**강의명: 실시간 커널**

**실습 번호: 2**

**실습 제목: Mutual exclusion(상호 배제)**

**학생 이름: 황귀훈**

**학번: 201710885**

**1. 상호 배제 문제 분석**

**1.1**

**void Task1(void \*pdata)**

**{**

**for (;;) {**

**UARTprintf("11111111111111111111111111111111111111111111111111\n");**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

Task1은 for(;;)문으로 작성되어 무한 루프를 형성하고 있다. Task1은 UARTprintf문을 통해 11111111111111111111111111111111111111111111111111을 한 줄에 출력하고 줄을 바꾼다. 그리고 OSTimeDly()함수를 통해 주어진 tick만큼 wait 상태가 되도록 만드는데 OS\_GetRandom(n)은 0보다 크거나 같고 n보다 작은 랜덤 번호를 리턴 하는 함수(‘utility.h’을 참고)이기 때문에 OSTimeDly(OS\_GetRandom(5))는 0 tick, 1 tick, 2 tick, 3 tick 4 tick중 랜덤 tick 만큼 task의 상태가 wait 가 됨을 의미한다.

**void Task2(void \*pdata)**

**{**

**for (;;) {**

**UARTprintf("22222222222222222222222222222222222222222222222222\n");**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

Task2는 for(;;)문으로 작성되어 무한 루프를 형성하고 있다. Task2는 UARTprintf문을 통해 22222222222222222222222222222222222222222222222222를 한 줄에 출력하고 줄을 바꾼다. 그리고 OSTimeDly()함수를 통해 주어진 tick만큼 wait 상태가 되도록 만드는데 OS\_GetRandom(n)은 0보다 크거나 같고 n보다 작은 랜덤 번호를 리턴 하는 함수(‘utility.h’을 참고)이기 때문에 OSTimeDly(OS\_GetRandom(5))는 0 tick, 1 tick, 2 tick, 3 tick 4 tick중 랜덤 tick 만큼 task의 상태가 wait 가 됨을 의미한다.

**1.2**

OSTaskCreate()함수의 마지막 인수를 보면 Task1은 priority가 20이고 Task2는 priority가 21임을 알 수 있다. 이 프로그램을 수행시키면 한 줄에 Task1()의 출력과 Task2()의 출력이 겹치는 경우가 가끔 발생하는데 그 이유는 다음과 같다. 실시간 커널은 multitasking을 지원하는데 이것은 priority기반의 preemptive scheduling을 한다. multitasking을 하기 때문에 shared resource를 다루는데 문제가 발생하는데 실습 번호 1에서 발생하는 문제는 priority가 다른 두개의 Task인 Task1, Task2가 screen이라는 shared resource를 다루며 multitasking을 하기 때문이다. Multitasking을 할 때 Task1의 priority가 Task2의 priority보다 크기 때문에 Task1의 delay값이 Task2의 delay값보다 작다면 Task2가 수행하고 있을 때 context switching이 발생하여 Task1이 수행되어 한 줄에 Task1()의 출력과 Task2()의 출력이 겹치는 경우가 가끔 발생하는 것이다.

**1.3**

이 프로그램의 출력을 좀더 자세하게 관찰하면, 한 줄에 Task1()의 출력과 Task2()의 출 력이 겹치는 경우에, Task2()의 한 줄 출력이 완성되지 못하는 경우는 발생하지만 Task1()의 한 줄 출력이 완성되지 못하는 경우는 발생하지 않는다. 그 이유는 Task1의 priority는 20, task2의 priority는 21으로 Task1의 priority가 높기 때문에 Task1의 수행이 끝나고 wait 상태가 되어야 Task2가 실행되기 때문에 Task1()의 한 줄 출력은 끊기지 않는 것이고, Task2()의 한 줄 출력이 완성되지 못하는 경우는 Task2() 함수의 실행으로 Task2()의 한 줄이 출력되는 도중에 Task1이 wait에서 빠져나오게 되면 preemptive scheduling을 하기 때문에 context switching이 일어나게 되어 Task2()의 한 줄이 완성되지 못한 채 Task1()의 한 줄이 완성되고 Task1의 수행이 완료되어 wait상태가 되면 그 이후에 다시 Task2가 수행되어 나머지 부분을 출력하기 때문이다.

