

설계와 구현을 통한 임베디드 OS의 이해와 응용 ⑩

IPC의 설계 및 구현(하)

지난 호(4월)에 우리는 일반 큐와 우선순위 큐를 이용하여 라운드 로빈 방식과 우선순위 방식의 스케줄링을 구현해보았다. 이번 호에서는 SLEEP_ON 함수와 WAKE_UP 함수를 구현 해보기로 하자. 또한 이 함수들을 이용하여 뮤텝스, 세마포어를 구현해 보고, IPC 루틴도 작성해 보기로 하자.

글 : 서민우 선임연구원 / 새롬전자

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

지난달에 이어 이번호에서는 뮤텝스를 구현해 보기로 하자. 먼저 include 디렉터리 내의 sleep.h 파일에 다음의 내용을 추가하자.

```
include/sleep.h
...
typedef unsigned int mutex_t;
int mutex_lock(mutex_t * lock, priorQ * pwaitq);
void mutex_unlock(mutex_t * lock, priorQ * pwaitq);
```

즉, mutex_t의 변수 타입을 정의해 주고, mutex_lock 함수와 mutex_unlock 함수의 프로토타입을 선언해 준다.

다음은 mutex.c 파일의 내용이다.

```
mutex.c
#include <task.h>
#include <prior-queue.h>
#include <system.h>
```

```
#include <sleep.h>

int mutex_lock(mutex_t * lock, priorQ * pwaitq)
{
    local_irq_disable();
    if((*lock) == 1) {
        (*lock) = 0;
        local_irq_enable();
        return 1;
    }
    SLEEP_ON(pwaitq);
    local_irq_enable();

    return 0;
}

void mutex_unlock(mutex_t * lock, priorQ * pwaitq)
{
    local_irq_disable();
    (*lock) = 1;
```



서민우(mwseo@e-serome.co.kr)

* 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

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.

```
WAKE_UP(pwaitq);
local_irq_enable();
}
```

mutex_lock 함수는 인터럽트를 막은 후 lock 포인터 변수가 가리키는 변수의 값이 1인지 확인하고 1이면 0으로 만든 후 인터럽트를 열고 1 값을 리턴한다. 그렇지 않을 경우에는 대기큐에서 대기한 후 깨어나 인터럽트를 열고 0 값을 리턴한다. mutex_unlock 함수는 인터럽트를 막은 후 lock 포인터 변수가 가리키는 변수의 값을 1로 만든 후 대기큐에서 대기하는 태스크가 있을 경우 태스크를 깨우고 인터럽트를 연다.

다음은 뮤텍스를 이용하여 동기화의 문제를 해결하는 예제로써, 이전 예제와 논리적으로 내용이 같다.

```
task1.c
...
int main(unsigned int arg)
{
#define SUM *(volatile unsigned int *)0x200000
    unsigned int tmp_sum;
    unsigned int count = 0;
```

```
priorQ * psum_waitq = (priorQ *)0x200008;
mutex_t * sum_lock = (mutex_t *)0x200004;

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

while(1) {
    while(1) {if(mutex_lock(sum_lock, psum_waitq))
break;}

    tmp_sum = SUM;
    if(tmp_sum >= 0x100000) {
        mutex_unlock(sum_lock, psum_waitq);
        break;
    }

    tmp_sum++;
    SUM = tmp_sum;

    mutex_unlock(sum_lock, psum_waitq);

    count++;
}
```

```
...
    for(;;);
}
```

마지막으로 Makefile의 TASK0_OBJ, TASK1_OBJ 변수에 다음과 같이 mutex.o 파일을 추가한다.

```
TASK0_OBJ = task0.o uart.o mutex.o
TASK1_OBJ = task1.o uart.o mutex.o
```

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

```
jump to task0
sum in task1: 0x00100000
count in task1: 0x0007613d
sum in task2: 0x00100000
count in task2: 0x00089ec3
```

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

이상에서 뮤텍스를 구현해 보았다. 다음은 세마포어를 구현해 보기로 하자.

먼저 include 디렉토리 내의 sleep.h 파일에 다음의 내용을 추가하자.

```
include/sleep.h
...
typedef unsigned int semaphore_t;
int sem_wait(semaphore_t * sema, priorQ * pwaitq);
void sem_post(semaphore_t * sema, priorQ * pwaitq);
```

즉, semaphore_t의 변수 타입을 정의해 주고, sem_wait 함수와 sem_post 함수의 프로토타입을 선언해 준다.

다음은 semaphore.c 파일의 내용이다.

