

설계와 구현을 통한 임베디드 OS의 이해와 응용 ⑩

## IPC의 설계 및 구현 (상)

지난 호(4월)에 우리는 일반 큐와 우선순위 큐를 이용하여 라운드 로빈 방식과 우선순위 방식의 스케줄링을 구현해 보았다. 이번 호에서는 SLEEP\_ON 함수와 WAKE\_UP 함수를 구현해 보기로 하자. 또한 이 함수들을 이용하여 뮤텍스, 세마포어를 구현해 보고, IPC 루틴도 작성해 보기로 하자.

글 : 서민우 선임연구원 / 새롬전자

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

먼저 sleep\_on 함수와 wake\_up 함수를 구현한 후 이 두 함수를 시스템 콜을 통해 접근할 수 있도록 SLEEP\_ON 매크로와 WAKE\_UP 매크로를 구현해 보기로 하자.

다음은 sleep.c 파일의 내용이다.

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

extern process_state * current;
extern priorQ * active;

void sleep_on(priorQ * pwaitq)
{
    enpriorQ(pwaitq, current);
    current->need_resched = 1;
}

void wake_up(priorQ * pwaitq)
```

```
{
    process_state * pps;

    pps = depriorQ(pwaitq);
    if(pps == NULL) return;

    enpriorQ(active, pps);
    if(current->prior < pps->prior)
        current->need_resched = 1;
}
```

sleep\_on 함수에서는 현재 태스크를 대기큐에 넣고 스케줄링을 요청한다. wake\_up 함수에서는 대기큐에서 대기하는 태스크가 있을 경우 꺼내서 active 큐(ready 큐)에 넣는다. 그리고 현재 수행중인 태스크와 대기큐에서 꺼낸 태스크의 우선순위를 비교한 후, 현재 수행중인 태스크의 우선순위가 작을 경우 스케줄링을 요청한다.

다음은 태스크 루틴에서 sleep\_on 함수와 wake\_up 함수를 접근할 수 있도록 SLEEP\_ON 매크로와 WAKE\_UP 매크로를 정의해 보기로 하자.

## \* 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

필 / 자 / 소 / 개

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 32비트 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.

다음은 include 디렉터리 내의 sleep.h 파일의 내용이다.

```
include/sleep.h
#define SLEEP_ON(pwaitq) \
({ \
    unsigned long pwq=(unsigned long)pwaitq; \
    __asm__ __volatile__( \
        "mov r1,%0\n" \
        "swi 0x7f" \
        ::"r"(pwq):"r1"); \
})

#define WAKE_UP(pwaitq) \
({ \
    unsigned long pwq=(unsigned long)pwaitq; \
    __asm__ __volatile__( \
        "mov r1,%0\n" \
        "swi 0x7e" \
        ::"r"(pwq):"r1"); \
})

void sleep_on(priorQ * pwaitq);
void wake_up(priorQ * pwaitq);
```

SLEEP\_ON 매크로에서는 r1 레지스터에 대기큐의 주소 값을 넘겨준 후 <swi 0x7f> 명령어를 수행한다. WAKE\_UP 매크로에서는 r1 레지스터에 대기큐의 주소 값을 넘겨준 후 <swi 0x7e> 명령어를 수행한다.

다음은 swihandler.c 파일의 내용이다.

```
swihandler.c
#include <task.h>
#include <timer-list.h>
#include <prior-queue.h>
#include <sleep.h>

extern process_state * current;
extern timer_list tlist;

extern unsigned int jiffies;

void eventsSWIHandler(unsigned int syscallnum,
unsigned int arg1)
{
    if(syscallnum == 0x80) {
        unsigned int alarm_time = arg1;
```

```

        current->expires = jiffies+alarm_time;
        insert_timer_list(&tlist, current);
        current->need_resched = 1;
    }

    if(syscallnum == 0x7f) { // SLEEP_ON
        priorQ * pwaitq = (priorQ *)arg1;
        sleep_on(pwaitq);
    }

    if(syscallnum == 0x7e) { // WAKE_UP
        priorQ * pwaitq = (priorQ *)arg1;
        wake_up(pwaitq);
    }

    if(current->need_resched == 1) {
        current->need_resched = 0;
        schedule();
    }
}

```

swihandler.c 파일에서는 먼저 prior-queue.h, sleep.h 파일을 include 해준다. 그리고 syscallnum 인자를 통해 넘어온 값이 0x7f일 경우 sleep\_on 함수를, 0x7e일 경우 wake\_up 함수를 호출할 수 있도록 해 준다.

다음은 SLEEP\_ON 매크로와 WAKE\_UP 매크로를 이용하여 task1과 task2가 공유 영역을 접근하는 루틴을 작성해 보기로 하자.

다음은 main.c 파일의 내용이다.

