

설계와 구현을 통한 임베디드 OS의 이해와 응용 ⑧

동기화 문제의 해결과 우선 순위 큐

지난 호에 우리는 태스크 루틴과 하드웨어 인터럽트 처리 루틴간에 또 태스크 루틴과 태스크 루틴간에 공유 영역에 대한 경쟁 상태가 발생하는 상황을 보았다. 이번 호에는 이러한 문제들에 대한 일반적인 해결책을 알아보고 필요한 루틴을 추가해 보자. 또 라운드 로빈 방식과 우선 순위 방식의 스케줄링을 구현하기 위해 필요한 일반 큐와 우선 순위 큐를 구현해 보기로 하자.

글 : 서민우 / 새롬전자

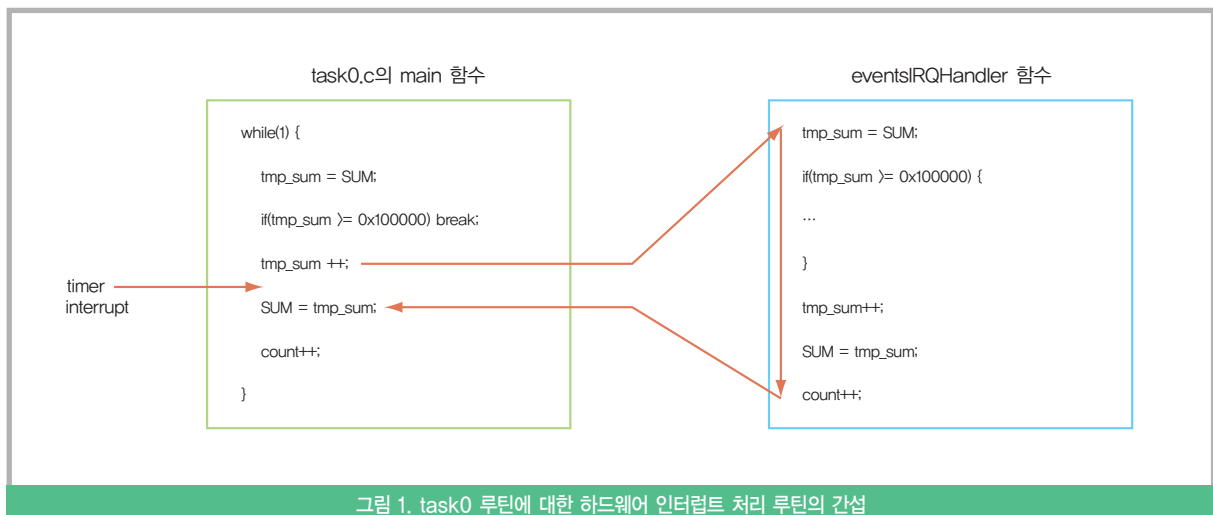
mwseo@e-serome.co.kr / www.e-serome.co.kr

그림 1은 태스크 루틴과 하드웨어 인터럽트 루틴간에 공유 영역이 있을 경우 경쟁 상태가 발생할 수 있는 상황을 나타낸다.

이러한 상황은 태스크 루틴에서 공유영역을 접근하는 도중에 하드웨어 인터럽트가 발생하여 하드웨어 인터럽트 처리 루틴에서 같은 공유영역을 접근할 경우 발생한다. 따라서 하드웨어 인터럽트가 발생하면 문제가 발생할 수 있는 영역에서 하드웨어 인터럽트를 막아 주어야 한다.

즉, 그림 2에서 SUM 값을 읽어 값을 1 증가시키고 다시 저장하는 루틴, 즉, critical section에서 인터럽트를 막아줌으로써 문제를 해결할 수 있다. 따라서 task0.c 파일의 main 루틴을 다음과 같이 수정한다.

local_irq_enable, local_irq_disable 매크로는 include/system.h 파일에 정의되어 있으며 이전 기사에서 언급한 바 있다. 다른 파일들은 그대로 둔다. 이상 작성한 내용을 컴파일하여



* 연재순서

설계와 구현을 통한 임베디드 OS의 이해와 응용

- 1회 Overview
- 2회 개발환경 구성 및 부트로더 작성
- 3회 cuteOS의 시작
- 4회 MMU와 캐시 설정
- 5회 주요 루틴 작성하기
- 6회 하드웨어 인터럽트 처리 및 태스크 추가 루틴
- 7회 문맥 전환 및 스케줄링 루틴 작성
- 8회 동기화 문제의 해결과 우선 순위 큐**
- 9회 우선 순위 기반 스케줄링 작성
- 10회 IPC의 설계 및 구현

서민우(mwseo@e-serome.co.kr)

리눅스 커널, RTOS 커널의 구조에 관심을 가지고 연구하고 있으며, 리눅스나 RTOS 디바이스 드라이버 개발을 주업으로 하고 있다.

