

## 설계와 구현을 통한 임베디드 OS의 이해와 응용 ⑦

## 문맥 전환 및 스케줄링 루틴 작성



이번 호에서는 multitasking을 수행하기 위해 반드시 필요한 문맥 전환 루틴과 스케줄링 루틴을 작성해 보고, 이 루틴들을 시스템 콜 처리 루틴과 하드웨어 인터럽트 처리 루틴에 추가해 보자.

글 : 서민우 / 과학기술 정보연구소([www.stii.co.kr](http://www.stii.co.kr))

[minucy@hanmail.net](mailto:minucy@hanmail.net)

OS가 펌웨어와 다른 특징 중에 하나는 multitasking을 수행한다는 점이다. 즉, 펌웨어는 하나의 태스크를 수행하는 반면에, OS는 여러 태스크를 관리한다. 이러한 multitasking을 수행하기 위해 필수적인 태스크간 문맥 전환 루틴을 시스템 콜 루틴과 하드웨어 인터럽트 루틴에 추가하고 시험하여 그 원리를 정확히 이해해야 한다.

태스크간 전환을 수행하는 시점에서 새로운 태스크를 선택하는 동작을 태스크 스케줄링이라 한다. 태스크 스케줄링을 어떻게 설계하고 구현하느냐에 따라 우선순위 기반의 스케줄링, 라운드 로빈 방식의 스케줄링 등이 결정된다. 여기서는 간단한 형태의 라운드 로빈 방식의 스케줄링 루틴을 작성해 볼 것이다.

OS 내에서는 비동기적으로 발생하는 하드웨어 인터럽트와 하드웨어 인터럽트 루틴 내에서의 태스크 스케줄링으로 인해 하드웨어 인터럽트 처리 루틴과 태스크 루틴간 또는 태스크 루틴과 태스크 루틴간 경쟁 상태가 발생할 수 있다. 이러한 경쟁 상태는 공유 영역을 중심으로 발생하게 된다. 그러면

먼저 문맥 전환 루틴과 스케줄링 루틴을 작성해보자.

다음은 main.c 파일의 내용이다.

```
main.c
...
void kmain(void)
{
    uart_init();
    timer_init();
    mmu_fill_l2pt(TASK0L2PT, TASK0L2PT+8,
TASK0BASE);
    mmu_fill_l2pt(TASK1L2PT, TASK1L2PT+8,
TASK1BASE);
    flush_cache_tlb();
    mmu_map_l2pt(MASTERL1PT+4, TASK0L2PTBASE);
```

```
init_task0;
uart_puts("jump to task0\n");
jump2task00;
}
```

kmain 함수에는 이전 기사의 예제에 init\_task 함수를 호출하는 부분이 추가되었다. init\_task 함수는 문맥 전환을 수행하기 위해서 필요한 전역 변수와 태스크를 관리하기 위한 전역 변수를 초기화하는 역할을 하며, 이번 예제에 새롭게 추가된 task.c 파일에 정의되어 있다.

다음은 task.c 파일의 내용이다.

```
task.c
typedef struct {
    unsigned int context[18];
} process_state;

process_state process[2];
process_state * current;
process_state * prev, * next;
unsigned int pid = 0;

void init_task()
{
    current = &process[0];
    next = current;
    process[1].context[13+2] = 0x408000;
    process[1].context[1] = 0x400000;
    process[1].context[0] = 0x50;
}

#define MASTERL1PT (unsigned int *) (0x100000)
```

```
#define TASK0L2PTBASE (0x30004400|0x03<<5|1<<4|0x1)
#define TASK1L2PTBASE (0x30004800|0x03<<5|1<<4|0x1)

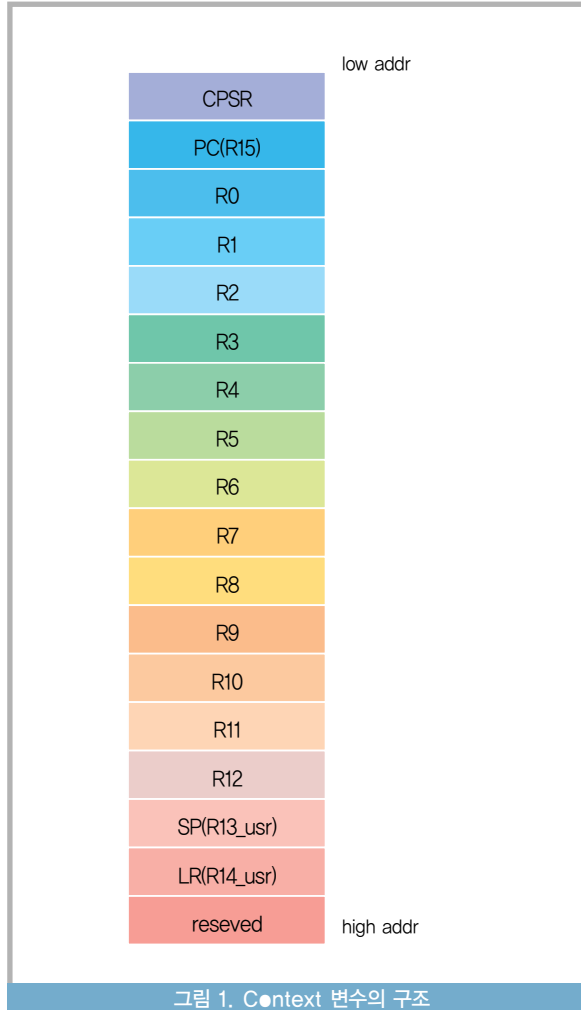
void schedule()
{
    pid = (pid+1)%2;
    current = &process[pid];
    next = current;

    flush_cache_tlb();
    if(pid == 0) mmu_map_l2pt(MASTERL1PT+4, TASK0L2PTBASE);
    else mmu_map_l2pt(MASTERL1PT+4, TASK1L2PTBASE);
}
```

