uC/OS-Ⅱ 뛰어넘기(Ⅳ)

공유자원, Semaphore, Mutex

지난 시간에는 Task에 관한 내용들을 다뤘다. Task 동작에 대해서 정확하게 이해했다면 Multitasking 환경에 대해서 충분히 이해를 했다고 할 수 있다. 하지만, multitasking 환경은 미묘한 여러 가지 상황들이 발생하여 우리가 예상치 못한 일들이 발생하곤 한다. 그래서 이번 회에는 이러한 여러 가지 문제들을 해결할 수 있는 동기화나 내부통신과 같은 여러 가지 서비스들을 공부해 보도록 하자.

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

우리가 앞에서 공부한 내용들로 Task나 multitasking이라는 것에 대해서 어느 정도 감을 잡았으리라 생각된다. 그렇지만, 이것은 다음에 나올 내용들을 배우는데 있어서 기초가 되므로 확실하게 이해를 해두시기를 당부 드리고 싶다.

지난 회의 내용을 간단히 정리하자면, Task들 자체의 동작과 OSTimeDly() 함수를 이용하여 RUN 상태의 task가 WAIT 상태로 변화되고 다시 READY 상태로 동작되는 일련의 과정들을 공부했었다. 그러면 이것으로 모든 것이 끝났을까? 그렇지 않다. multitasking이라는 환경에는 우리가 보통 상상하는 것과 달리 복잡하고 미묘한 관계가 숨어있다. 그렇다고 너무 걱정하지 말고 이러한 문제들을 하나씩 짚어보면서 이야기를 풀어보도록 하자.

공유자원

그림 1은 일반 PC 환경에서 네트워크를 이용하여 프린터를 공유하는 일반적일 형태의 구성이다. 여기서 여러분이 10장의 문서를 출력하는 경우를 생각해보자. 5장 정도가 출력됐을 때, 중간에 다른 누군가가 이 프린터기를 이용하여 출력을 한다면 내 출력물 중간에 다른 사람의 출력물이 들어가게 되어 우리가 원하는 출력물을 얻기 힘들 것이다. 물론, 현재의 시스템에서는

그림 1. PC 환경에서의 네트워크 공유 형태

이러한 문제가 발생하지 않는다. 이러한 상황은 전산학 개론과 같은 책에서 옛날 초창기의 컴퓨터 시스템에 대해 가끔 언급될 때 나오는 내용들이다. 그렇다면 이것이 우리가 공부하고 있는 RTOS, 즉 multitasking 환경과 무슨 관계일까?

이번에는 그림 1의 PC들을 Task라고 가정하고 생각해보자. 여러 개의 PC에서 출력을 내보내는 것이나 여러 개의 Task가

출력을 내보내는 것이나 상황은 비슷할 것이다. 즉, 하나의 Task가 어떠한 자원(여기서는 프린터)을 사용하는 중간에 다른 Task가 똑같은 자원을 사용한다면 이와 같은 현상이 발생되게 된다.

이것은 프린터와 같은 장치에 국한되지 않는다. 보통은 프로그램 내부에서 사용하는 변수, 구조체와 같은 메모리에도 해당이 된다.

☞ 선점형 Kernel에서는 수행중인 Task가 언제든지 우선순위가 높은 Task에게 선점당할 수 있으므로 사용중인 자원의 제어권을 뺏길 가능성이 아주 많다.

```
int res;

void Task1(void* para)
{
      ...
      OSTimeDly(1); ---------------- (A)
      res = 0;
      OSTimeDly(1);
      ...

}

void Task2(void* para)
{
      ...
       res = 3;    -------------------- (B)
      OSTimeDly(1);
      res += 5;
      printf("Result = %d",res);
      ...
}
```

리스트 1

Task가 사용할 수 있는 모든 것을 우리는 "자원"이라고 호칭을 하고, 이 자원이 다른 Task에 의해서도 사용된다면 "공유자원"이라고 호칭을 하게 된다.

공유자원은 여러 개의 Task에 의해서 사용이 되므로 위에서 설명한 것과 같은 문제가 발생될 수 있게 된다. 그래서 **리스트 1**의 예제를 보면서 좀 더 실질적인 문제점을 살펴보도록 하겠다.

리스트 1에서 Task1이 Task2보다 우선순위가 높고, Task2는 (B)의 위치에서 Task1에게 선점되어 있고 Task1은 (A)를 수행할 차례라고 가정을 해보자.

위의 출력결과는 얼마일까?

답은 5이다. 만약, 8일라고 답하신 분이 있으시면 지난달의 기사를 보면서 복습을 하시기 바란다.

아래는 프로그램의 수행순서이다.