본 기사를 통하여 MMU 기능이 있는 cuteOS라는 마이크로 커널을 구현하였다. 과거에 32 비트 마이크로 프로세서를 설계하고 VHDL을 이용하여 구현한 경험이 있다. 현재 (주)새롭전자에서 영상칩을 제어하기 위해 펌웨어, 리눅스 디바이스 드라이버, USB WDM 디바이스 드라이버 구현과 관련된 일들을 하고 있다.

확인해 보면 다음과 같은 결과를 볼 수 있다.

```
jump to task0
sum in task0: 0x00100000
count in task0: 0x000ffa9
sum in timer interrupt handler: 0x00100000
count in timer interrupt handler: 0x00000057
```

여기서 SUM 값은 0x100000으로 나온다. 또 task0 루틴의 count 값과 인터럽트 처리 루틴의 count 값을 더할 경우에도 0x100000이 나오는 걸 확인할 수 있다. 즉, 하드웨어 인터럽트를 막음으로써 두 루틴이 SUM 변수 값을 차례대로 접근하도록 하여 공유영역에 대한 경쟁 상태를 해결하였다.

어떤 하드웨어 인터럽트(인터럽트 A)를 처리하는 도중에 또 다른 하드웨어 인터럽트(인터럽트 B - 중첩 인터럽트)가 발생하여 그 하드웨어 인터럽트(인터럽트 B)를 처리해야 할 경우에도 두 루틴간에 공유영역이 있을 경우에는 문제가 발생할 수 있다. 그림 3을 보자.

그림 3과 같은 경우에도 그림 2에서처럼 하드웨어 인터럽트가 발생하면 문제가 발생할 수 있는 영역에서 하드웨어 인터럽트를 막아 주어야 한다.

일반적으로 태스크 루틴과 하드웨어 인터럽트 처리 루틴간 또는 하드웨어 인터럽트 처리 루틴과 하드웨어 인터럽트 처리

루틴간에 공유 영역이 있을 경우 또 그 공유영역을 접근하는 시간이 일정할 경우 인터럽트를 막음으로써 문제를 해결할 수 있다.

다음은 태스크 루틴과 태스크 루틴간에 공유 영역에 대한 경쟁 상태가 발생할 경우 어떻게 문제를 해결할 지 생각해 보자. 그림 4는 태스크 루틴과 태스크 루틴간에 공유 영역이 있을 경우 경쟁 상태가 발생할 수 있는 상황을 나타낸다.

그림 4와 같은 경우에도 공유영역을 접근하는 시간이 일정하므로 인터럽트를 막음으로써 문제를 해결할 수 있다. 즉, 그림 2와 같이 처리할 수 있다. 그러나 태스크 루틴에서 공유영역을 접근하는 시간이 일정하지 않을 경우 또는 태스크 루틴에서 공유

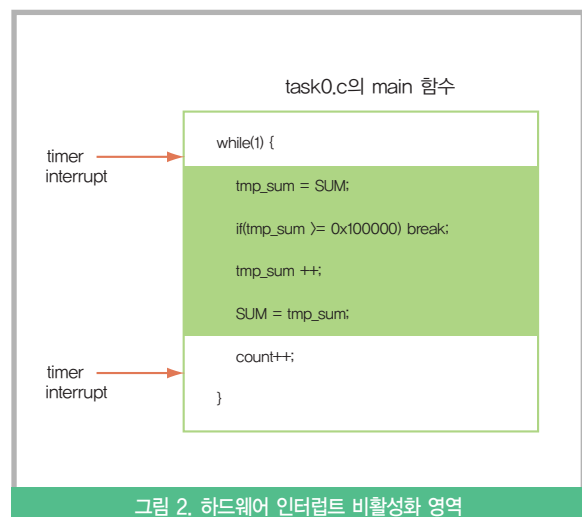


그림 2. 하드웨어 인터럽트 비활성화 영역

영역에 대한 접근을 시작하고 하드웨어 인터럽트 루틴에서 공유영역에 대한 접근을 끝낼 경우에는 인터럽트를 막는 방법으로 문제를 해결할 수 없다. 이런 경우에는 key나 flag 역할을 하는 변수를 두어 문제가 발생할 수 있는 영역을 태스크간에 순차적으로 접근하게 해야 한다.

그림 5는 SUM 값을 읽어 값을 1 증가시키고 다시 저장하는 루틴, 즉, critical section의 시작 지점에서 flag를 내리고 끝나는 지점에서 flag를 올리고 나오는 상황을 나타낸다. flag가 이미 내려가 있을 경우에는 flag가 올라갈 때까지 busy waiting을 수행한다.

task0 루틴에서 flag를 내리고 공유 변수 SUM을 접근하는 도중에 timer interrupt가 발생하여 coreIRQHandler 커널 루틴내에서 스케줄링과 문맥 전환을 수행하고, task1 루틴으로 뛴 후 task1 루틴에서 공유 변수를 접근하고자 하여도 flag가 이미 내려가 있으므로 task1 루틴에서는 flag가 올라갈 때까지 busy waiting을 수행한다. 그러면 [그림 5]의 내용에 맞게 파일의 내용을 수정해 보기로 하자.