process\_state는 하나의 태스크를 관리하기 위한 구조체이며, 여기서는 어느 순간 태스크의 상태를 저장하기 위한 context 배열 변수만 두었다. context 배열 변수는 태스크를 수행하던 CPU의 어느 순간 레지스터의 상태를 저장하기 위한 공간이다. 현재 우리가 사용하는 CPU는 ARM 코어를 내장하고 있다. 따라서 context 배열 변수에 그림 1과 같이 CPU의 레지스터 세트를 저장할 것이다. 마지막 엔트리는 사용하지 않는 공간이다.

process 배열 변수는 두 개의 태스크(task0, task1)를 관리하기 위하여 선언했으며, process[0], process[1] 변수로 각각 task0, task1 태스크를 관리한다. current 포인터 변수는 현재 수행 중인 태스크를 가리키는 역할을 한다. prev, next 포인터 변수는 태스크간 문맥 전환을 수행할 때 사용하기 위해 선언했으며, pid 변수는 현재 수행 중인 태스크의 process 배열 변수의 인덱스이다.

init\_task 함수에서는 current, next 포인터 변수를 초기화하고 있다. 최초로 수행할 태스크가 task0이므로 current 포인터 변수로 process[0] 변수를 가리키게 했다. next 변수는 일단 process[0] 변수를 가리키게 했다. process[1] 변수는



앞에서도 말했듯이 task1 태스크를 관리하는 역할을 하며, 나중에 문맥 전환시 task1 태스크의 SP(R13), PC(R15), CPSR 레지스터에 각각 0x408000, 0x400000, 0x50 값이 들어가도록 context 변수를 초기화 한다(MASTERL1PT, TASK0L2PTBASE, TASK1L2PTBASE 매크로에 대해서는 과월호 참조).

schedule 함수에서는 pid 변수를 갱신하여 다음에 수행할 태스크를 선택하고, current 변수는 새로 수행할 태스크를 가리키게 한다.

또 문맥 전환을 완료하기 위하여 next 변수도 새로 수행할 태스크를 가리키게 한다. 새로 수행할 태스크 루틴으로 뛰기 위해 flush\_cache\_tlb 함수를 이용해 cache와 tlb를 비운 후,

mmu\_map\_l2pt 함수를 이용하여 master page table에 다음 수행할 태스크의 level 2 page table을 맵핑시킨다. schedule 함수는 eventsSWIHandler, events IRQHandler 함수에서 호출한다.

이상에서 task.c 파일의 내용을 살펴보았다. 다음은 exceptions.S 파일 내의 coreSWIHandler, coreIRQHandler 함수에 문맥을 저장하고 복구하는 루틴을 추가해 보자.

다음은 exceptions.S 파일의 내용이다.

```
exceptions.S
...
#define svc_stack      0x100000
.globl coreSWIHandler
coreSWIHandler:
    ldr r13,=current
    ldr r13,[r13]          @ r13->current
    add r13,r13,#8
    stmia r13,{r0-r14}^    @ save user registers
    mrs r0,spsr
    stmdb r13,{r0,r14}     @ save the rest
    ldr r13,=svc_stack
    ldr r10,[r13,#-4]
    bic r10,r10,#0xff000000
    mov r0,r10
    bl eventsSWIHandler
    ldr r13,=next
    ldr r13,[r13]          @ r13->next
    add r13,r13,#8
    ldmdb r13,{r0,r14}
    msr spsr_cxsf,r0
    ldmia r13,{r0-r14}^    @ load user registers
    movs pc,lr             @ return next task

#define irq_stack      (0x100000-0x1000)
```

```
.globl coreIRQHandler
coreIRQHandler:
    sub r14,r14,#4
    ldr r13,=current
    ldr r13,[r13]
    add r13,r13,#8
    stmia r13,{r0-r14}^
    mrs r0,spsr
    stmdb r13,{r0,r14}
    ldr r13,=irq_stack
    bl eventsIRQHandler
    ldr r13,=next
    ldr r13,[r13]
    add r13,r13,#8
    ldmdb r13,{r0,r14}
    msr spsr_cxsf,r0
    ldmia r13,{r0-r14}^
    movs pc,lr
```

coreSWIHandler 함수는 크게 세 부분으로 구성된다. 현재 수행하던 태스크의 문맥을 저장하는 부분, 시스템 콜을 처리하는 부분, 다음에 수행할 태스크의 문맥을 복구하는 부분이다. 여기에서는 편의상 coreSWIHandler 함수 내에 스케줄링을 수행하는 루틴을 넣었다.