```

main.c
...

```

```

#include <task.h>
#include <prior-queue.h>
...
void kmain(void)
{
    ...

    *(volatile unsigned int *)0x200000 = 0;
    *(volatile unsigned int *)0x200004 = 1;
    {
        priorQ * psum_waitq = (priorQ *)0x200008;
        initpriorQ(psum_waitq);
    }

    jump2task00;
}

```

main.c 파일에서는 먼저 task.h, prior-queue.h 파일을 include 해 준다. 그리고 kmain 함수에서는 0x200008 번지에 대기큐를 하나 할당한 후 초기화해 준다.

다음은 task.c 파일의 내용이다.

```

task.c
...
void init_task()
{
    ...

    process[2].context[0+2] = 2;
    process[2].context[13+2] = 0x408000;
    process[2].context[1] = 0x400000;
    process[2].context[0] = 0x5f;
    process[2].pid = 2;
    process[2].time_remain = 10;
}

```

```

process[2].time_slice = 10;
process[2].need_resched = 0;
process[2].prior = 0;
process[2].next = NULL;
process[2].expires = 0;

for(i=3;i<16;i++) {
    ...
}

```

init\_task 함수에서는 process[2]의 우선순위와 타임 슬라이스 값이 process[1]과 같도록 수정한다. 즉, prior 값은 process[1]과 같이 0 값을, time\_remain 값과 time\_slice 값은 10 값을 준다. 이와 같이 하는 이유는 task1과 task2의 우선순위를 같게 함으로써 경쟁 상태를 발생시켜 SLEEP\_ON 매크로와 WAKE\_UP 매크로가 제 역할을 하는지 보기 위해서이다. for 문의 i의 초기값도 2에서 3으로 바꾼다.

다음은 task1.c 파일의 내용을 살펴보자. task1.c 파일의 내용은 3월호 첫 번째 예제에서 보았던 task0.c 파일의 내용과 논리적으로 같다. 3월호에서는 busy waiting을 수행하여 동기화의 문제를 해결하였으나, 여기서는 sleep\_on 함수를 이용하여 대기큐에서 대기하거나, wake\_up 함수를 이용하여 대기큐에서의 대기를 해제해 줌으로써 동기화의 문제를 해결하고자 하였다.

```

#include <system.h>
#include <task.h>
#include <prior-queue.h>
#include <sleep.h>

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

```

```

#define SUM_LOCK *(volatile unsigned int *)0x200004

unsigned int tmp_sum;
unsigned int count = 0;
priorQ * psum_waitq = (priorQ *)0x200008;

if(arg>2) for(;;);

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

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

    tmp_sum++;
    SUM = tmp_sum;

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

```

```

        WAKE_UP(psum_waitq);

        count++;
    }

    local_irq_disable();
    uart_puts("sum in task");
    (arg==1)?uart_puts("1"):uart_puts("2");
    uart_puts(": ");
    uart_puthexnl(SUM);
    uart_puts("count in task");
    (arg==1)?uart_puts("1"):uart_puts("2");
    uart_puts(": ");
    uart_puthexnl(count);
    local_irq_enable();