먼저 main.c 파일의 내용을 다음과 같이 수정한다.

```
main.c
...
void kmain(void)
{
...

    uart_puts("jump to task0\n");

    *(volatile unsigned int *)0x200000 = 0;
    *(volatile unsigned int *)0x200004 = 1;

    jump2task0();
}
```

가상 주소 0x200004 번지의 4 바이트는 flag로 사용할 영역이며 이 영역을 1로 초기화하고 있다. 즉, flag가 올라가 있는 상

태로 초기화를 하고 있다.

다음은 task0.c 파일의 내용을 다음과 같이 수정한다.

```
task0.c
#include <system.h>

int main()
{
#define SUM    *(volatile unsigned int *)0x200000
#define SUM_LOCK    *(volatile unsigned int *)0x200004

    unsigned int tmp_sum;
    unsigned int count = 0;

    while(1) {
        while(1) {
            local_irq_disable();
            if(SUM_LOCK == 1) {
                SUM_LOCK = 0;

                local_irq_enable();
                break;
            }
            local_irq_enable();
        }

        tmp_sum = SUM;
        if(tmp_sum >= 0x100000) {
            local_irq_disable();
            SUM_LOCK = 1;
            local_irq_enable();
            break;
        }
    }
}
```

```

    }

    tmp_sum++;
    SUM = tmp_sum;

    local_irq_disable();
    SUM_LOCK = 1;
    local_irq_enable();

    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 루틴에서 flag를 내리는 동작은 다음 부분이다.

```

while(1) {
    local_irq_disable();
    if(SUM_LOCK == 1) {
        SUM_LOCK = 0;
        local_irq_enable();
        break;
    }
    local_irq_enable();
}

```

즉, SUM_LOCK이 올라가 있으면 SUM_LOCK을 내리고 critical section으로 들어가고 SUM_LOCK이 내려가 있으면 SUM_LOCK이 올라갈 때까지 busy waiting을 수행하며 기다린다. 여기서 한 가지 주의할 점은 SUM_LOCK 변수 역시 SUM 변수처럼 공유영역이기 때문에 인터럽트를 막아 보호해 주어야 한다.

critical section에 대한 접근을 마친 후에 flag를 올리는 동작은 다음 부분이다.

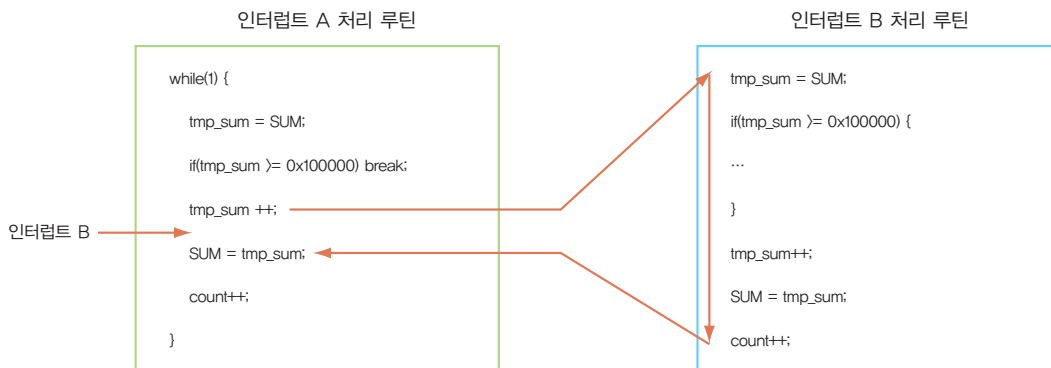


그림 3. 하드웨어 인터럽트 처리 루틴간의 간섭

```
local_irq_disable();
SUM_LOCK = 1;
local_irq_enable();
```

즉, SUM_LOCK을 올리며 critical section을 빠져 나온다.
task1.c 파일의 내용도 task0.c 파일의 내용과 같이 수정한다. 이상 작성한 내용을 컴파일하여 확인해 보면 다음과 같은 결과를 볼 수 있다.