문맥을 저장하는 부분에서는 먼저 current 포인터 변수의 위치 값을 R13\_svc 레지스터로 읽어온 후, current 포인터 변수가 가리키는 주소 값을 R13\_svc 레지스터로 읽어온다. 그리고 R13\_svc 레지스터의 값을 8바이트만큼 증가시킨 후, R13\_svc 레지스터가 가리키는 위치를 기준으로 메모리의 높은 주소 부분에 R0~R14의 15개의 레지스터를 저장한다.

이 때 R13, R14는 USER 모드의 레지스터이다. 그리고 SPSR\_svc, R14\_svc 레지스터의 값을 R13\_svc 레지스터가 가리키는 위치를 기준으로 메모리의 낮은 주소 부분에 저장한다. 이 두 레지스터는 USER 모드에서의 CPSR, PC 레지

스터의 값을 가지고 있다. 이상에서 현재 수행 중이던 태스크의 문맥은 그림 1과 같이 저장된다. 태스크의 문맥을 저장하고 나면 R13\_svc 레지스터에 SVC 모드의 스택(=0x100000) 값을 넣고 시스템 콜을 처리한다.

문맥을 복구하는 부분에서는 next 포인터 변수의 위치 값을 R13\_svc 레지스터로 읽어온 후, next 포인터 변수가 가리키는 주소 값을 R13\_svc 레지스터로 읽어온다. 그리고 R13\_svc 레지스터의 값을 8바이트만큼 증가시킨 후, R13\_svc 레지스터가 가리키는 위치를 기준으로 메모리의 낮은 주소 부분으로부터 SPSR\_svc, R14\_svc 레지스터의 값을 복구한다.

그리고 R13\_svc 레지스터가 가리키는 위치를 기준으로 메모리의 높은 주소 부분으로부터 R0~R14의 15개의 레지스터의 값을 복구한다. 이상의 내용이 다음에 수행할 태스크의 문맥을 복구하는 부분이다.

태스크의 문맥을 복구한 후에는 사용자 영역으로 빠져나간다. 그림 2는 task0 태스크의 문맥을 저장하고, task1 태스크의 문맥을 복구하는 예이다.

coreIRQHandler 함수도 크게 세 부분으로 구성된다. 즉, 현재 수행하던 태스크의 문맥을 저장하는 부분, 하드웨어 인터럽트를 처리하는 부분, 그리고 다음에 수행할 태스크의 문맥을 복구하는 부분으로 구성된다. 편의상 여기에서는 coreIRQHandler 함수 내에 스케줄링을 수행하는 루틴을 넣었다. coreIRQHandler 함수에서 문맥을 저장하고 복구하는 과정은 coreSWIHandler와 그 구조가 동일하므로 비교해 보기 바란다. 이상으로 exceptions.S 파일의 내용을 살펴보았다.

OS를 이해하기 위해서는 multitasking의 원리를 이해해야 하고, multitasking의 원리를 이해하기 위해서는 문맥 전환의 원리를 반드시 이해해야 한다. 따라서 위 코드의 내용이 잘 이해가 되지 않는다면 반복적으로 코드의 내용을 살펴보고 수행시켜 가며 꼭 그 원리를 이해해야만 할 것이다.

다음은 swihandler.c 파일의 내용이다.

```
swihandler.c
void eventsSWIHandler(unsigned int syscallnum)
```

