DOI: 10.3745/KIPSTA.2009.16A.6.463

MicroC/OS-II 실시간 운영체제에서의 우선순위 역전현상 해결을 위한 일시적 우선순위 교환 프로토콜 설계 및 구현

전 영 식[†]·김 병 곤[††]·허 신[†††]

요 약

실시간 운영체제는 효율적인 스케줄링, 최소화된 인터럽트 지연, 우선순위 역전현상 해결 등의 다양한 조건을 만족시킴과 동시에, 응용프로그램이 정해진 기한 내에 수행되는 것을 보장하여야 한다. 따라서 실시간 운영체제는 상기 조건을 만족시킬 수 있도록 설계/개발되어야 한다. 대중적인 실시간 커널의 한 종류인 MicroC/OS-II에서는 우선순위 역전 현상에 대한 해결 기법으로 뮤텍스(Mutex)를 사용한 기본적인 우선순위 상속(Basic Priority Inheritance) 프로토콜을 사용한다. 뮤텍스를 구현하려면 우선순위가 같은 여러 태스크를 사용할 수 있도록 커널이 지원해야 하나 MicroC/OS-II 운영체제는 우선순위가 같은 여러 태스크의 동시 사용을 지원하지 않는다. 이를 해결하기 위해 추가적인 우선순위 예약을 사용할 수밖에 없게 되고, 결과적으로 제한된 우선순위 자원을 낭비하게 된다. 본 논문에서는 MicroC/OS-II의 불필요한 우선순위 자원을 낭비하는 문제점을 해결할 수 있는 일시적 우선순위 교환 프로토콜(Temporary Priority Swapping Protocol; TPSP)을 설계 및 구현하여, 한정된 자원 환경을 가진 임베디드 장비에 효율적으로 운용되도록 하는데 목적을 둔다.

키워드 : MicroC/OS-II, 실시간 운영체제, 우선순위 역전현상, 임베디드 시스템

Design and Implementation of a Temporary Priority Swapping Protocol for Solving Priority Inversion Problems in MicroC/OS-II Real-time Operating System

Young Sik Jeon[†] · Byung Kon Kim[††] · Heu Shin[†††]

ABSTRACT

Real-time operating systems must have satisfying various conditions such as effective scheduling policies, minimized interrupt delay, resolved priority inversion problems, and its applications to be completed within desired deadline. The real-time operating systems, therefore, should be designed and developed to be optimal for these requirements. MicroC/OS-II, a kind of Real-time operating systems, uses the basic priority inheritance with a mutex to solve priority inversion problems. For the implementation of mutex, the kernel in an operating system should provide supports for numerous tasks with same priority. However, MicroC/OS-II does not provide this support for the numerous tasks of same priority. To solve this problem, MicroC/OS-II cannot but using priority reservation, which leads to the waste of unnecessary resources. In this study, we have dealt with new design a protocol, so called TPSP(Temporary Priority Swap Protocol), by an effective solution for above-mentioned problem, eventually enabling embedded systems with constrained resources environments to run applications.

Keywords : MicroC/OS-II, Real-time Operation Systems, Priority Inversion, Embedded System

1. 서 론

실시간 시스템은 시스템의 정확성이 논리적 정확성뿐만 아니라 결과를 마감시한에 맞추는 시간적 정확성에 의존하는 시스템이다. 실시간 운영체제는 최적의 실시간 시스템에 맞도록 설계되고 개발되어야 하며[1], 이를 위해서는 효율적인 스케줄링 정책, 인터럽트 지연 최소화, 우선순위 역전현상 해결 등의 조건을 만족시켜야 한다. 실시간 운영체제의 한 종류인 MicroC/OS-II에서는 태스크간 동기화를 위해 뮤텍스(Mutex)를 사용하며, 우선순위 역전현상에 대한 해결 기법으로 기본적인 우선순위 상속(Basic Priority Inheritance;

† 준 회 원 : 한양대학교 컴퓨터공학과 석사과정
†† 정 회 원 : 한국건설기술연구원 건설정보연구실 선임연구원
††† 정 회 원 : 한양대학교 컴퓨터공학과 교수
논문접수 : 2009년 7월 9일
수 정 일 : 1차 2009년 10월 15일, 2차 2009년 10월 19일
심사완료 : 2009년 10월 19일

BPI) 프로토콜[2]을 사용한다.