① Task1 : OSTimeDly(1)을 수행 후, Wait 상태
② Task2 : res = 3
③ Task2 : OSTimeDly(1)을 수행 후, Wait 상태
④ 1Tick후 Task1과 Task2가 Ready됨
⑤ Task1 : res = 0
⑥ Task1 : OSTimeDly(1)을 수행 후, Wait 상태
⑦ Task2 : res += 5
⑨ Task2 : printf() 수행

④에서는 Task1과 Task2가 동시에 ready가 됐을 때, 우선순위에 의해 Task1이 수행된다.

Task2를 프로그램 할 때는 8이란 결과를 예측했을 것이다. 하지만, 이것은 multitasking 환경이 아닌 경우에는 해당이 되지만, multitasking 환경에서는 이러한 자원(전역변수, 여기서는 int a)이 여러 개의 task에 의해서 공유가 되어 위와 같은 상황이 발생하게 된다. 그래서 이러한 공유자원을 안전하게 사용하기 위해서 Semaphore('세마포어' 라고 읽는다)를 사용하게 된다.

Semaphore

위와 같은 공유자원 문제를 해결하기 위한 방법을 생각해보도록 하자. 문제는 어떤 Task가 자원을 사용하는 중간에 다른

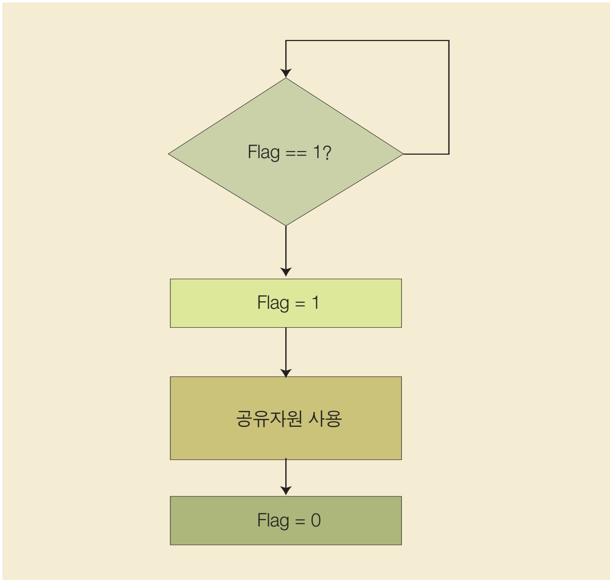

그림 2

Task에 의해서 간섭을 받기 때문이다. 결국, 이러한 간섭만 배제한다면 공유자원 문제는 발생하지 않게 된다. 그렇다면, 사용 중인지 아닌지를 알 수 있는 방법은 어떻게 구현할 것인가? 아마도 flag를 떠올리는 분들이 많으리라 생각된다.

자원을 사용하기 전에 flag가 1인지 체크를 해서 1이면 기다리고, 0이면 flag를 1로 만들고 그 자원을 사용하면 될 것이다. 그리고 다 사용하고 나면 그 flag를 다시 0으로 만든다면 여러 개의 Task가 동시에 같은 자원을 사용하지는 않게 될 것이다(그림 2).

하지만 여기에도 여러 가지 문제점들이 존재한다.

1) flag도 전역변수
2) flag가 1인 경우 0인 될 때까지 기다리는 Task들의 처리 방법
3) 자원이 여러 개인 경우

기타 등등의 여러 가지 골치 아픈 문제점들이 존재하게 된다. 결국 이 방법 또한 좋은 방법은 아니다.

(위의 내용들이 무엇을 의미하는지 모르겠다면 직접 구현해 보면 좀 더 쉽게 이해가 갈 것이다.)

이러한 문제를 쉽게 해결해주는 것이 바로 Semaphore이다.

Semaphore를 쉽게 이해하려면 아래의 예를 살펴보면 이해가 쉬울 것이다.

도서관에 20석의 자리가 있다. 그리고 도서관을 사용하려는 사람들이 50명이 있다면 어떻게 될까? 통제가 되지 않는 상황에서는 서로 자리를 차지하기 위해 엉망이 될 것이다. 그래서 이러한 것을 방지하기 위해 도서관에서는 표를 나누어 준다(모르다면 지금 도서관을 한번 방문해 보시길...).

표는 좌석 개수만큼만 있기 때문에 20명의 사람이 들어갔다면 21번째 사람은 표가 없기 때문에 도서관을 이용할 수 없게 된다. 그래서 이 사람들은 이미 들어간 사람들이 나오면서 표를 반환할 때까지 밖에서 기다리게 된다. 이렇게 된다면 20좌석을 원활하게 사용할 수 있게 된다.

내용을 바꿔서 20좌석을 공유자원이라고 하고 이것을 사용하려는 사람들을 Task라고 해보자. 그렇다면 표가 필요한데 이 표에 해당되는 것이 바로 Semaphore이다. 그래서 Semaphore에는 표를 받고 돌려주는 함수가 존재하고, 표를 사용하려면 공유자원의 개수만큼 만들어 주어야 하기 때문에 이러한 함수가 존재하게 된다.