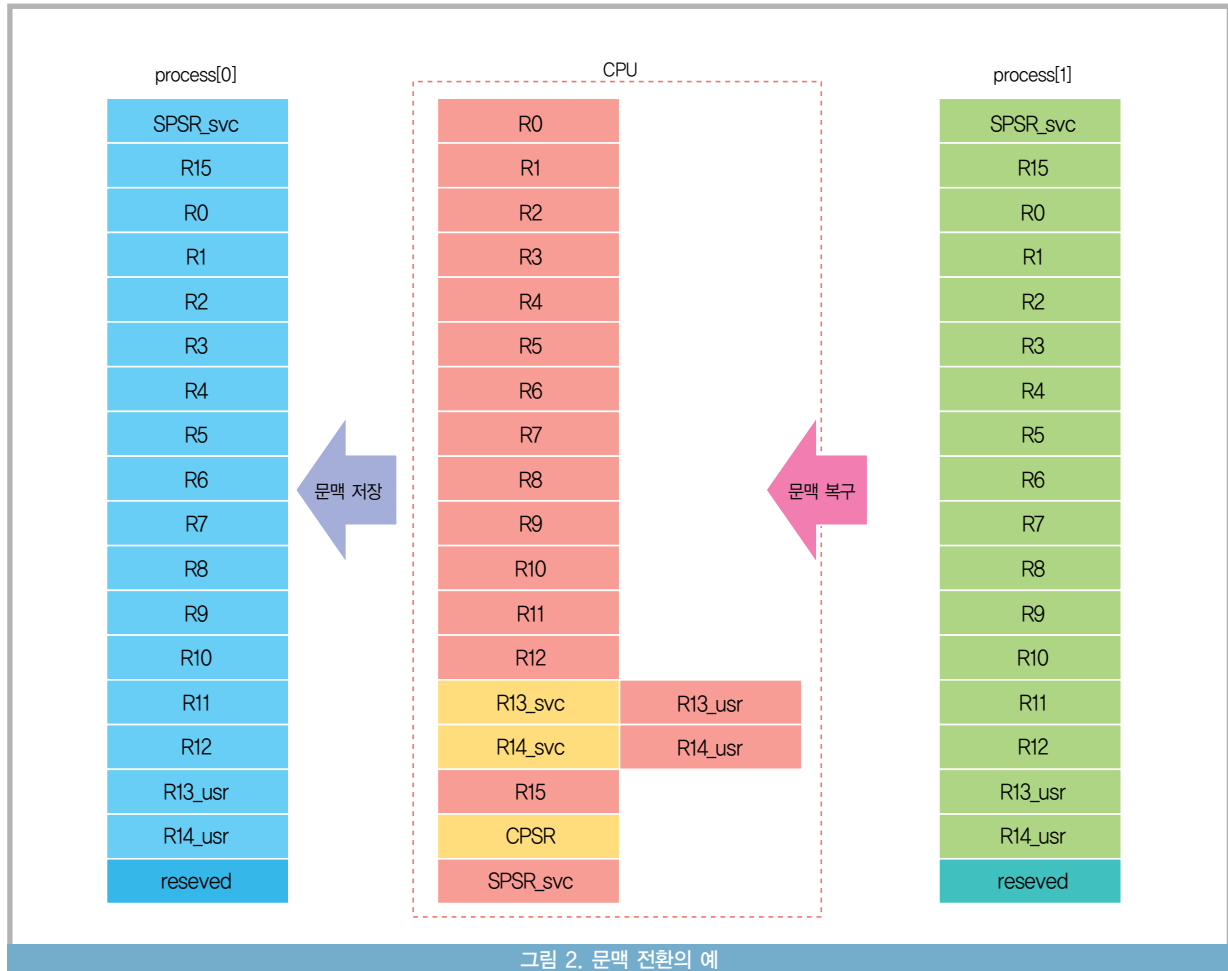


그림 2. 문맥 전환의 예

```
{
    static unsigned int cnt = 0;
    uart_puts("SWI handler: ");
    uart_puts("syscall number-");
    uart_puthex(syscallnum);
    uart_puts(", cnt-");
    uart_puthexn(cnt++);
    schedule();
}
```

eventsSWIHandler 함수는 schedule 함수를 호출하는 부분이 추가된 점 외에는 이전 기사의 예제와 같다. 참고로 여

기서는 편의상 schedule 함수를 항상 호출하는데, 실제 OS에서는 그때그때 시스템 콜 처리 루틴 내에서 어떤 조건에 따라 schedule 함수를 호출한다.

예를 들어 시스템 콜 처리 루틴 내에서 공유 영역 등을 접근하려고 하지만, 다른 태스크가 이미 접근하고 있어 당장 접근이 불가능할 경우 sleep\_on과 같은 함수를 쓰게 되는데 이러한 함수 내에서 현재 태스크를 대기 큐에 넣고 schedule 함수를 호출한다.

다음은 irqhandler.c 파일의 내용이다.

irqhandler.c

```
include <timer.h>
void eventsIRQHandler()
{
    unsigned int rintoffset;
    unsigned int rintpnd;
    uart_puts("IRQ handler: ");
    rintoffset = rINTOFFSET;
    uart_puts("rINTOFFSET - ");
    uart_puthex(rintoffset);
    rintpnd = rINTPND;
    uart_puts(", rINTPND - ");
    uart_puthexnl(rintpnd);
    rSRCPND |= rintpnd;
    rINTPND |= rintpnd;
    schedule();
}
```

eventsIRQHandler 함수도 schedule 함수를 호출하는 부분이 추가된 점 외에는 이전 예제와 같다. 참고로 여기에서도 편의상 schedule 함수를 항상 호출하는데, 실제 OS에서는 그때그때 하드웨어 인터럽트 처리 루틴 내에서 어떤 조건에 따라 schedule 함수를 호출한다.

예를 들어 하드웨어 인터럽트 처리 루틴으로부터 어떤 데이터를 기다리는 태스크가 대기 큐에 머물러 있을 경우 wake\_up과 같은 함수 내에서 그 태스크를 대기 큐에서 준비 큐로 옮기고 대기 큐로 옮겨진 태스크가 현재 수행중인 태스크보다 우선순위가 클 경우 스케줄링을 수행할 수 있다.

이상 multitasking을 수행하기 위해 필요한 루틴의 내용을 살펴보았다. 마지막으로 Makefile 파일의 KERNEL\_OBJ 변수에 다음과 같이 task.o 부분을 추가한다.

```
KERNEL_OBJ=head.o main.o mmusetup.o uart.o
exceptions.o swihandler.o timer.o
irqhandler.o mmu.o task.o
```

이상 작성한 내용을 컴파일하면 수행한 결과가 아래와 같이 나타난다. task0와 task1 태스크간에 문맥 전환을 수행함을 알 수 있다.

```
...
SWI handler: syscall number-0x00000081, cnt-0x00000365
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-0x00000400
SWI handler: syscall number-0x00000081, cnt-0x00000366
...
SWI handler: syscall number-0x00000080, cnt-0x00000d3f
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-0x00000400
SWI handler: syscall number-0x00000080, cnt-0x00000d40
...
```

이상에서 multitasking을 수행하기 위해 필요한 문맥 전환과 스케줄링 루틴을 추가해 보았다. 다음은 하드웨어 인터럽트 처리 루틴과 태스크 루틴간 경쟁 상태가 발생하는 이유를 알아보자.