    for(;;);
}

```

```

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

local_irq_enable();
SLEEP_ON(psum_waitq);
}

루틴 2:
local_irq_disable();
SUM_LOCK = 1;
local_irq_enable();
WAKE_UP(psum_waitq);

```

즉, 이 두 루틴은 본래 다음과 같은 순서로 수행되어야 논리적인 문제가 없다.

main 함수에서는 arg 인자를 체크하여 task1과 task2만 그 다음 루틴을 수행하도록 하였다. task1과 task2는 SUM 변수를 접근하기 위해 먼저 SUM\_LOCK 변수의 값을 체크한다. SUM\_LOCK 변수의 값이 1일 경우는 SUM\_LOCK 변수의 값을 0으로 만들고 SUM 변수를 접근한다. SUM\_LOCK 변수의 값이 0일 경우는 대기큐에서 대기한다. SUM 변수에 대한 접근을 마친 태스크는 SUM\_LOCK의 값을 1로 돌려놓고 대기큐에서 대기하는 태스크가 있을 경우 깨운다. 그런데 이 소스 코드는 논리적으로 문제가 있다. 즉, 다음 두 루틴 간에 대기큐를 두고 동기화 문제가 발생할 수 있다.

루틴 1:

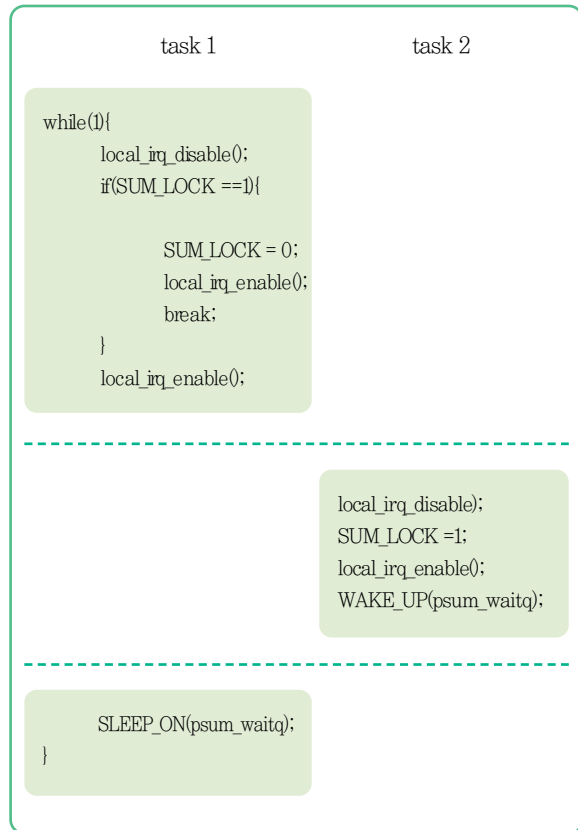
```

while(1) {
    local_irq_disable();

```

task 1	task 2
<pre> while(1){     local_irq_disable();     if(SUM_LOCK == 1){         SUM_LOCK = 0;         local_irq_enable();         break;     }     local_irq_enable();     SLEEP_ON(psum_waitq); } </pre>	<pre> local_irq_disable(); SUM_LOCK = 1; local_irq_enable(); WAKE_UP(psum_waitq); </pre>

그러나 이 두 루틴의 순서는 다음과 같이 수행될 수도 있으며, 이 경우 논리적으로 문제가 발생한다.



이와 같이 두 태스크의 루틴이 수행될 경우 상황에 따라 dead lock 내지는 starvation이 발생할 수 있다. 따라서 우리는 루틴 1과 루틴 2를 각각 다음과 같이 수정해야 한다.

루틴 1:

```

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

```

```

}
SLEEP_ON(psum_waitq);
local_irq_enable();
}

```

루틴 2:

```

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

```

이상에서 task1.c 파일의 내용을 다음과 같이 수정한다.

```

task1.c
...
int main(unsigned int arg)
{
    ...
    while(1) {
        while(1) {
            local_irq_disable();
            if(SUM_LOCK == 1) {
                SUM_LOCK = 0;
                local_irq_enable();
                break;
            }
            SLEEP_ON(psum_waitq);
            local_irq_enable();
        }

        tmp_sum = SUM;
        if(tmp_sum >= 0x100000) {

```

```

        local_irq_disable();
        SUM_LOCK = 1;
        WAKE_UP(psum_waitq);
        local_irq_enable();
        break;
    }

    tmp_sum++;
    SUM = tmp_sum;

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

    count++;
}
...
}

```

참고로 task0.c 파일의 내용은 다음과 같다.

마지막으로 Makefile의 KERNEL\_OBJ 변수에 다음과 같이 sleep.o 파일을 추가한다.

```

task0.c
#include <system.h>

int main(unsigned int arg)
{
    for(;;) ;
}

```

KERNEL\_OBJ = head.o main.o mmusetup.o uart.o  
exceptions.o swihandler.o timer.o \  
irqhandler.o mmu.o task.o queue.o prior-queue.o  
timer-list.o sleep.o

이상 작성한 내용을 컴파일하여 확인해 보면 다음과 같은 결과를 볼 수 있다.

```

jump to task0
sum in task2: 0x00100000
count in task2: 0x0007f8df
sum in task1: 0x00100000
count in task1: 0x00080721

```

task1의 count 값과 task2의 count 값의 합이 sum 값과 같음을 볼 수 있다.

이상에서 sleep\_on 함수와 wake\_up 함수를 구현해 보고, SLEEP\_ON 매크로와 WAKE\_UP 매크로를 이용하여 task1과 task2가 공유 영역을 접근하는 루틴을 작성해 보았다. Real Time

편집자 주 \_\_\_\_\_  
지면 관계상 'IPC의 설계 및 구현' 기사는  
상, 하로 나누어 연재합니다.