[함수원형]
OS_EVENT* OSSemCreate(INT16U value)

[설명]
세마포어를 만든다.
[인자]
value – 초기 세마포어의 값(표를 주어진 개수만큼 만든다).
[결과]
만들어진 세마포어 정보

[함수원형]
INT8U OSSemPost(OS_EVENT* pevent)

[설명]
세마포어 pevent에 신호를 준다(표를 반납한다).
[인자]
pevent – OSSemCreate()에 의해서 만들어진 세마포어 정보

[함수원형]
void OSSemPend(OS_EVENT* pevent, INT16U timeout, INT8U *err)

[설명]
세마포어 pevent에서 신호를 받는다(표를 받는다).
[인자]
pevent – OSSemCreate()에 의해서 만들어진 세마포어 정보
timeout – 대기시간
err – 에러코드

리스트 2는 리스트 1을 수정해서 공유자원 문제를 Semaphore를 사용하여 해결한 것이다. 이것을 실행시키면 원하는 결과인 8이 출력된다.

```
sem = OSSemCreate(1);
```

이 함수는 Semaphore를 만드는데, 이때 표의 개수는 한 개이다. 다시 말하면 공유자원의 개수는 한 개이다. 그리고 표를 관리하는 표통의 이름은 sem이 된다. 표는 sem이란 곳에서 관리하기 때문에 앞으로 표를 가져가고 반납할 때마다 이 sem란 것을 이용해야 한다.

```
OSSemPend(sem, 0, &err);
OSSemPost(sem);
```

OSSemPend() 함수는 sem이란 표통에서 표를 가져오는 동작을 한다. 그래서, 표가 있으면 표만 가져가고 다음 문장을 수행하게 되지만, 만약 표가 없다면 이 Task는 OSSemPend() 함수에서 Wait 상태로 빠지게 된다. OSSemPend() 함수에 의해서 Wait 상태로 빠진 Task는 계속 Wait 상태에 있다가 표를 받는 순간 Ready 상태가 된다.

☞ OSSemPend()는 Wait 상태가 되게 하는 함수이다.

☞ OSSemPost()는 OSSemPend() 함수에 의해 Wait 상태에 있는 Task를 Ready가 되게 하는 함수이다.

```
int res;
OS_Event* sem

void Task1(void* para)
{
      INT8U err;
      sem = OSSemCreate(1);
      …
      OSTimeDly(1); ------------------- (A)
      OSSemPend(sem, 0, &err);
      res = 0;
      OSSemPost(sem);
      OSTimeDly(1);
      …
}

void Task2(void* para)
{
      INT8U err;
      …
      OSSemPend(sem, 0, &err);
      res = 3;   ---------------------- (B)
      OSTimeDly(1);
      res += 5;
      printf("Result = %d",res);
      OSSemPost(sem);
      …
}
```

리스트 2

이러한 사항들을 기억하고 리스트 2의 수행을 살펴보자.(초기상태는 리스트 1과 같다고 가정하자)

① Task1 : OSTimeDly(1)을 수행 후, Wait 상태
② Task2 : OSSemPend()를 호출하여 표를 얻는다.
③ Task2 : res = 3
④ Task2 : OSTimeDly(1)을 수행 후, Wait 상태
⑤ 1Tick 후 Task1과 Task2가 Ready됨
⑥ Task1 : OSSemPend()를 호출하지만 표가 없어서 Wait 상태가 된다.

⑦ Task2 : res += 5

⑧ Task2 : printf() 수행

⑨ Task2 : OSSemPend()를 수행하여 표를 돌려준다. 이때 표를 받은 Task1은 ready 상태가 되고 Task1의 우선순위가 높기 때문에 Task1이 Task2를 선점하여 Task1이 수행되게 된다.

⑩ Task1 : res = 0

⑪ Task1 : OSTimeDly(1)을 수행 후, Wait 상태

리스트 1과 비교하면 res라는 공유자원이 잘 보호되고 있는 것을 확인할 수 있다.

우리가 공유자원이라고 정한 것을 사용 할 때는, 아래와 같은 형식을 통하여 보호를 하게 된다. 만약 어떠한 Task가 Semaphore를 사용하지 않고 바로 공유자원을 사용하게 되면 오류가 발생하게 되므로, 공유자원이란 것이 명시가 되면 이것을 사용하는 모든 Task는 아래와 같은 형식으로 반드시 사용하여야 한다.

```
sem = OSSemCreate(1);
…
…
OSSemPend(sem, 0, &err);
  공유 자원의 사용
OSSemPost(sem);
```

〈공유자원을 Semaphore로 보호〉

*공유자원의 개수

리스트 1과 리스트 2에서 사용된 자원은 전역변수였다. 그래서 이 자원은 한 개이므로 OSSemCreate(1)이라고 호출하여 표를 1개 만들어주었다.