OS 상에서 루틴간 경쟁 상태는 공유 영역을 중심으로 발생한다. 따라서 먼저 공유영역을 만들어 보자.

다음은 main.c 파일의 내용이다.

```
main.c
#include <system.h>
void uart_init();
#define TASK0L2PT (unsigned int *) (0x100000+0x4400)
#define TASK1L2PT (unsigned int *) (0x100000+0x4800)
#define SHARED_L2PT (unsigned int *) (0x100000+0x4c00)
```

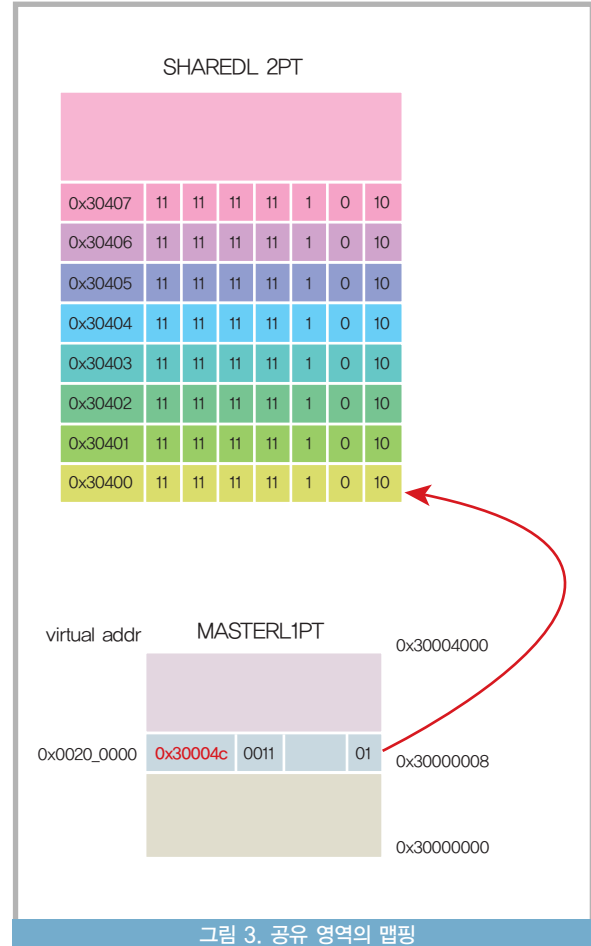
```
#define TASK0BASE (0x30200000|0xff<<4|0x2<<2|0x2)
#define TASK1BASE (0x30300000|0xff<<4|0x2<<2|0x2)
#define SHAREDBASE (0x30400000|0xff<<4|0x2<<2|0x2)

#define MASTERL1PT (unsigned int *) (0x100000)
#define TASK0L2PTBASE (0x30004400|0x03<<5|1<<4|0x1)
#define TASK1L2PTBASE (0x30004800|0x03<<5|1<<4|0x1)
#define SHAREDL2PTBASE (0x30004c00|0x03<<5|1<<4|0x1)

void mmu_fill_l2pt(unsigned int *, unsigned int *,
unsigned int);
void flush_cache_tlb0;
void mmu_map_l2pt(unsigned int *, unsigned int);

#define jump2task0() \
({ \
    unsigned long spsr; \
    unsigned long task0addr=0x400000; \
    unsigned long svcsp=0x100000; \
    __asm__ __volatile__ ( \
        "sub sp,%2,#4\n" \
        "mrs %0,spsr\n" \
        "bic %0,%0,#0xff\n" \
        "orr %0,%0,#0x5f\n" \
        "msr spsr_c,%0\n" \
        "movs pc,%1" \
        :: "r" (spsr), "r" (task0addr) \
        , "r" (svcsp) \
        : "memory", "cc"); \
    })

void kmain(void)
{
```



```
uart_init();
timer_init();
mmu_fill_l2pt(TASK0L2PT, TASK0L2PT+8, TASK0BASE);
mmu_fill_l2pt(TASK1L2PT, TASK1L2PT+8, TASK1BASE);
mmu_fill_l2pt(SHARED L2PT, SHARED L2PT+8,
SHARED BASE);
flush_cache_tlb0;
mmu_map_l2pt(MASTERL1PT+4, TASK0L2PTBASE);
mmu_map_l2pt(MASTERL1PT+2, SHARED L2PTBASE);
init_task0;
uart_puts("jump to task0\n");
```

```

*(volatile unsigned int *)0x200000 = 0;

jump2task0();
}

```

SHARED\_L2PT 매크로는 공유 영역을 관리할 level 2 page table의 가상 주소를 나타낸다. SHARED\_BASE 매크로는 공유 영역의 물리 주소와 AP, C, B 속성 등을 나타낸다. SHARED\_L2PT\_BASE 매크로는 공유 영역을 관리할 level 2 page table의 물리 주소와 Domain, C, B 속성 등을 나타낸다.

kmain 함수에서는 mmu\_fill\_l2pt 함수를 이용해 공유 영역의 level 2 page table을 초기화하고, master page table에 공유 영역의 level 2 page table도 맵핑시킨다. 그림 3은 이 과정에 의해서 생성된 공유 영역의 level 2 page table과 master page table을 나타낸 것이다. 공유 영역은 가상 주소 공간상에 0x200000 번지에 존재하도록 설정했다. 여기서는 0x200000 번지의 4바이트를 공유 영역으로 사용한다. kmain에서 0x200000 번지의 4바이트를 0으로 초기화 했다. 나머지 부분은 이전 기사의 예제에서 이미 설명한 부분이므로 참조하기 바란다.