```
jump to task0
sum in task1: 0x00100000
count in task1: 0x0008fe35
sum in task0: 0x00100000
count in task0: 0x000701cb
```

여기서 SUM 값은 0x100000으로 나온다. 또 task0 루틴의 count 값과 task1 루틴의 count 값을 더할 경우에도

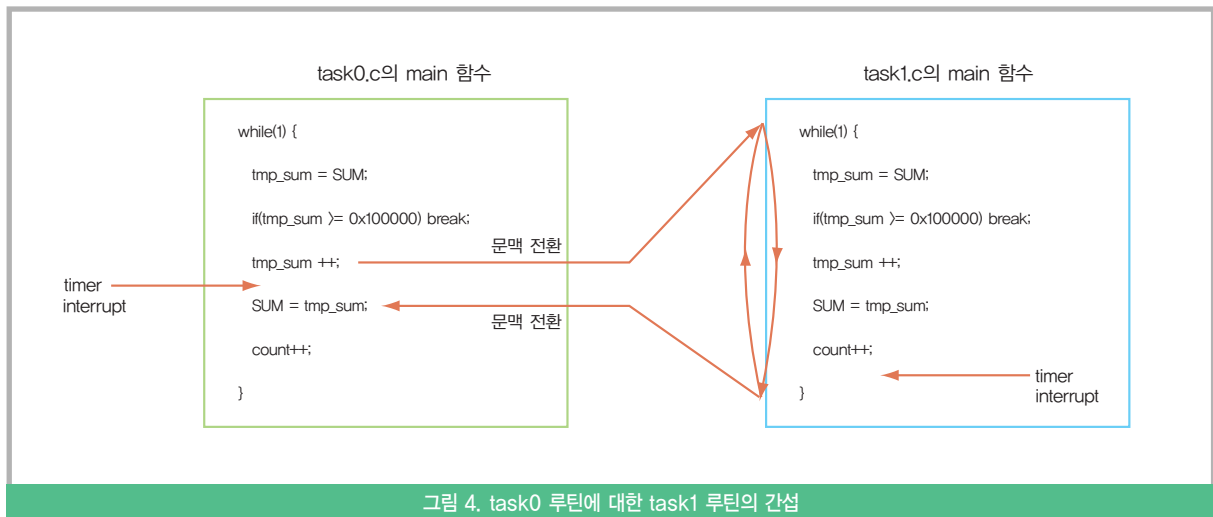


그림 4. task0 루틴에 대한 task1 루틴의 간섭

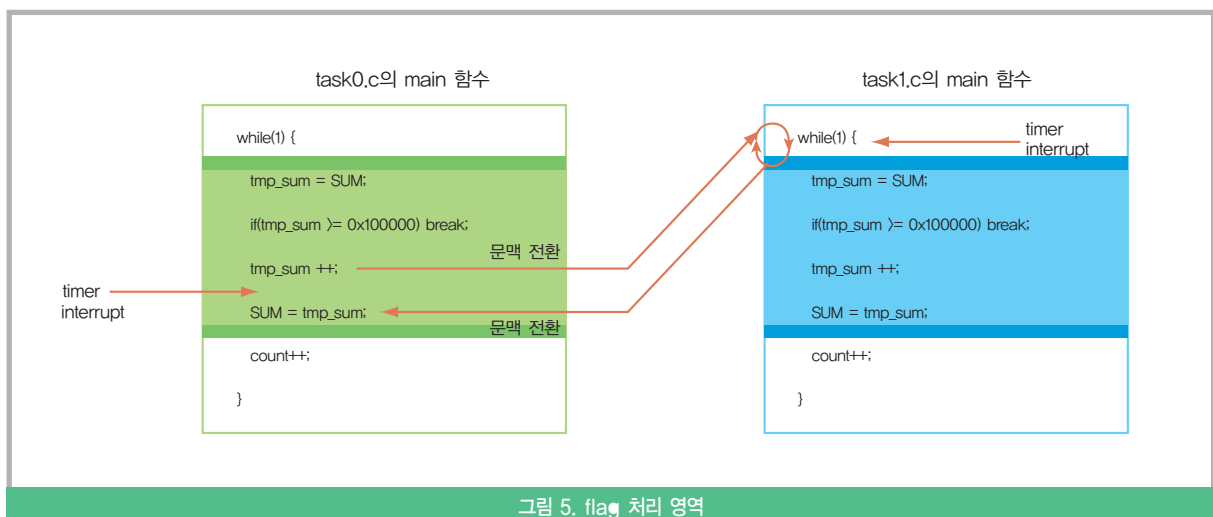


그림 5. flag 처리 영역

0x100000이 나오는 걸 확인할 수 있다. 즉, flag를 사용함으로써 두 루틴이 SUM 변수 값을 차례대로 접근하도록 하여 공유 영역에 대한 경쟁 상태를 해결하였다.

이상에서 태스크 루틴과 태스크 루틴간에 공유 영역에 대한 경쟁 상태가 발생할 경우에 대한 해결책을 알아보았다.

flag를 내리는 동작에서 flag가 이미 내려가 있을 경우에는 busy waiting을 수행하며 기다린다고 하였다. 이렇게 처리하여도 논리적인 문제는 발생하지 않는다. 그러나 태스크가 busy waiting을 수행하는 동안에는 CPU 시간을 낭비하게 되며, 따라서 이에 대한 적절한 처리를 해 주어야 한다. 일반적으로 태스크 루틴에서 예측할 수 없는 시간 동안 busy waiting을 수행해야 할 경우 현재 태스크를 blocking 시킨다. 즉, 현재 프로세스를 특정한 wait queue에 넣고 태스크 스케줄링을 수행한다. 리눅스 커널에서는 sleep_on 류의 함수가 이러한 역할을 한다. 따라서 flag를 내리는 동작은 다음과 같은 형태로 수정되어야 한다.