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

int sem_wait(semaphore_t * sema, priorQ * pwaitq)
{
    local_irq_disable();
    if((*sema) > 0) {
        (*sema)--;
        local_irq_enable();
        return 1;
    }
    SLEEP_ON(pwaitq);
    local_irq_enable();

    return 0;
}

void sem_post(semaphore_t * sema, priorQ * pwaitq)
{
    local_irq_disable();
    (*sema)++;
    WAKE_UP(pwaitq);
    local_irq_enable();
}
```

sem_wait 함수는 인터럽트를 막은 후 sema 포인터 변수가 가리키는 변수의 값이 0보다 큰지 확인하고 0보다 크면 값을 1

감소시키고, 인터럽트를 열고 1 값을 리턴한다. 그렇지 않을 경우에는 대기큐에서 대기한 후 깨어나 인터럽트를 열고 0 값을 리턴한다. sem_post 함수는 인터럽트를 막은 후 sema 포인터 변수가 가리키는 변수의 값을 1 증가시킨 후 대기큐에서 대기하는 태스크가 있을 경우 태스크를 깨우고 인터럽트를 연다.

다음은 세마포어를 이용하여 동기화의 문제를 해결하는 예제로서, 이전 예제와 논리적으로 내용이 같다.

```
task1.c
...
int main(unsigned int arg)
{
#define SUM *(volatile unsigned int *)0x200000
    unsigned int tmp_sum;
    unsigned int count = 0;
    priorQ * psum_waitq = (priorQ *)0x200008;
    semaphore_t * sum_sema = (semaphore_t *)0x200004;

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

    while(1) {
        while(1) if(sem_wait(sum_sema, psum_waitq))
            break;}

        tmp_sum = SUM;
        if(tmp_sum >= 0x100000) {
            sem_post(sum_sema, psum_waitq);
            break;
        }

        tmp_sum++;
        SUM = tmp_sum;

        sem_post(sum_sema, psum_waitq);
```

```
count++;
}
...
}
```

마지막으로 Makefile의 TASK0_OBJ, TASK1_OBJ 변수에 다음과 같이 semaphore.o 파일을 추가한다.

```
TASK0_OBJ = task0.o uart.o mutex.o semaphore.o
TASK1_OBJ = task1.o uart.o mutex.o semaphore.o
```

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

```
jump to task0
sum in task1: 0x00100000
count in task1: 0x0008696a
sum in task2: 0x00100000
count in task2: 0x00079696
```

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

이상에서 세마포어를 구현해 보았다. 참고로 세마포어 변수의 초기값이 1일 경우 binary semaphore라고 하며, 1보다 클 경우 counting semaphore라고 한다. 일반적으로 binary semaphore의 경우 공통의 영역이 오직 하나일 경우 사용하며, counting semaphore의 경우 같은 종류의 공통의 영역이 하나 이상일 경우 사용한다.

마지막으로 인터럽트를 막는 방법과 세마포어를 이용해서 하드웨어 인터럽트 루틴에서 task1 루틴으로 데이터를 주는 루틴을 구현해 보기로 하자. 또 task1 루틴에서 task2 루틴으로 데이터를 주는 루틴을 구현해 보기로 하자. 전자의 경우 하드웨어

인터럽트 루틴이 데이터에 대한 생산자 역할을 하며, task1 루틴이 데이터에 대한 소비자 역할을 한다. 후자의 경우 task1 루틴이 데이터에 대한 생산자 역할을 하며, task2 루틴이 데이터에 대한 소비자 역할을 한다. 이와 같이 하드웨어 인터럽트 루틴에서 태스크 루틴으로, 또 태스크 루틴에서 또 다른 태스크 루틴으로 데이터를 주고받는 동작을 IPC라고 한다.

다음은 main.c 파일의 내용이다.

```
main.c
...
#include <sleep.h>
...
void kmain(void)
{
    ...
    init_task();

    uart_puts("jump to task0\n");

    *(volatile int *)0x200000 = 0;    // buffer
    *(volatile semaphore_t *)0x200004 = 1; // number
of empty buffer
    *(volatile semaphore_t *)0x200008 = 0; // number
of full buffer
    {
        priorQ * buffer_waitq = (priorQ *)0x20000c;
        initpriorQ(buffer_waitq);
    }

    jump2task0();
}
```

main.c 파일에서는 먼저 sleep.h 파일을 include해 준다. kmain 함수에서는 하드웨어 인터럽트 루틴과 task1 루틴간의

IPC를 위해, 데이터를 저장할 버퍼 변수를 0x200000 번지에, 빈 버퍼의 개수를 관리할 변수를 0x200004 번지에, 찬 버퍼의 개수를 관리할 변수를 0x200008 번지에, 대기큐를 0x20000c 번지로 정한다. 그리고 데이터를 저장할 버퍼 변수의 값을 0으로, 빈 버퍼의 개수를 관리할 변수의 값을 1로, 찬 버퍼의 개수를 관리할 변수의 값을 0으로 초기화하고, 대기큐도 초기화한다. 여기서는 편의상 데이터를 저장할 버퍼의 개수를 하나만 두기로 한다.

다음은 irqhandler.c 파일의 내용이다.