여기서 주의할 점은 jump2task0 매크로에서 SVC 모드의 spsr 레지스터의 하위 8 비트를 0으로 초기화하고 0x5f 값으로 채우는 부분이다. 이렇게 할 경우 movs 명령어를 이용하여 가상 주소 0x400000 번지로 점프를 수행하면 CPU 코어의 모드는 SYSTEM 모드로 바뀐다. 여기서는 디버깅의 편의상 task0, task1 루틴에서 uart 영역을 직접 접근하게 했다. 그래서 편의상 태스크는 SYSTEM 모드에서 동작한다. 따라서 다음과 같이 task.c 파일의 init\_task 함수에서 task1 태스크의 CPSR 레지스터에도 0x5f 값이 놓인다. 나머지 부분은 이전 예제와 같다.

```

task.c
...
void init_task()
{
    current = &process[0];
    next = current;
}

```

```

process[1].context[13+2] = 0x408000;
process[1].context[1] = 0x400000;
process[1].context[0] = 0x5f;
}
...

```

이상에서 main.c 파일과 task.c 파일의 내용을 살펴보았다. 다음은 irqhandler.c 파일의 내용이다.

```

irqhandler.c
#include <timer.h>

void eventsIRQHandler()
{
    unsigned int rintoffset;
    unsigned int rintpnd;
    rintoffset = rINTOFFSET;
    rintpnd = rINTPND;
    {
        #define SUM *(volatile unsigned int *)0x200000
        static unsigned int tmp_sum;
        static unsigned int count = 0;
        static unsigned int do_sum = 1;
        if(do_sum == 0) goto sched;
        tmp_sum = SUM;
        if(tmp_sum >= 0x100000) {
            do_sum = 0;
            uart_puts("sum in timer interrupt handler: ");
            uart_puthexnl(SUM);
            uart_puts("count in timer interrupt handler: ");
            uart_puthexnl(count);
            goto sched;
        }
    }
}

```



```

    tmp_sum++;
    SUM = tmp_sum;
    count++;
}

rSRCPND |= rintpnd;
rINTPND |= rintpnd;

sched:
    schedule();
}

```

eventsIRQHandler 함수 내의 중첩된 블록을 살펴보자. 다른 부분은 이전 예제와 같다. 중첩된 블록 내에서는 공유 변수 SUM 값(kmain 함수에서 0으로 초기화)을 지역 변수 tmp\_sum으로 읽어와 0x100000 값과 비교한 후, 작으면 tmp\_sum 변수 값을 1 증가시킨 후 SUM 변수 값에 넣고 지역 변수 count 값을 하나 증가시키고 루틴을 빠져 나간다. 만약 tmp\_sum 변수의 값이 0x100000 값보다 크거나 같으면 지역 변수 do\_sum 값을 0으로 채운 후, 공유 변수 SUM 값과 지역 변수 count 값을 uart로 찍고 나간다. do\_sum 변수 값은 중첩된 블록의 앞부분에서 1로 초기화 되며, 일단 0이 되면 공유 변수 SUM을 접근하지 못하도록 하는 역할을 한다. count 변수는 SUM 변수 값이 0x100000이 되는 동안 eventsIRQHandler 함수 내의 중첩된 블록에서 SUM 변수 값에 몇 번 접근했는지 알려준다.

다음은 task0.c 파일의 내용을 살펴보자.

```

task0.c
#include <system.h>

int main()
{
#define SUM *(volatile unsigned int *)0x200000

```

```

    unsigned int tmp_sum;
    unsigned int count = 0;
    while(1) {
        tmp_sum = SUM;
        if(tmp_sum >= 0x100000) break;
        tmp_sum++;
        SUM = tmp_sum;
        count++;
    }

    local_irq_disable();
    uart_puts("sum in task0: ");
    uart_puthexnl(SUM);
    uart_puts("count in task0: ");
    uart_puthexnl(count);
    local_irq_enable();

    for(;;);
}

```

task0.c 파일의 main 루틴 내에서는 무한 루프를 돌면서 공유 변수 SUM 값을 지역 변수 tmp\_sum으로 읽어와 0x100000 값과 비교한 후, 작으면 tmp\_sum 변수 값을 1 증가시킨 후 SUM 변수 값에 넣고 지역 변수 count 값을 하나 증가시킨다. 만약 tmp\_sum 변수의 값이 0x100000 값보다 크거나 같으면 루프를 빠져 나와 공유 변수 SUM 값과 지역 변수 count 값을 uart로 찍고 나간다. count 변수는 SUM 변수 값이 0x100000이 되는 동안 task0 태스크 루틴에서 SUM 변수 값을 몇 번 접근했는지를 알려주는 역할을 한다.

eventsIRQHandler 함수와 task0.c 파일의 main 함수에서는 공유 변수 SUM에 대해 경쟁 상태를 발생시킨다. 경쟁 상태의 결과가 어떻게 나오는지 뒤에서 살펴보자.