```
while(1) {
    local_irq_disable();
    if(SUM_LOCK == 1) {
        SUM_LOCK = 0;
        local_irq_enable();
        break;
    }
    local_irq_enable();
    SLEEP_ON();
}
```

즉, flag가 이미 내려가 있을 경우에는 현재 태스크를 blocking 시킨다. 세마포어의 down 동작이나 뮤텍스의 lock 동작도 이와 같은 형태로 구현된다.

또 flag를 올리는 동작은 다음과 같은 형태로 수정되어야 한다.

```
local_irq_disable();
SUM_LOCK = 1;
local_irq_enable();
WAKE_UP();
```

즉, flag를 올리고 나서 blocking 되어 있는 태스크가 있을 경우 그 태스크를 unblocking 시킨다. 세마포어의 up 동작이나 뮤텍스의 unlock 동작도 이와 같은 형태로 구현된다.

SLEEP_ON 함수와 WAKE_UP 함수는 wait queue나 ready queue등에 대한 설계 및 구현을 먼저 한 후에 구현하기로 한다.

다음은 ready queue와 wait queue를 구현하기 전에 먼저 여러 개의 태스크가 수행될 수 있는 환경을 구성해 보자. 여기서는 16개의 태스크가 수행될 수 있도록 하기로 한다.

먼저 main.c 파일의 내용을 다음과 같이 수정한다.

```
main.c
...
#define TASK0L2PT (unsigned int *) (0x100000+0x4400)
#define TASK1L2PT (unsigned int *) (0x100000+0x4800)
#define SHARED_L2PT (unsigned int *) (0x100000+0x4c00)
#define TASK_L2PT(n) (unsigned int *) (0x100000+0x4800+n*0x400)

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

```
#define MASTERL1PT      (unsigned      int
*(0x100000)
...
#define jump2task0      \
({                        \
    unsigned long spsr;   \
    unsigned long task0addr=0x400000; \
    unsigned long svcsp=0x100000; \
    __asm__ __volatile__ ( \
        "sub sp,%2,#4\n"   \
        "mov r0,#0\n"     \
        "mrs %0,spsr\n"   \
        "bic %0,%0,#0xff\n" \
        "orr %0,%0,#0x5f\n" \
        "msr spsr_c,%0\n"  \
        "movs pc,%l"       \
        ::"r"(spsr),"r"(task0addr) \
        ,"r"(svcsp)       \
        : "memory", "cc"); \
    })

void kmain(void)
{
    ...
    mmu_fill_l2pt(TASK0L2PT, TASK0L2PT+8,
TASK0BASE);
    mmu_fill_l2pt(TASK1L2PT, TASK1L2PT+8,
TASK1BASE);
    mmu_fill_l2pt(SHAREDL2PT, SHAREDL2PT+8,
SHAREDBASE);

    {
        int i;
        for(i=2;i<16;i++) {
```

```
        mmu_fill_l2pt(TASKL2PT(i), TASKL2PT(i)+6,
TASKBASE(i));
        mmu_fill_l2pt(TASKL2PT(i)+6,TASKL2PT(i)+8,
TASKBASE(i)+0x4000+i*0x2000);
    }

    flush_cache_tlb();
    ...
}
```

TASKL2PT(n) 매크로는 n번 태스크를 관리하는 level 2 page table의 가상 주소를 나타낸다. n은 2~15의 값을 갖는다고 가정한다. TASKBASE(n) 매크로는 n번 태스크의 물리 주소와 AP, C, B 속성 등을 나타낸다. 참고로 n번 태스크는 1번 태스크가 수행하는 루틴을 수행하게 된다. 즉, 0번 태스크는 task0.c에 있는 main 루틴을 수행하고 1번 태스크부터 15번 태스크는 task1.c에 있는 main 루틴을 수행한다.

jump2task0 매크로에는 0번 태스크에 인자 값을 넘겨주기 위해 r0 레지스터에 0값을 넣어 주는 부분을 추가하였다. 참고로 뒤에서 1번 태스크에는 1값을, 2번 태스크는 2값을, n번 태스크는 n값을 인자로 받게 task.c 파일을 수정하는 부분이 있다. n은 2~15의 값을 갖는다고 가정한다.

kmain 함수에서는 mmu_fill_l2pt 함수를 이용해 task2~15의 level2 page table을 초기화한다. kmain 함수를 수행하고 난 후에 페이지 테이블과 task 2~15의 스택은 물리 메모리 공간상에서 [그림 6]과 같이 배치된다.

다음은 task.c 파일의 내용을 아래와 같이 수정하자.