```
irqhandler.c
...
#include <sleep.h>
...
void eventsIRQHandler()
{
    ...

    jiffies++;
    current->time_remain--;
    if(current->time_remain <= 0) {
        current->time_remain = current->time_slice;
        current->need_resched = 1;
    }

    // produce data
    {
        int * pbuf = (int *)0x200000; // buffer
        semaphore_t * pnum_empty_buf =
(semaphore_t *)0x200004;
        semaphore_t * pnum_full_buf = (semaphore_t
*)0x200008;

        priorQ * pbuffer_waitq = (priorQ *)0x20000c;
        static int input = 0;
```

```

if((*pnum_empty_buf) > 0) {
    (*pnum_empty_buf)--;

    (*pbuf) = input++;

    (*pnum_full_buf)++;
    wake_up(pbuffer_waitq);

    uart_puts("produce: ");
    uart_puthexnl((*pbuf));
}
}

rSRCPND |= rintpnd;
rINTPND |= rintpnd;
...
}

```

irqhandler.c 파일에서는 먼저 sleep.h 파일을 include해 준다. eventsIRQHandler 함수에는 데이터를 공급하는 루틴을 추가한다. 데이터를 공급하는 루틴에서는 먼저 빈 버퍼가 있는지 확인하고 빈 버퍼가 있으면 빈 버퍼의 개수를 하나 감소시키고 버퍼에 값을 채운다. 그리고 찬 버퍼의 개수를 하나 증가시키고 대기큐에서 대기하고 있는 태스크가 있을 경우 태스크를 깨운다. 참고로 C에서 다음 두 연산은 서로 다름에 주의하라.

```

(*pnum_empty_buf)--;

*pnum_empty_buf--;

```

다음은 task1.c 파일의 내용이다.

task1.c

```

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

int main(unsigned int arg)
{
    int * pbuf = (int *)0x200000;
    semaphore_t * pnum_empty_buf = (semaphore_t *)0x200004;
    semaphore_t * pnum_full_buf = (semaphore_t *)0x200008;
    priorQ * pbuffer_waitq = (priorQ *)0x20000c;
    int i;

    if(arg>1) for(;;);

    for(i=0;i<0x1000;i++) {
        int output;

        // consume data
        while(1) {if(sem_wait(pnum_full_buf,
pbuffer_waitq)) break;}

        output = (*pbuf);

        local_irq_disable();
        (*pnum_empty_buf)++;
        uart_puts("consume: ");
        uart_puthexnl(output);
        local_irq_enable();
    }

    for(;;) ;
}

```

task1은 대기큐에서 버퍼가 차기를 기다린다. 버퍼가 차면 데이터를 꺼낸 후 빈 버퍼의 개수를 하나 증가시킨다. 이 동작을 0×1000번 반복한다.

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

```
jump to task0
produce: 0x00000000
consume: 0x00000000
produce: 0x00000001
consume: 0x00000001
produce: 0x00000002
consume: 0x00000002
produce: 0x00000003
consume: 0x00000003
...
produce: 0x00000ffc
consume: 0x00000ffc
produce: 0x00000ffd
consume: 0x00000ffd
produce: 0x00000ffe
consume: 0x00000ffe
produce: 0x00000fff
consume: 0x00000fff
produce: 0x00001000
```

하드웨어 인터럽트 루틴에서 데이터를 생산하고, task1 루틴에서 데이터를 소비하는 상황을 볼 수 있다.

다음은 task1 루틴에서 task2 루틴으로 데이터를 주는 루틴을 구현해 보기로 하자.

먼저 디버깅의 편의상 irqhandler.c에서 다음 부분을 제거하자.

```
uart_puts("produce: ");
uart_puthexnl((*pbuf));
```

다른 부분은 그대로 둔다.

다음은 main.c 파일의 내용이다.

```
main.c
...
void kmain(void)
{
    ...
    uart_puts("jump to task0\n");

    *(volatile int *)0x200000 = 0; // buffer
    *(volatile semaphore_t *)0x200004 = 1; // number
of empty buffer
    *(volatile semaphore_t *)0x200008 = 0; // number
of full buffer
    {
        priorQ * buffer_waitq = (priorQ *)0x20000c;
        initpriorQ(buffer_waitq);
    }

    *(volatile int *)0x20000c+sizeof(priorQ) = 0; // buffer 2
    // number of empty buffer 2
    *(volatile semaphore_t *)0x200010+sizeof(priorQ) = 1;
    // number of full buffer 2
    *(volatile semaphore_t *)0x200014+sizeof(priorQ) = 0;
    {
        priorQ * pempty_buffer_waitq = (priorQ *)0x200018+sizeof(priorQ);
        priorQ * pfull_buffer_waitq = (priorQ *)0x200018+2*sizeof(priorQ);
```

