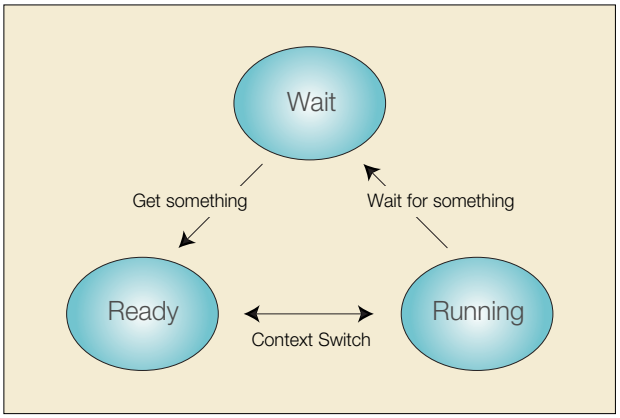


그림 2. 동작중인 Task의 상태

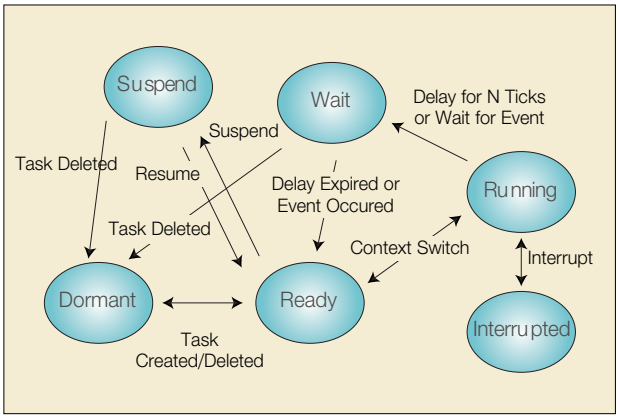


그림 3. 확장된 Task의 상태

어떠한 조건을 기다리고 있는 상태로, 이 상태에 있는 Task는 scheduling의 대상이 아니다.

• Dormant

Task로 생성되지 못하고 코드만 존재하는 상태이다. 리스트 1의 (3)과 (4)가 수행되기 전까지 Task1과 Task2는 이 상태에 있다고 말할 수 있다.

• Suspend

그림 2의 상태는 Task가 일할 때의 상태라고 한다면 Suspend 상태의 Task는 휴가중이라고 생각을 하면 된다. 그러므로 이 상태의 Task는 scheduling의 대상에서 제외된다.

• Interrupt

인터럽트 서비스 루틴이 수행중인 상태를 말한다. 이 상태는 일반 Task와는 분리된 개념으로, 이 상태와 Task가 연동하게 되면 device driver와 같은 동작을 수행할 수 있다.

이 내용을 정리하면 Ready와 Running 상태에 있는 Task만이 scheduling의 대상이 되며, Wait 상태에 있는 Task가 scheduling의 대상이 되려면 반드시 Ready 상태로 와야 한다고 정리할 수 있다.

☞ scheduling에 의해 다음에 수행할 Task를 결정하게 된다.

Task 제어

위의 내용을 기억하고 아래의 함수를 보도록 하자.

[함수원형]
void OSTimeDly(INT16U ticks)

[인자]
ticks – Task가 주어진 tick만큼 쉰다.

[함수원형]
INT8U OSTimeDlyHMSM(INT8U hr, INT8U min, INT8U sec, INT8U ms)

[인자]
hr – Task가 주어진 시간만큼 쉰다.
min – Task가 주어진 분만큼 쉰다.
sec – Task가 주어진 초만큼 쉰다.
ms – Task가 주어진 1/1000초만큼 쉰다.

이 두 함수는 Task를 Wait 상태가 되게 하는 대표적인 함수이다. Task가 이 함수들을 호출하게 되면 호출한 Task는 Wait 상태에 들어간 후 함수에 주어진 시간이 지난 후에 자동적으로 Ready 상태가 된다.

리스트 2에서 (3)과 (4)의 코드를 수행하면 Task1과 Task2가 생성되지만, TaskStart가 우선순위가 가장 높으므로, 이 두 개의 Task들은 ready 상태를 유지하고 TaskStart가 계속 수행되게 된다.