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

```

process_state process[16];
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] = 0x5f;
    process[1].context[0+2] = 1;

    for(int i;
        for(i=2;i<16;i++) {
            process[i].context[13+2] = 0x408000;
            process[i].context[1] = 0x400000;
            process[i].context[0] = 0x5f;
            process[i].context[0+2] = i;
        }
    }

#define MASTERL1PT (unsigned int *) (0x100000)
#define TASK0L2PTBASE (0x30004400|0x03<<5|1<<4|0x1)
#define TASK1L2PTBASE (0x30004800|0x03<<5|1<<4|0x1)
#define TASKL2PTBASE(n) ((0x30004800+(n*0x400))|0x03<<5|1<<4|0x1)

void schedule()
{
    pid = (pid+1)%16;
    current = &process[pid];

```

```

next = current;

flush_cache_tlb();
if(pid == 0) mmu_map_l2pt(MASTERL1PT+4,
TASK0L2PTBASE);
else if(pid == 1) mmu_map_l2pt(MASTERL1PT
+4, TASK1L2PTBASE);
else mmu_map_l2pt(MASTERL1PT+4, TAS
KL2PTBASE(pid));
}

```

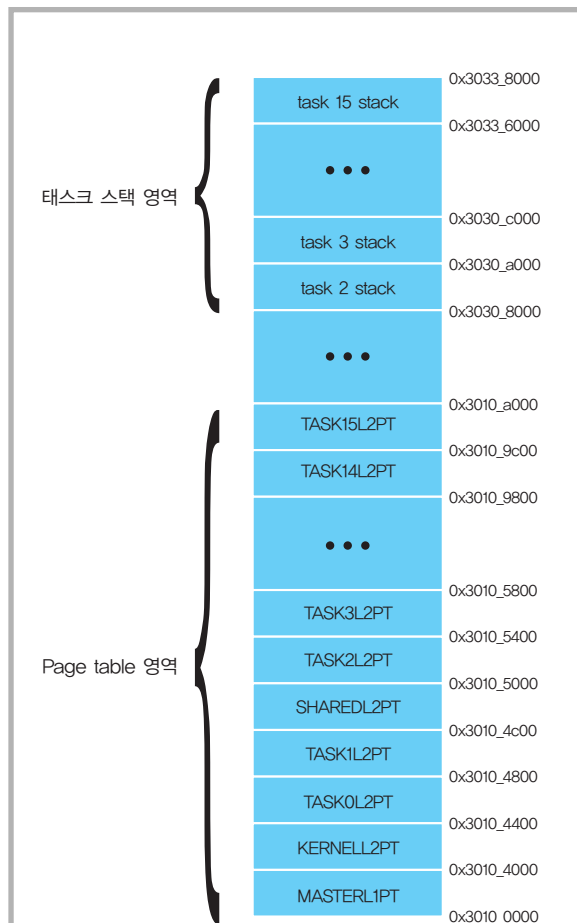


그림 6. 페이지 테이블과 스택의 물리적인 배치

먼저 16개의 태스크를 관리할 수 있도록 process 배열 변수의 엔트리 갯수를 16으로 수정한다. init_task 함수에서는 1번 태스크가 1 값을 인자로 받을 수 있는 루틴을 추가한다. 그리고 task2~task15의 context 변수도 초기화한다. TASKL2PTBASE(n) 매크로는 n 번 태스크를 관리하는 level 2 page table의 물리 주소와 Domain, C, B 속성 등을 나타낸다. schedule 함수에서는 pid 변수를 갱신하는 부분과 다음에 수행할 태스크의 level 2 page table을 맵핑시키는 부분을 수정한다.

다음은 task0.c 파일의 내용을 아래와 같이 수정한다.

```
#include <system.h>

int main(unsigned int arg)
{
    for(;;) {
        local_irq_disable();
        uart_puts("task ");
        uart_puthexnl(arg);
        local_irq_enable();
    }
}
```

main 함수에서 인자 값을 받도록 수정하였다. main 함수는 단순히 인자 값을 찍는 동작만 수행 하도록 하였다.

task1.c 파일의 내용도 task0.c 파일의 내용과 같이 수정한다. 이상 작성한 내용을 컴파일하여 확인해 보면 다음과 같은 수행 결과를 볼 수 있다.

```
...
task 0x00000000
task 0x00000001
task 0x00000002
```

```
task 0x00000003
task 0x00000004
task 0x00000005
task 0x00000006
task 0x00000007
task 0x00000008
task 0x00000009
task 0x0000000a
task 0x0000000b
task 0x0000000c
task 0x0000000d
task 0x0000000e
task 0x0000000f
...
```

16개의 태스크가 순서대로 수행되고 있는 걸 볼 수 있다.