```

    initpriorQ(pempty_buffer_waitq);
    initpriorQ(pfull_buffer_waitq);
}

jump2task00;
}

```

kmain 함수에서는 task1과 task2간의 데이터를 주고받기 위하여 버퍼를 하나 두고, 빈 버퍼의 개수를 관리하는 변수, 찬 버퍼의 개수를 관리하는 변수, 그리고 빈 버퍼용 대기큐, 찬 버퍼용 대기큐를 하나씩 두었다. 그리고 버퍼의 값은 0으로, 빈 버퍼의 개수를 관리하는 변수를 1로, 찬 버퍼의 개수를 관리하는 변수를 0으로 초기화하였다. 빈 버퍼용 대기큐와 찬 버퍼용 대기큐도 초기화하였다.

다음은 task1.c 파일의 내용이다.

```

task1.c
...
int main(unsigned int arg)
{
    int * pbuf = (int *)0x200000;
    semaphore_t * pnum_empty_buf = (semaphore_t *)
0x200004;
    semaphore_t * pnum_full_buf = (semaphore_t *)
0x200008;
    priorQ * pBuffer_waitq = (priorQ *)0x20000c;

    int * pbuf2 = (int *) (0x20000c+sizeof(priorQ));
    semaphore_t * pnum_empty_buf2 = (semaphore_t *)
(0x200010+sizeof(priorQ));
    semaphore_t * pnum_full_buf2 = (semaphore_t *)
(0x200014+sizeof(priorQ));

```

```

    priorQ * pempty_buffer_waitq = (priorQ *) (0x200018
+sizeof(priorQ));
    priorQ * pfull_buffer_waitq = (priorQ *) (0x200018+
2*sizeof(priorQ));
    int i;
    int output;

    if(arg == 1) {
        for(i=0;i<0x1000;i++) {
            while(1) {if(sem_wait(pnum_full_buf, pBuffer_
waitq)) break;}

            output = (*pbuf);

            local_irq_disable();
            (*pnum_empty_buf)++;
            local_irq_enable();

            while(1) {if(sem_wait(pnum_empty_buf2, pempty
_buffer_waitq)) break;}

            (*pbuf2) = output;

            uart_puts("produce2: ");
            uart_puthexnl(output);

            sem_post(pnum_full_buf2, pfull_buffer_waitq);
        }
    } else if(arg == 2) {
        for(i=0;i<0x1000;i++) {

while(1){if(sem_wait(pnum_full_buf2,pfull_buffer_waitq))
break;}

```



```

        output = *pbuf2;

        uart_puts("consume2: ");
        uart_puthexnl(output);

        sem_post(&num_empty_buf2, pempty_buffer
        _waitq);
    }
}

for(;;) ;
}

```

여기서 task1은 pbuf 포인터 변수가 가리키는 곳으로부터 데이터를 공급받은 후 pbuf2 포인터 변수가 가리키는 곳에 데이터를 공급하는 역할을 한다. task2는 pbuf2 포인터 변수가 가리키는 곳으로부터 데이터를 공급받는다. task1과 task2는 이러한 동작을 0x1000번 반복한다.

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

```

jump to task0
produce2: 0x00000000
consume2: 0x00000000
produce2: 0x00000001
consume2: 0x00000001
produce2: 0x00000003
consume2: 0x00000003
...
produce2: 0x00000ffc
consume2: 0x00000ffc
produce2: 0x00000ffd

```

```

consume2: 0x00000ffd
produce2: 0x00000ffe
consume2: 0x00000ffe
produce2: 0x00000fff
consume2: 0x00000fff

```

task1 루틴에서 데이터를 생산하고, task2 루틴에서 데이터를 소비하는 상황을 볼 수 있다.

이상에서 하드웨어 인터럽트 루틴에서 태스크 루틴으로, 또 태스크 루틴에서 또 다른 태스크 루틴으로 데이터를 주고받는 IPC를 구현해 보았다.

지금까지 10여 회에 걸쳐 임베디드용 OS를 설계하고 구현해 보았다. 사실 OS에 대한 설계라기보다는 OS의 구현이 어떻게 이루어지는지에 초점이 맞추어져 있다. 필자의 의도는 독자들이 그 과정에서 OS의 동작을 이해하고 나아가 응용하는 데에 있다. 임베디드 시스템이란 용어는 OS와 밀접한 관련이 있으며, 임베디드 시스템과 관련된 개발을 하기 위해서는 OS에 대한 이해가 필수불가결하다고 필자는 생각하고 있다. 부족하지만 본 기사가 국내 임베디드 시스템의 발전에 조금이라도 밑거름이 되기를 바라며 그 동안 기사를 읽어 주신 독자들에게 감사의 말을 드리고 싶다. 글을 마친다. ^{Real}Time