**2. 상호 배제 문제 해결 (인터럽트)**

**2.1**

**void Task1(void \*pdata)**

**{**

**OS\_CPU\_SR cpu\_sr;**

**for (;;) {**

**OS\_ENTER\_CRITICAL;**

**UARTprintf("11111111111111111111111111111111111111111111111111\n");**

**OS\_EXIT\_CRITICAL;**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

우선 interrupt의 상태를 보존하기 위해 지역변수로 OS\_CPU\_SR 타입의 변수 cpu\_sr을 선언했다.

실시간 커널은 multitasking을 지원하는데 이것은 priority기반의 preemptive scheduling을 한다. multitasking을 하기 때문에 shared resource를 다루는데 문제가 있기 때문에 synchronization을 지원한다. Mutual exclusion이 그 한 종류이다. 2번은 인터럽트를 이용한 상호 배제 문제 해결이다.

μC/OS-II에서 제공하는 두개의 매크로를 사용하여 interrupts을 disabling, enabling할 수 있는데 Interrupt이 발생하면 안 되는 critical sections의 시작 바로 전에 OS\_ENTER\_CRITICAL()을 사용하여 interrupts을 disable하고 critical sections 끝 바로 뒤에 OS\_EXIT\_CRITICAL()을 사용하여 interrupts을 enable함으로써 문제를 해결할 수 있다. 이번 실습에서 shared resource는 screen으로 Task1에서 critical sections은 UARTprintf()함수 이므로 Task1함수를 위와 같이 작성했다.

OS\_ENTER\_CRITICAL()과 OS\_EXIT\_CRITICAL()은 항상 한쌍으로 사용되어 critical sections을 감싼다.

critical section은 shared resource에 여러 Task가 동시 접근하는 것을 방지해주기 위해 한 Task의 독점을 보장해주는 영역이다.

위와 같이 코드를 작성함으로써 critical section안의 프로그램이 수행되는 동안 interrupt이 발생하지 않아 ISR로 가지 않고 context switching 없이 독점적으로 shared resource를 사용할 수 있어 상호배제 시켜줄 수 있는 것이다

**void Task2(void \*pdata)**

**{**

**OS\_CPU\_SR cpu\_sr;**

**for (;;) {**

**OS\_ENTER\_CRITICAL;**

**UARTprintf("22222222222222222222222222222222222222222222222222\n");**

**OS\_EXIT\_CRITICAL;**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

우선 interrupt의 상태를 보존하기 위해 지역변수로 OS\_CPU\_SR 타입의 변수 cpu\_sr을 선언했다.

실시간 커널은 multitasking을 지원하는데 이것은 priority기반의 preemptive scheduling을 한다. multitasking을 하기 때문에 shared resource를 다루는데 문제가 있기 때문에 synchronization을 지원한다. Mutual exclusion이 그 한 종류이다. 2번은 인터럽트를 이용한 상호 배제 문제 해결이다.

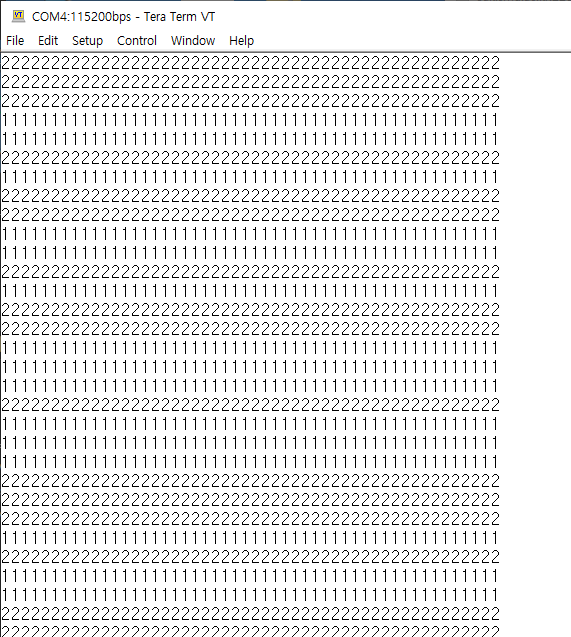
μC/OS-II에서 제공하는 두개의 매크로를 사용하여 interrupts을 disabling, enabling할 수 있는데 Interrupt이 발생하면 안 되는 critical sections의 시작 바로 전에 OS\_ENTER\_CRITICAL()을 사용하여 interrupts을 disable하고 critical sections 끝 바로 뒤에 OS\_EXIT\_CRITICAL()을 사용하여 interrupts을 enable함으로써 문제를 해결할 수 있다. 이번 실습에서 shared resource는 screen으로 Task2에서 critical sections은 UARTprintf()함수 이므로 Task2함수를 위와 같이 작성했다.