하지만, 뮤텍스를 이용하여 BPI 프로토콜을 구현할 경우 MicroC/OS-II 자체가 우선순위가 같은 여러 태스크를 지원해야 하는 경우가 발생하는데 MicroC/OS-II는 동일한 우선순위를 지원하지 않는다.

MicroC/OS-II는 BPI를 구현하기 위한 우회적인 방법으로 우선순위 예약 방식을 사용한다. 이 방식은 뮤텍스를 접근하는 태스크들 중 가장 높은 우선순위를 가지는 태스크보다 높은 우선순위를 미리 예약해 뮤텍스를 할당한다. 그리고 우선순위 역전현상이 발생될 경우, 예약된 우선순위를 태스크가 상속받아 작업을 최대한 빨리 마치게 한다.

이것은 태스크간 동기화에 필요한 자원(예. 공유데이터 또는 파일갱신, 입출력 장치 등의 비공유 자원)의 수에 비례하여 우선순위 예약을 추가로 해야 한다는 것을 의미하며 태스크가 사용할 수 있는 우선순위 수의 감소를 초래할 것이다.

이것은 시스템에서 예약한 우선순위를 제외한 실제 사용할 수 있는 우선순위의 개수가 56개뿐인 MicroC/OS-II에서는 매우 비효율적이고 중요한 문제이다[3].

본 논문에서는 이런 문제점을 해결할 수 있는 우선순위 교환 프로토콜(Temporary Priority Swap Protocol; 이하 TPSP)을 제안하고자 한다. TPSP는 동기화를 요구하는 두 태스크간의 우선순위 역전현상이 발생할 경우, 두 태스크간의 우선순위를 서로 교환하는 방식을 사용한다.

2. 관련 연구

2.1 우선순위 역전현상과 기존의 해결 기법

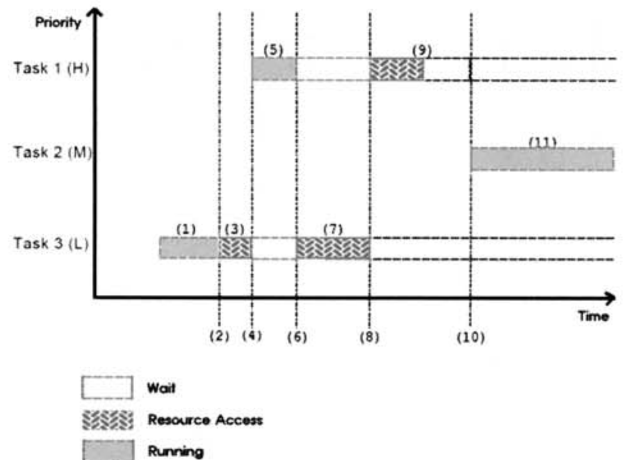
우선순위 역전현상이란, 실시간 운영체제에서 특정 프로세스가 특정 자원을 필요로 하고, 그 해당되는 자원을 소유한 프로세스보다 높은 우선순위를 가짐에도 불구하고, 할당받지 못해 대기하는 현상을 말한다[2].

2.1.1 우선순위 상속(Basic Priority Inheritance: BPI)

BPI 프로토콜[2]은 자원을 소유하고 있는 프로세스에게, 해당 자원을 요청하고 대기하는 프로세스들 중에서 가장 높은 우선순위를 상속받아 해당 프로세스가 선점되지 않고 실행을 완료하도록 한 후, 자원을 반환할 때 원래의 우선순위로 복귀하는 방식이다.

(그림 1)은 우선순위 역전현상이 발생했을 때, BPI 프로토콜을 적용한 예이다.