다음은 디버깅의 편의상 swihandler.c 파일의 eventsSWIHandler 함수와 task1.c 파일의 main 함수를 다음과 같이 수정한다.

```
swihandler.c
void eventsSWIHandler(unsigned int syscallnum)
{
    schedule();
}
task1.c
int main()
{
    while(1) ;
}
```

두 함수에서 디버깅 메시지를 찍지 않도록 했다. 그리고 Makefile의 TASK0, TASK1 부분을 다음과 같이 수정한다. 다른 부분은 그대로 둔다.

```
Makefile
...
# TASK0
TASK0_OBJ = task0.o uart.o
task0.bin.o: task0
    arm-linux-ld -r -o task0.bin.o -b binary task0.bin
task0: $(TASK0_OBJ)
```

```
arm-linux-ld $(TASK0_OBJ) -o task0 -e main -Ttext
0x400000 -N
    arm-linux-objcopy task0 task0.bin -O binary
# TASK1
TASK1_OBJ = task1.o uart.o
task1.bin.o: task1
    arm-linux-ld -r -o task1.bin.o -b binary task1.bin
task1: $(TASK1_OBJ)
    arm-linux-ld $(TASK1_OBJ) -o task1 -e main -Ttext
0x400000 -N
    arm-linux-objcopy task1 task1.bin -O binary
...
```

task1, task2 루틴에 uart 관련 함수를 추가하도록 Make file을 수정했다. 이상 작성한 내용을 컴파일하면 다음과 같은 결과를 볼 수 있다.

```
jump to task0
sum in task0: 0x00100000
count in task0: 0x000fffe6
sum in timer interrupt handler: 0x00100000
count in timer interrupt handler: 0x0000006a
```

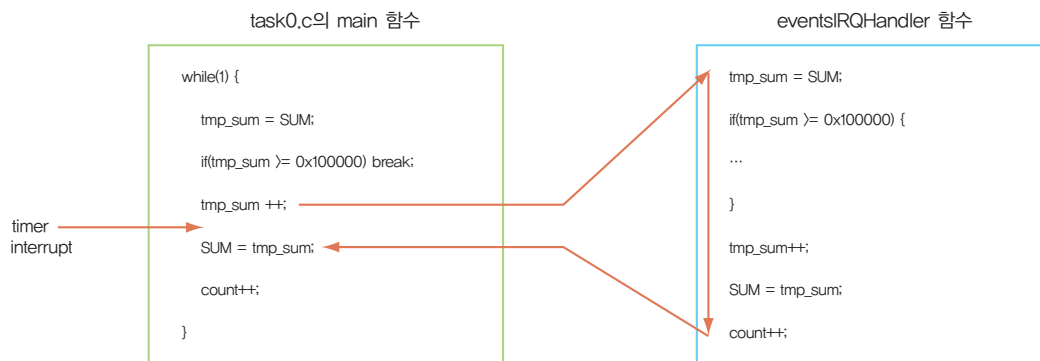


그림 4. task0 루틴에 대한 하드웨어 인터럽트 처리 루틴의 간섭

여기서 SUM 값은 0x100000인데 반해, 인터럽트 처리 루틴의 count 값과 task0 루틴의 count 값을 더할 경우 0x100050이 나오는 걸 확인할 수 있다. 이러한 결과는 두 루틴이 SUM 변수를 동시에 접근하는 경우가 발생하기 때문이다. 예를 들어 그림 4와 같이 두 루틴이 겹칠 경우 이와 같은 문제가 발생한다.

그림 4는 task0 루틴에서 공유 변수 SUM을 접근하는 도중에 timer interrupt가 발생해 eventsIRQHandler 함수 내에서 SUM 변수를 접근한 후 task0 루틴으로 되돌아와 SUM 변수를 다시 접근하는 상황을 나타낸다. 어느 순간의 SUM 값이 0x1000이라고 생각해보자. 위의 경우 두 루틴이 한 번씩 수행되고 나서는 0x1001이 된다. 즉, eventsIRQHandler 함수에서 수행한 동작은 의미 없는 동작이 되고 만다.

이와 같이 일반적으로 태스크 루틴과 하드웨어 인터럽트 루틴간에 공유 영역이 있을 경우 경쟁 상태가 발생할 수 있으며, 이는 시스템에 치명적인 결과를 가져올 수 있다. 이에 대한 해결책은 다음 호에서 살펴볼 것이다.

마지막으로 태스크 루틴과 태스크 루틴간의 경쟁 상태가 발생하는 상황을 살펴보자. 먼저 task1.c 파일의 내용을 다음과 같이 작성하자. task1.c는 task0.c의 내용 및 구조가 동일하다.

```
task1.c
#include <system.h>
int main()
{
#define SUM *(volatile unsigned int *)0x200000
    unsigned int tmp_sum;
    unsigned int count = 0;
    while(1) {
        tmp_sum = SUM;
        if(tmp_sum >= 0x100000) break;
        tmp_sum++;
        SUM = tmp_sum;
        count++;
    }
```

```
local_irq_disable();
uart_puts("sum in task1: ");
uart_puthexnl(SUM);
uart_puts("count in task1: ");
uart_puthexnl(count);
local_irq_enable();
for(;;);
}
```

irqhandler.c 파일의 eventsIRQHandler 함수도 다음과 같이 수정한다.

```
irqhandler.c
#include <timer.h>
void eventsIRQHandler()
{
    unsigned int rintoffset;
    unsigned int rintpnd;
    rintoffset = rINTOFFSET;
    rintpnd = rINTPND;
    rSRCPND |= rintpnd;
    rINTPND |= rintpnd;

    sched:
        schedule();
}
```