OS\_ENTER\_CRITICAL()과 OS\_EXIT\_CRITICAL()은 항상 한쌍으로 사용되어 critical sections을 감싼다.

critical section은 shared resource에 여러 Task가 동시 접근하는 것을 방지해주기 위해 한 Task의 독점을 보장해주는 영역이다.

위와 같이 코드를 작성함으로써 critical section안의 프로그램이 수행되는 동안 interrupt이 발생하지 않아 ISR로 가지 않고 context switching 없이 독점적으로 shared resource를 사용할 수 있어 상호배제 시켜줄 수 있는 것이다

**2.2**

****

**3. 상호 배제 문제 해결 (스케줄러)**

**3.1**

**void Task1(void \*pdata)**

**{**

**for (;;) {**

**OSSchedLock();**

**UARTprintf("11111111111111111111111111111111111111111111111111\n");**

**OSSchedUnlock();**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

실시간 커널은 multitasking을 지원하는데 이것은 priority기반의 preemptive scheduling을 한다. multitasking을 하기 때문에 shared resource를 다루는데 문제가 있기 때문에 synchronization을 지원한다. Mutual exclusion이 그 한 종류이다. 3번은 스케줄러를 이용한 상호 배제 문제 해결이다.

μC/OS-II에서 제공하는 두개의 매크로를 사용하여 scheduling을 disable 할 수 있는데 OSSchedLock()함수는 OSSchedUnlock()함수가 호출될 때까지 Task가 다시 scheduling 되는 것을 방지한다. 이 두개의 매크로도 한 쌍으로 사용되며 critical section을 감싼다. 이번 실습에서 shared resource는 screen으로 Task1에서 critical sections은 UARTprintf()함수 이기 때문에 UARTprintf()함수를 OSSchedLock()과 OSSchedUnlock()로 감싸서 UARTprintf()함수가 실행되는 동안은 scheduling이 일어나지 않아 이 Task보다 priority가 높은 Task가 실행되는 것을 막아준다.

따라서 Task1의 UARTprintf()함수가 수행되는 동안 scheduling이 발생하지 않기 때문에 독점적으로 shared resource를 사용할 수 있어 상호배제 시켜줄 수 있는 것이다

**void Task2(void \*pdata)**

**{**

**for (;;) {**

**OSSchedLock();**

**UARTprintf("22222222222222222222222222222222222222222222222222\n");**

**OSSchedUnlock();**

**OSTimeDly(OS\_GetRandom(5));**

**}**

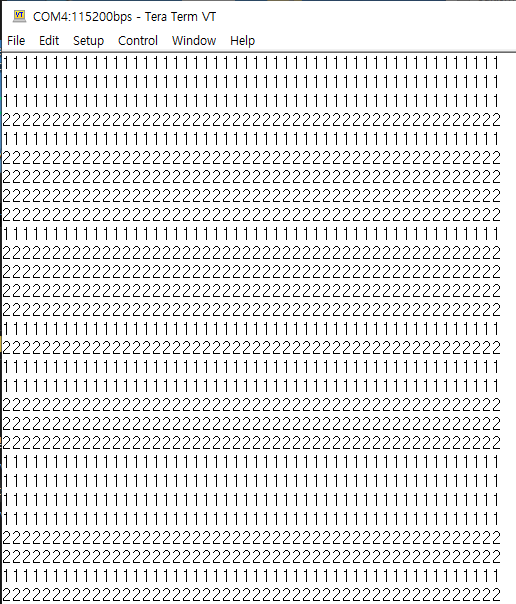
**}**

실시간 커널은 multitasking을 지원하는데 이것은 priority기반의 preemptive scheduling을 한다. multitasking을 하기 때문에 shared resource를 다루는데 문제가 있기 때문에 synchronization을 지원한다. Mutual exclusion이 그 한 종류이다. 3번은 스케줄러를 이용한 상호 배제 문제 해결이다.