(그림 1)에서 볼 수 있듯이 태스크(Task) 1이 뮤텍스를 획득하고자 시도할 때(6), 커널은 태스크 3이 뮤텍스를 소유하고 있고 태스크 3의 우선순위가 태스크 1보다 낮다는 것을 안다. 이 경우 커널은 태스크 3의 우선순위를 태스크 1과 같은 수준으로 올린다(6). 커널은 태스크 1을 뮤텍스의 대기목록에 올린 뒤 공유 자원을 계속 사용하도록 태스크 3을 재실행한다(7). 태스크 3이 자원을 다 사용한 뒤에는 뮤텍스를 양도한다. 이때 커널은 태스크 3의 우선순위를 원래



(그림 1) BPI 프로토콜을 지원하는 커널

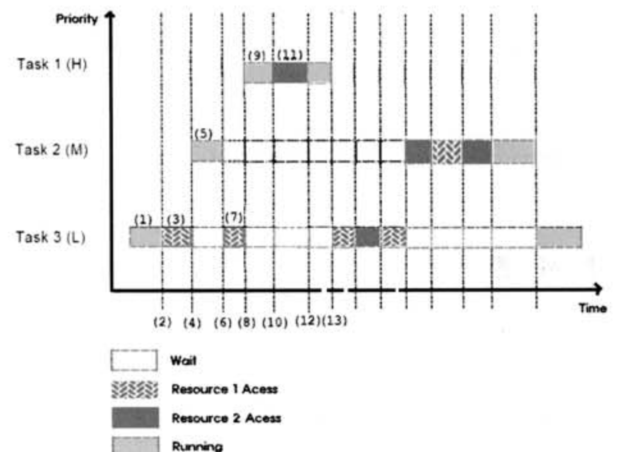
우선순위 값으로 되돌려 놓고 뮤텍스를 기다리는 태스크가 있는지 알아내기 위해 대기 목록을 조사한다(8). 커널은 태스크 1이 뮤텍스를 기다리고 있다는 사실을 발견하고 태스크 1한테 뮤텍스를 준다. 태스크 1이 작업을 마치면 중간 우선순위인 태스크 2가 실행된다[4].

2.1.2 우선순위 상한 프로토콜(Priority Ceiling Protocol: PCP)

우선순위 상한 프로토콜에서 모든 태스크의 우선순위와 자원 요구 사항은 미리 알려져 있다. 특정 자원에 대해서 우선순위 상한 값(Priority Ceiling)이란 해당 자원을 사용할지도 모르는 태스크 중에서 가장 우선순위가 높은 태스크의 우선순위를 의미한다[5].

(그림 2)는 우선순위 상한 프로토콜의 예를 보여준다.

(그림 2)에서 태스크 3이 자원 R1을 사용하려고 요청할 때(2), 자원 R1이 사용 가능 상태일 경우 태스크 2는 자원 R1의 우선순위 상한 값으로 높아진다. 우선순위 상한 값이



(그림 2) 우선순위 상한 프로토콜의 예

란 자원을 사용할 태스크들중 가장 높은 우선순위(R1의 경우 태스크 2의 우선순위가 상한 값이 됨)를 말하며, 특정 태스크가 모든 자원을 반환하면 태스크의 우선순위는 본래의 우선순위로 변경된다[6].

2.2 MicroC/OS, MicroC/OS-II에서 우선순위 역전현상 해결을 위한 기존의 연구

관련 연구를 비롯하여 우선순위 역전을 방지하는 방법은 BPI[2], PCP[5], IPCP[7], DPCP[8], RPI[9]등 오래전부터 연구가 진행되어 왔다. 하지만 이들을 MicroC/OS-II에 그대로 적용하기에는 다소 무리가 있다.

우선, 태스크당 다수의 자원을 가졌을 때 발생하는 우선순위 역전현상을 고려한 PCP, IPCP, DPCP, RPI 프로토콜은 MicroC/OS-II에서는 불필요하다. MicroC/OS-II에서의 각 태스크는 오직 하나의 자원을 할당 받을 수 있기 때문이다.