TaskStart가 (5)를 수행하게 되면, 이 Task는 Wait 상태에 들어가게 되고 scheduler는 ready 상태에 있는 Task1과

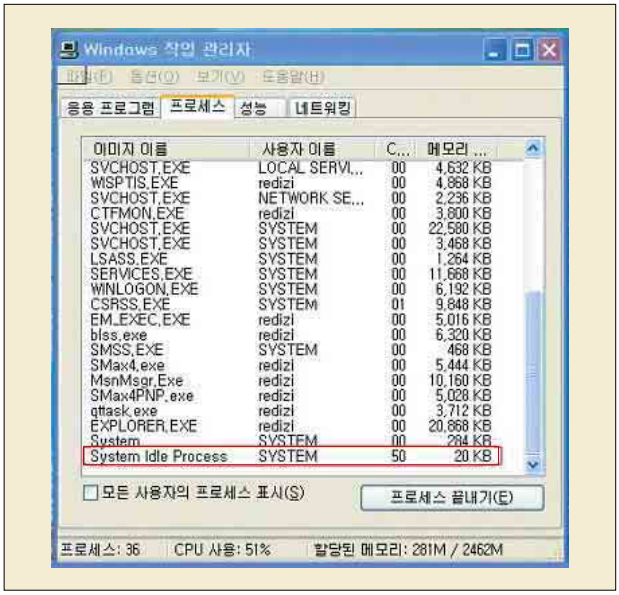


그림 4. Windows의 Idle Process

Task2를 가지고 scheduling을 하게 된다. Task1이 Task2에 비해서 우선순위가 높으므로 수행권은 Task1에게 넘어간다.

☞ Task의 상태가 변하게 되면 scheduler가 호출된다.
☞ Tick 발생시 scheduler가 호출된다.

Task1이 수행을 하다가 (6)을 수행하게 되면 Task1도 wait 상태에 들어가게 된다. 그러면 ready 상태에 있는 Task는 Task2 밖에 없으므로 이제 수행권이 Task2에게 넘어가게 된다. Task2도 마찬가지로 (7)의 코드를 수행하는 순간 wait 상태로 들어가게 된다.

여기서 질문!!

현재 모든 Task들은 wait 상태에 있다. 그런데 앞의 "Task 상태"에서 언급하기를 Running 상태에 있는 Task가 반드시 존재해야 한다고 했는데, 그렇다면 이 상황은 어떻게 된 것일까?

사실, 이때 수행되는 Task가 있다. 이 Task는 Idle task라고 불리는데, 이것은 OS가 내부적으로 만들게 되고 프로그래머의 모든 Task들이 수행권이 없을 때, 이 Task가 수행되게 된다.

참고로 Windows에서도 이러한 종류의 process를 확인할 수 있다. Ctrl+Alt+Del키를 눌러서 작업관리자를 불러보자. 그림 4와 같은 창을 확인할 수 있다.

* Scheduler
수행할 Task를 결정하는 루틴으로 사람의 머리에 해당된다. 일반적으로 이 과정은 Task의 우선순위를 가지고 결정하게 되는데, OS에 따라서 우선순위번호가 낮을수록 우선순위가 높아지거나 우선순위번호가 높을수록 우선순위가 높아지게 된다. uC/OS의 경우 전자에 해당된다.

* TICK
OS가 사용하는 시간단위이다.
Tick은 대개 1초당 50~200tick 정도로 설정하는데, uC/OS에서는 OS_CFG.H 파일에서 이것을 설정할 수 있게 되어있다.
예) 200tick/초로 설정하면 1tick = 5ms가 된다.

Tick의 발생은 하드웨어적인 Timer를 사용하여 발생시키게 되는데, 이때 해당 ISR(Interrupt Service Routine)에서는 scheduler가 같이 호출되게 된다. 그러므로 1초당 발생하는 Tick 수를 크게 설정하면 OS가 관리할 수 있는 시간의 분해능(시간을 정밀하게 사용)이 높아지지만 그만큼 scheduler가 과도하게 호출되어 연산에 부담이 되고, 너무 작게 설정하면 분해능이 떨어져 충분한 제어효과를 가져오기 힘들게 된다.

OSTimeDlyHMSM() 함수는 OSTimeDly()와 기능은 동일하지만 사람이 사용하는 일반적인 시간 개념으로 값을 입력할 수 있어서 편리하게 사용이 가능하다.