μC/OS-II에서 제공하는 두개의 매크로를 사용하여 scheduling을 disable 할 수 있는데 OSSchedLock()함수는 OSSchedUnlock()함수가 호출될 때까지 Task가 다시 scheduling 되는 것을 방지한다. 이 두개의 매크로도 한 쌍으로 사용되며 critical section을 감싼다. 이번 실습에서 shared resource는 screen으로 Task2에서 critical sections은 UARTprintf()함수 이기 때문에 UARTprintf()함수를 OSSchedLock()과 OSSchedUnlock()로 감싸서 UARTprintf()함수가 실행되는 동안은 scheduling이 일어나지 않아 이 Task보다 priority가 높은 Task가 실행되는 것을 막아준다.

따라서 Task2의 UARTprintf()함수가 수행되는 동안 scheduling이 발생하지 않기 때문에 독점적으로 shared resource를 사용할 수 있어 상호배제 시켜줄 수 있는 것이다

**3.2**

****

**4. 상호 배제 문제 해결 (세마포어)**

**4.1**

**void TaskStart(void \*pdata);**

**void Task1(void \*pdata);**

**void Task2(void \*pdata);**

**OS\_STK TaskStartStk[TASK\_STK\_SIZE];**

**OS\_STK Task1Stk[TASK\_STK\_SIZE];**

**OS\_STK Task2Stk[TASK\_STK\_SIZE];**

**OS\_EVENT \*Sem;**

**int main(void)**

**{**

**OSInit();**

**Sem = OSSemCreate(1);**

**OSTaskCreate(TaskStart, (void \*) 0, &TaskStartStk[TASK\_STK\_SIZE - 1],**

**10);**

**OSStart();**

**return 0;**

**}**

실시간 커널은 multitasking을 지원하는데 이것은 priority기반의 preemptive scheduling을 한다. multitasking을 하기 때문에 shared resource를 다루는데 문제가 있기 때문에 synchronization을 지원한다. Mutual exclusion이 그 한 종류이다. 4번은 세마포어를 이용한 상호 배제 문제 해결이다.

semaphore를 사용해야 하기 때문에 함수 시작 전에 OS\_Event 타입의 변수 Sem을 선언했다.

main()함수에서 semaphore를 생성했는데 이번 실습은 Task1, Task2두개의 Task를 사용하고 1개의 semaphore를 사용하는 경우로 semaphore Sem을 OSSemCreate()함수를 사용하여 만들었다. 이때 인수는 1로 Sem의 초기 값은 1이다.

**void Task1(void \*pdata)**

**{**

**INT8U err;**

**for (;;) {**

**OSSemPend(Sem, 0, &err);**

**UARTprintf("11111111111111111111111111111111111111111111111111\n");**

**OSSemPost(Sem);**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

INT8U 타입의 err을 선언했는데 INT8U는 unsigned char와 같다.

Task와 ISR은 하나이상의 Semaphore를 사용하여 Synchronization을 할 수 있는데 semaphore는 초기 값을 갖는 integer variable이다. 또한 semaphore는 두개의 atomic operations을 갖는데 PEND()와 POST()이다. PEND()는 만약 Sem이 양수일 때 Sem값을 1감소시킨다. 즉 PEND()함수는 Sem이 0보다 작거나 같을 때에는 계속 wait 상태임을 의미한다. POST()는 Sem을 1 증가시킨다.