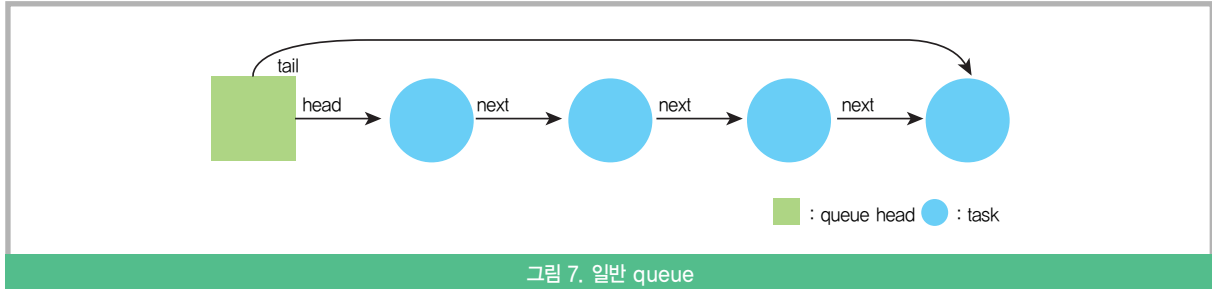
다음은 라운드 로빈 방식의 스케줄링과 우선 순위 기반 방식의 스케줄링 루틴을 추가하기 위해 일반 큐와 우선 순위 큐를 구현해 보기로 하자. 이 두 가지 큐를 이용하여 준비 큐(ready queue)와 대기 큐(wait queue)를 구현할 것이다.

먼저 일반 큐는 그림 7과 같은 모양을 갖는다. 큐 구조체를 include 디렉토리 내에 queue.h 파일에 다음과 같이 정의한다.

```
include/queue.h

typedef struct _queue {
    process_state * head;
    process_state * tail;
    unsigned int num;
} queue;

void initqueue(queue * q);
void enqueue(queue * q, process_state * proc);
process_state * dequeue(queue * q);
```



queue 구조체 내의 head 변수는 맨 처음에 큐에 들어온 태스크를 가리키며, tail 변수는 가장 최근에 큐에 들어온 태스크를 가리킨다. num 변수는 큐 내에 있는 태스크의 개수를 나타낸다.

각각의 태스크는 다음 태스크를 가리키는 next 포인터 변수를 가진다. include 디렉토리에 있는 task.h 파일의 process_state 구조체를 다음과 같이 수정한다.

```
include/task.h
typedef struct _process_state {
    unsigned int context[18];
    unsigned int pid;
    unsigned int time_remain;
    unsigned int time_slice;
    unsigned int need_resched;
    unsigned int prior;
    struct _process_state * next;
} process_state;
```

pid 변수는 태스크 번호를 나타내며 각각의 태스크는 고유한 번호를 갖는다. time_remain 변수는 태스크가 사용할 수 있는 time slice의 남은 개수를 가지며, 타이머 인터럽트가 발생할 때 마다 하나씩 감소한다. time_slice 변수는 태스크가 새로이 time slice를 할당 받을 때 할당 받을 수 있는 time slice의 개수를 가진다. need_resched 변수는 태스크 스케줄링이 필요할 경우에 사용하며, prior 변수는 태스크의 우선순위를 나타낸다.

next 포인터 변수는 준비 큐나 대기큐 내에서 다음 태스크를 가리키는 역할을 한다.

initqueue, enqueue, dequeue 함수는 queue.c 파일에 다음과 같이 정의되어 있으며, 각각 큐를 초기화하고 큐에 태스크를 넣거나 빼는 역할을 한다. 여기서는 큐에 대한 자세한 설명을 하지 않기로 한다.

```
queue.c
#include <task.h>
#include <queue.h>
#include <null.h>

void initqueue(queue * q)
{
    q->head = NULL;
    q->tail = NULL;
    q->num = 0;
}

void enqueue(queue * q, process_state * proc)
{
    if(q->num == 0) {
        q->head = proc;
        q->tail = proc;
    } else {
        q->tail->next = proc;
```

```

        q->tail = proc;
    }
    q->num ++;
}

process_state * dequeue(queue * q)
{
    process_state * tmp;

    if(q->num == 0) return NULL;
    else if(q->num == 1) {
        tmp = q->head;
        q->head = NULL;
        q->tail = NULL;

    } else {
        tmp = q->head;
        q->head = q->head->next;
    }

    q->num --;

    return tmp;
}

```

NULL 매크로는 include 디렉토리의 null.h 파일에 다음과 같이 정의하였다.

```

include/null.h
#define NULL ((void *)0)

```

다음으로 우선 순위 큐는 그림 8과 같은 모양을 갖는다. 큐 구조체를 include 디렉토리 내에 prior-queue.h 파일에 다음과 같이 정의한다.

```

include/prior-queue.h
#include <queue.h>

typedef struct _priorQ {
    queue Q[16];
    unsigned int num;
} priorQ;

void initpriorQ(priorQ *);
void enpriorQ(priorQ *, process_state *);
process_state * depriorQ(priorQ *);

```

priorQ 구조체는 총 16개의 일반 큐를 가지며 우선 순위 큐 내에 있는 모든 태스크의 개수를 저장하는 num 변수가 있다. Q 배열 변수의 인덱스 번호가 작을수록 우선 순위가 높다.

initpriorQ, enpriorQ, depriorQ 함수는 prior-queue.c 파일에 다음과 같이 정의되어 있다.