이번에는 리스트 2의 Task가 수행되는 경우를 생각해보자.

```
void Task1(void *data)
{
    for (;;)
    {
        printf("Task1₩n");
        OSTimeDly(100);
    }
}

void Task2(void *data)
{
    for (;;)
    {
    }
}
```

리스트 2

리스트 1과 달라진 점은 Task2 안에 있던 코드들이 없어져서 Task2가 수행을 하게 되면 이전과는 다르게 계속 수행이 되므로 절대로 wait 상태로 들어가지 않는다. 그러다가 Task1이 wait 상태로 들어간 후로 100tick이 지나게 되면, Task1은 ready 상태로 오게 된다. 이때 scheduler가 호출이 되고, Task1의 우선순위가 높으므로 Task2는 강제로 제어권을 빼앗겨 running 상태에서 ready 상태로 오게 되고, Task1은 다시 running 상태가 되어 다시 실행을 하게 된다.

이러한 상태를 Task1이 Task2를 선점(Preempt)했다고 표현한다.

☞ RTOS는 선점형 커널을 갖고 있어 우선순위가 높은 task가 하위 task를 누르고 항상 수행하게 된다. 이러한 속성으로 인해 RTOS는 "우선순위가 높은 Task가 실시간적인 응답"을 보이게 되는 것이다.

```
void Task1(void *data)
{
        for (;;)
        {
        }
}

void Task2(void *data)
{
        for (;;)
        {
                printf("Task2₩n");
                OSTimeDly(100);
        }
}
```

리스트 3

그렇다면 리스트 3의 결과는 어떻게 될까? 이 경우는 Task2가 절대로 수행을 할 수 없게 된다. Task2는 Task1보다 우선순위가 낮으므로 Task1이 wait 상태나 suspend 상태에 들어가야 제어권을 얻게 되는데, 현재의 상태에서는 Task1이 항상 수행을 하게 되므로 Task2가 수행을 할 수 있는 기회가 없어지게 된다.

이러한 상태를 기근(Starvation)현상이라고 표현한다. 그러므로 상위 Task는 하위 Task가 수행할 수 있도록 배려를 해주어야만 한다.

리스트 4는 키보드로 's'를 누르면 Task1을 suspend시키고 'r'을 누르면 resume을 시키는 예제이다.

[함수원형]
INT8U OSTaskSuspend(INT8U prio)

[인자]
prio - 주어진 우선순위의 task를 suspend 상태로 만든다.

[함수원형]
INT8U OSTaskResume(INT8U prio)

[인자]
prio - 주어진 우선순위의 task의 suspend 상태를 해제한다.

```
void TaskStart (void *data)
{
        UBYTE  i;
        char   s[100];
        WORD   key;

        data = data

        OS_ENTER_CRITICAL();
        PC_VectSet(0x08, OSTickISR);
        PC_SetTickRate(OS_TICKS_PER_SEC);
        OS_EXIT_CRITICAL();

        OSTaskCreate(Task1, (void *)&TaskData[0],
(void *)&TaskStk[0][TASK_STK_SIZE - 1], 1);
        OSTaskCreate(Task2, (void *)&TaskData[1],
(void *)&TaskStk[1][TASK_STK_SIZE - 1], 2);

        for (;;)
        {
                if (PC_GetKey(&key))
                {
                        if (key == 0x1B)
```

리스트 4〈계속〉

```
                        {

                            PC_DOSReturn();
                        }
                        if( key == 's' ) OSTask
Suspend(1);
                        if( key == 'r' ) OSTask
Resume(1);
                }
                OSTimeDlyHMSM(0, 0, 1, 0);
        }
}

void Task1(void *data)
{
        for (;;)
        {
                printf("Task1₩n");
                OSTimeDly(200);
        }
}

void Task2(void *data)
{
        for (;;)
        {
                printf("Task2₩n");
                OSTimeDly(200);
        }
}
```

리스트 4

Task1이 suspend되면 이 task는 resume이 될 때까지 scheduling의 대상에서 제외가 되어 그 자리에서 실행을 멈추게 된다.

여기까지 해서 기본적인 Task의 생성과 동작원리, 제어방법까지 살펴봤다.

다음 회에는 Task간의 통신 및 동기화에 대해서 알아보도록 하자. RTime