그 외의 다른 파일들은 그대로 두고 사용한다. 이상 작성한 내용을 컴파일하면 다음과 같은 결과가 나타난다.

```
jump to task0
sum in task1: 0x00100000
```

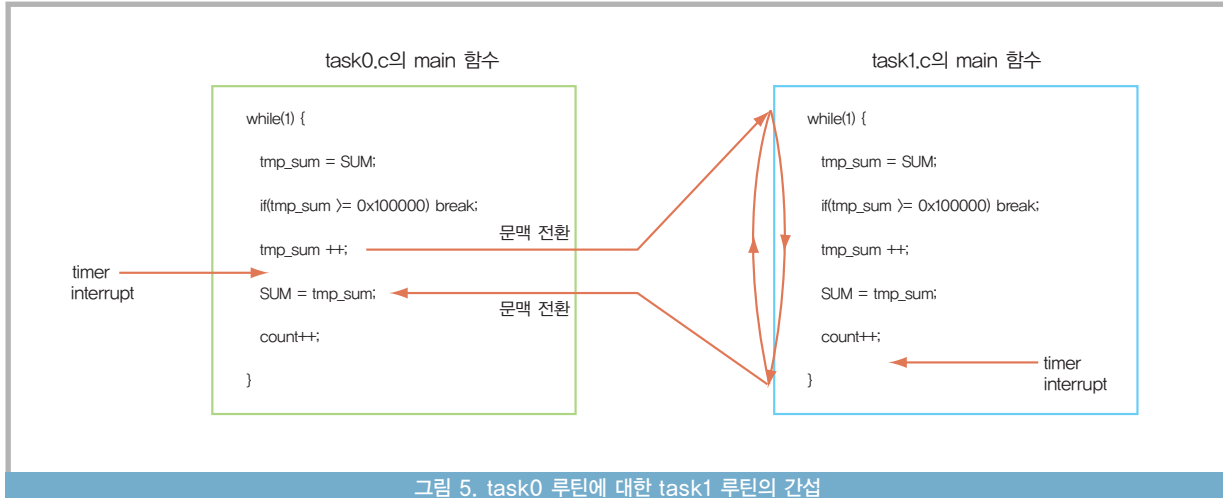


그림 5. task0 루틴에 대한 task1 루틴의 간섭

count in task1: 0x000dff3b  
sum in task0: 0x00100000  
count in task0: 0x000e3540

여기서 SUM 값은 0x100000으로 나오는데 반해 task0 루틴의 count 값과 task1 루틴의 count 값을 더할 경우 0x1c347b가 나오는 걸 확인할 수 있다. 이러한 결과는 두 루틴이 SUM 변수를 동시에 접근하는 경우가 발생하기 때문이다. 예를 들어 그림 5와 같이 두 루틴이 겹칠 경우 이와 같은 문제가 발생한다.

그림 5는 task0 루틴에서 공유 변수 SUM을 접근하는 도중에 timer interrupt가 발생하여 coreIRQHandler 커널 루틴 내에서 스케줄링과 문맥 전환을 수행하고, task1 루틴으로 뻗 후 한 번 이상(예를 들어, 0x100번)의 루프를 돈 다음에 다시 timer interrupt가 발생해 coreIRQHandler 커널 루틴 내에서 스케줄링과 문맥 전환을 수행하고 task0 루틴으로 되돌아와 SUM 변수를 다시 접근하는 상황을 나타낸다.

어느 순간의 SUM 값이 0x1000이라고 생각해보자. task0 루틴에서 루프 문을 한 번 수행하는 동안 task1 루틴에서 루프 문을 0x100 번 수행하더라도 위와 같이 수행될 경우는 결과적으로 공유 변수 SUM 값이 0x1001이 된다. 즉, task1 루틴이 수행한 0x100 번의 동작은 의미 없는 동작이 되고 만다.

이와 같이 일반적으로 태스크 루틴과 태스크 루틴간에 공

유 영역이 있을 경우 경쟁 상태가 발생할 수 있으며, 이는 시스템에 치명적인 결과를 가져올 수 있다. 이에 대한 해결책도 다음 호에서 알아보자.

이상에서 우리는 비동기적으로 발생하는 하드웨어 인터럽트에 의해 태스크 루틴과 하드웨어 인터럽트 처리 루틴간에 공유 영역에 대한 경쟁 상태가 발생하는 상황을 보았다. 또 하드웨어 인터럽트 처리 루틴 내에서의 문맥 전환으로 인해 태스크 루틴과 태스크 루틴간에 공유 영역에 대한 경쟁 상태가 발생하는 상황을 보았다.

일반적으로 공유 영역에 대한 경쟁 상태는 시스템에 치명적인 결과를 주게 된다. 따라서 이러한 문제는 반드시 해결해 주어야 한다. 이러한 문제에 대한 일반적인 해결책은 다음 호에서 살펴보도록 하자.  $R_{T_{int}}^{rel}$

### 참고자료

ARM System Developer's Guide: Designing and Optimizing System Software

저자: Andrew Sloss, Dominic Symes, Chris Wright

출판사: Morgan Kaufmann