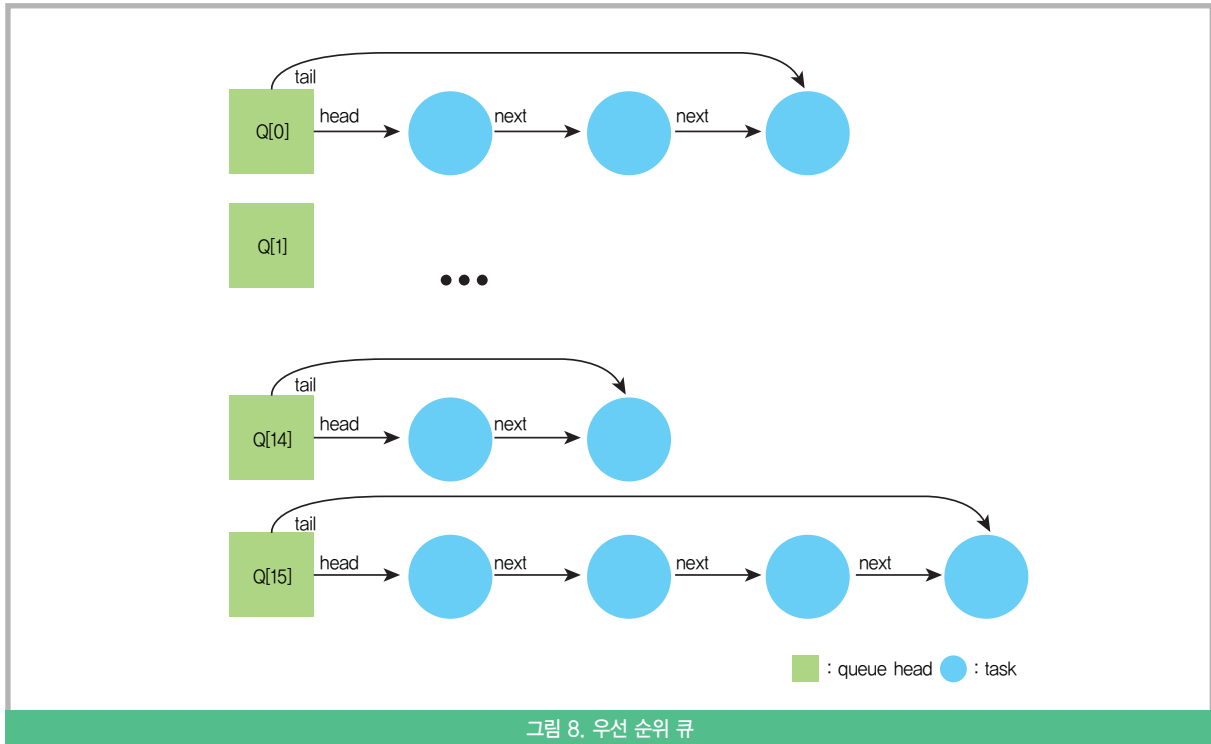
```

prior-queue.c
#include <task.h>
#include <prior-queue.h>
#include <null.h>

void initpriorQ(priorQ * ppQ)
{
    int i;
    for(i=0;i<16;i++) initqueue(&ppQ->Q[i]);
    ppQ->num = 0;
}

void enpriorQ(priorQ * ppQ, process_state * proc)
{
    enqueue(&ppQ->Q[proc->prior], proc);
}

```



```

        ppQ->num ++;
    }

    process_state * depriorQ(priorQ * ppQ)
    {
        if(ppQ->num == 0) return NULL;
        else {
            int i;
            process_state * tmp;

            for(i=0; i<16; i++) {
                tmp = dequeue(&ppQ->Q[i]);
                if(tmp != NULL) break;
            }
            ppQ->num --;
            tmp->next = NULL;

            return tmp;
        }
    }

```

```

    }
}

```

initpriorQ 함수는 우선 순위 큐를 초기화하는 역할을 하며, enpriorQ 함수는 넣고자 하는 태스크의 우선 순위해당하는 큐에 태스크를 넣는 역할을 하며, depriorQ 함수는 우선 순위가 가장 높은 태스크를 우선 순위 큐에서 꺼내는 역할을 한다.

이상에서 일반 큐와 우선 순위 큐를 구현해 보았다.

지금까지 우리는 태스크 루틴과 하드웨어 인터럽트 처리 루틴간에 또 태스크 루틴과 태스크 루틴간에 공유 영역에 대한 경쟁 상태가 발생할 경우에 대한 해결책을 알아보았다. 또 라운드 로빈 방식과 우선 순위 방식의 스케줄링을 구현하기 위해 필요한 일반 큐와 우선 순위 큐를 구현해 보았다.

다음 호에서는 이 두 가지 큐를 이용하여 라운드 로빈 방식과 우선 순위 방식의 스케줄링을 구현해 보기로 하자. 또, SLEEP_ON 함수와 WAKE_UP 함수도 구현해 보기로 하자. R_{time}^{rel}