shared resource인 screen에서 Task1의 수행 도중에 Task2가 출력되지 못하게 해주고자 출력을 나타내는 UARTprintf()함수 위, 아래로 OSSemPend()와 OSSemPost()함수를 작성해주었다. 동작원리는 다음과 같다. 무한 루프 속에서 OSSemPend(Sem, 0, &err)를 통해 Sem값이 양수가 될 때까지 기다리다 다른 Task의 결과 Sem이 양수가 되면 그 Sem을 1 감소시키고 UARTprintf()을 실행한다. 이 때 다른 Task는 간섭하지 못한다. UARTprintf()함수 수행이 완료되면 OSSemPost()함수에 의해 Sem을 1 증가시킨 뒤 OSTimeDly(OS\_GetRandom(5))함수에 의해 0 tick, 1 tick, 2 tick, 3 tick 4 tick중 랜덤 tick 만큼 task의 상태가 wait 된다. 그 후 다시 OSSemPend(Sem, 0, &err)를 통해 Sem값이 양수가 될 때까지 기다린다. UARTprintf()함수를 수행한뒤 Sem을 1 증가시키기 때문에 Task2가 수행이 가능해진다.

OSSemPend(Sem, 0, &err)에서 첫 번째 인수는 위에서 만들었던 OSEvent 타입의 Sem과 같다. 두 번째 인수 0은 timeout을 설정한 값인데 timeout은 최대 기다리는 시간을 설정하는 것으로 이번 실습에서는 0으로 timeout없이 무한히 기다림을 의미한다. 마지막으로 &err은 OSSemPend()함수결과 오류가 발생하면 알려주기 위한 오류의 주소이다.

**void Task2(void \*pdata)**

**{**

**INT8U err;**

**for (;;) {**

**OSSemPend(Sem, 0, &err);**

**UARTprintf("22222222222222222222222222222222222222222222222222\n");**

**OSSemPost(Sem);**

**OSTimeDly(OS\_GetRandom(5));**

**}**

**}**

INT8U 타입의 err을 선언했는데 INT8U는 unsigned char와 같다.

Task와 ISR은 하나이상의 Semaphore를 사용하여 Synchronization을 할 수 있는데 semaphore는 초기 값을 갖는 integer variable이다. 또한 semaphore는 두개의 atomic operations을 갖는데 PEND()와 POST()이다. PEND()는 만약 Sem이 양수일 때 Sem값을 1감소시킨다. 즉 PEND()함수는 Sem이 0보다 작거나 같을 때에는 계속 wait 상태임을 의미한다. POST()는 Sem을 1 증가시킨다.

shared resource인 screen에서 Task2의 수행 도중에 Task1가 출력되지 못하게 해주고자 출력을 나타내는 UARTprintf()함수 위, 아래로 OSSemPend()와 OSSemPost()함수를 작성해주었다. 동작원리는 다음과 같다. 무한 루프 속에서 OSSemPend(Sem, 0, &err)를 통해 Sem값이 양수가 될 때까지 기다리다 다른 Task의 결과 Sem이 양수가 되면 그 Sem을 1 감소시키고 UARTprintf()을 실행한다. 이 때 다른 Task는 간섭하지 못한다. UARTprintf()함수 수행이 완료되면 OSSemPost()함수에 의해 Sem을 1 증가시킨 뒤 OSTimeDly(OS\_GetRandom(5))함수에 의해 0 tick, 1 tick, 2 tick, 3 tick 4 tick중 랜덤 tick 만큼 Task의 상태가 wait 된다. 그 후 다시 OSSemPend(Sem, 0, &err)를 통해 Sem값이 양수가 될 때까지 기다린다. UARTprintf()함수를 수행한뒤 Sem을 1 증가시키기 때문에 Task1이 수행이 가능해진다.

OSSemPend(Sem, 0, &err)에서 첫 번째 인수는 위에서 만들었던 OSEvent 타입의 Sem과 같다. 두 번째 인수 0은 timeout을 설정한 값인데 timeout은 최대 기다리는 시간을 설정하는 것으로 이번 실습에서는 0으로 timeout없이 무한히 기다림을 의미한다. 마지막으로 &err은 OSSemPend()함수결과 오류가 발생하면 알려주기 위한 오류의 주소이다.

**4.2**

**테이블이(가) 표시된 사진

자동 생성된 설명**

끝.