또한, MicroC/OS-II의 이전 버전인 MicroC/OS에서부터 고수되어 왔던 정책인 같은 우선순위를 가지지 못하는 운영체제 특성상 BPI 프로토콜의 경우도 역시 곧바로 적용이 불가능하다.

따라서, MicroC/OS 및 MicroC/OS-II의 기존 연구는 대부분 동일 우선순위 구현이 가능하도록 같은 우선순위들을 연결리스트로 연결하는 방식으로 수정되어 왔다[3, 10, 11]. 하지만 이 방식은, 시스템의 반응성이 낮아지고 태스크간의 스케줄링 복잡성 증가의 원인이 되기 때문에 실시간 운영체제인 MicroC/OS-II에 상용화하기에는 무리가 있다.

2.3 기존 MicroC/OS-II에서의 프로토콜의 구현

우선순위 역전현상 해결을 위한 기존 MicroC/OS-II의 BPI 프로토콜은, 우선순위 상속을 위해 뮤텍스를 액세스하고자 하는 최상위 태스크 바로 위에 있는 우선순위를 예약해 놓고, 필요할 때만 낮은 우선순위 태스크의 우선순위를 임시로 올려서 사용하는 방식을 취한다[4].

이 방식은, 연결리스트 방식을 이용하여 커널을 수정하는 기존의 연구 방식에 비해 스케줄의 복잡성을 최소화할 수 있는 장점이 있지만, 우선순위의 낭비를 초래하게 된다.

그러므로, 기존 프로토콜의 장점을 그대로 유지하되 기존의 MicroC/OS-II에서 발생되었던 우선순위 낭비 문제를 해결하는 프로토콜을 새로 설계할 필요가 있다.

3. TPSP(Temporary Priority Swap Protocol) 정의 및 구현

3.1 전제

3.1.1 우선순위에 의한 프로세서 선점

실행되기 위해 준비리스트에 대기 중인 모든 태스크들보다 높은 우선순위를 가지는 태스크 T가 프로세서를 사용하기 위해 선점된다.

프로세서 선점은 준비리스트에 대기 중인 태스크들 중에서만 결정되며, 이들 태스크들 중 우선순위가 가장 높은 값을 가진 태스크가 선점된다.

3.1.2 뮤텍스 소유

태스크 T가 뮤텍스 M을 사용할 수 있는 경우는 오직 해당 뮤텍스 M을 소유하고 있는 태스크가 없는 경우뿐이다.

3.1.3 뮤텍스 반환

태스크 T가 뮤텍스 M을 반환할 때는 해당 뮤텍스 M의 사용을 종료하였거나 혹은 뮤텍스 M의 사용을 완료하지 않았지만 다른 뮤텍스의 사용을 요청할 때뿐이다.

즉, 각 태스크는 한 순간에 오직 하나의 자원만 가질 수 있다는 뜻이다. 이것은 기존의 BPI를 적용할 경우 발생할 수 있는 교착상태를 방지하기 위한 MicroC/OS-II의 정책이며, 마찬가지로 TPSP를 구현하는데 있어 발생할 수 있는 교착상태를 원천적으로 방지한다.

3.2 TPSP의 정의

3.2.1 뮤텍스의 요청

태스크 T_M이 뮤텍스 M을 소유하였고 태스크 T_M보다 높은 우선순위를 가지는 태스크 T_H가 뮤텍스 M을 요청한다. 먼저, 뮤텍스 M을 소유하고 있는 태스크 T_M의 우선순위 값이 T_H가 아닌 다른 태스크 T_L의 우선순위로 이미 바뀌어 있는 경우가 아닌지를 검사한다.

바뀌어 있는 경우라면, 태스크 T_M과 태스크 T_L의 우선순위를 교환하여 원래의 우선순위로 복귀한다. 그 후, 태스크 T_H는 태스크 T_M이 뮤텍스 M을 반납할 때 까지 블록 된다. 이때 태스크 T_M과 태스크 T_H의 우선순위는 서로 교환된다.