그럼, 2개 이상이 쓰이는 경우는 언제인가?

간단한 예로 Serial로 예를 들어 보면 일반적으로 PC에는 2개의 Serial이 존재한다. 이때 Serial이란 자원의 개수는 2개이므로 2개의 Task가 동시에 2개의 Serial을 사용할 수 있게 된다. 하지만, 이러한 경우는 Serial이란 자원을 관리해주는 전용 루틴(보통 Driver란 명칭을 사용한다.)이 필요하게 되므로 아직 익숙하지 않으신 분들께서는 2개 이상의 자원을 다루는 것은 좀 더 익숙해진 후에 연습해보는 것이 좋다.

Mutex

Semaphore의 기능에 대해서 충분히 이해하신 분들 중에 이상한 점이나 문제점을 발견하신 분들이 있으실 것이라 생각된다. Semaphore의 기능 자체는 공유자원을 보호하기 위해서 사용하다보니 기능상으로 우선순위가 높은 Task의 선점을 막는 결과를 가져오게 된다.

앞에서 봤던 리스트 2에서도 Task1이 수행되어야 하는 순간에 Semaphore에 의해 Task1이 Task2에 의해 제어권을 뺏겨서 Task2의 동작이 끝날 때까지 Task1의 수행이 보류되는 사태가 발생되게 된다.

물론, 위와 같은 상황은 순간적으로 발생하므로 일반적으로 그렇게 커다란 문제가 되지 않을 수도 있다. 하지만, 이러한 문제는 단순하게 일어나지 않고 많은 수의 Task에 의해 복잡한 양상에서 예상치 못한 순간에 발생이 되며 최악의 경우 최상의 Task가 오랜 기간동안 수행이 안될 수 있는 경우가 발생하게 되어 RTOS의 실시간 기능이 없어질 수도 있는 위험이 발생하게 된다.

이것을 'priority inversion 현상' 이라고 부른다.

즉, 우선순위가 낮은 Task가 우선순위가 높은 Task를 선점하는 효과가 발생된다는 의미이다. 사실 이러한 상황을 여러분들의 프로그래밍 능력이 뛰어나다고 해도 막기는 힘들다. 그래서 이러한 경우를 대비해서 지원되는 함수가 바로 Mutex이다.

Mutex의 동작원리는 Semaphore와 같다. 하지만, 공유자원의 개수가 1로 한정되어 있고 Priority Inversion을 막을 수 있는 기능이 내장되어 있다는 것이 차이점이다. 이 기능은 현재 Mutex를 소유한 Task보다 우선순위가 높은 Task가 Mutex에 접근(OSMutexPend() 함수를 이용하여)할 때 Mutex를 소유한 Task의 우선순위를 임시적으로 Mutex 생성시 설정된 우선순위로 높임으로써 구현되고 있다.

[함수원형]
OS_EVENT* OSMutexCreate(INT8U prio, INT8U* err)

[설명]
Mutex를 만든다.
[인자]
prio – 사용할 최상위 우선순위
err – 에러코드
[결과]
만들어진 Mutex 정보

[함수원형]
INT8U OSMutexPost(OS_EVENT* pevent)

[설명]
Mutex pevent에 신호를 준다.
[인자]
pevent – OSMutexCreate()에 의해서 만들어진 Mutex 정보

[함수원형]
void OSMutexPend(OS_EVENT* pevent, INT16U timeout, INT8U *err)

[설명]
Mutex pevent에서 신호를 받는다.
[인자]
pevent – OSMutexCreate()에 의해서 만들어진 Mutex 정보
timeout – 대기시간
err – 에러코드

사용방법은 Semaphore와 유사하므로 똑같이 사용하면 된다.

리스트 2에서 사용한 Semaphore를 Mutex로 대치하여 리스트 3과 같이 변경할 수 있다.

```
int res;
OS_Event* mu

void Task1(void* para)
{
     INT8U err;
     mu = OSMutexCreate(2, &err);
     …
     OSTimeDly(1); ------------------ (A)
     OSMutexPend(mu, 0, &err);
     res = 0;
     OSMutexPost(mu);
     OSTimeDly(1);
     …

}

void Task2(void* para)
{
     INT8U err;
     …
     OSMutexPend(mu, 0, &err);
     res = 3;      ------------------- (B)
     OSTimeDly(1);
     res += 5;
     printf("Result = %d",res);
     OSMutexPost(mu);
     …
}
```

리스트 3

다음 회에는 동기화 및 내부통신에 대해서 알아보도록 하겠다. RealTime