우선순위가 변경되지 않았을 경우, 즉 T_M이 자신의 우선순위를 그대로 가지고 있는 경우에는 T_M 과 T_H의 우선순위를 서로 교환한다.

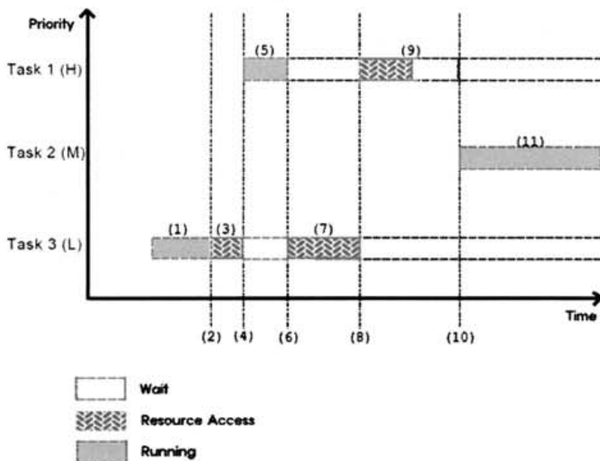
3.2.2 뮤텍스의 반납

태스크 T_M이 뮤텍스 M의 사용을 완료하면, 또 다른 태스크 T_L이 뮤텍스 M을 사용할 수 있도록 뮤텍스 M의 소유를 반납해야 한다.

만약 뮤텍스 M을 반납하기전에 태스크 T_M의 우선순위가 또 다른 태스크 T_L의 우선순위로 바뀌어 있는 경우, T_M과 T_L의 우선순위를 서로 교환하여 원래대로 우선순위를 복귀한 후 뮤텍스 M을 반납한다. 바뀌지 않았다면, 즉 T_M이 자신의 우선순위를 가지고 있는 경우에는 곧바로 뮤텍스 M을 반납한다.

(그림 3)은 우선순위 역전 현상이 발생한 상황에서의 TPSP를 적용 예를 보여준다.

(그림 3)에서처럼 Task 1은 Task 2보다 우선순위가 높고, Task 2는 Task 3보다 우선순위가 높다. 처음 준비리스트에 실행을 위해 대기 중인 태스크는 Task 3뿐 이므로 가장 먼저 Task 3 실행된다(1). 그 후 Task 3은 뮤텍스를 요청하는데(2), 뮤텍스를 소유한 태스크가 없으므로 Task 3이



(그림 3) TPSP를 적용한 예

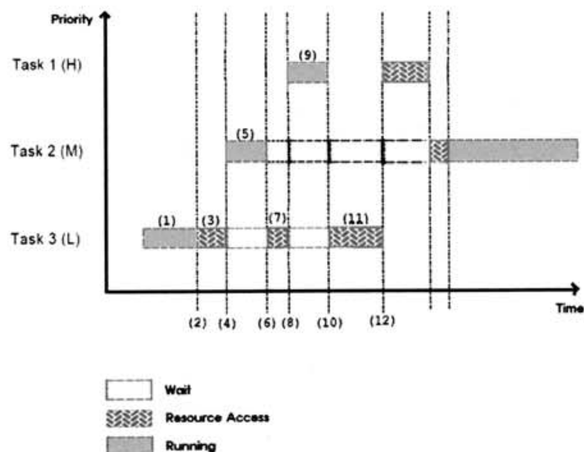
뮤텍스를 소유하고 사용한다(3). Task 3이 소유한 자원에 Task 1이 접근하려 할 때(6), Task 1과 Task 3의 우선순위는 서로 교환된다. Task 3의 우선순위로 바뀐 Task 1을 커널은 뮤텍스의 대기 목록에 올린 뒤 스케줄러 함수를 호출하여 우선순위가 바뀐 Task 3을 재실행한다.

Task 3이 뮤텍스 사용을 완료하고 커널에 반납하려고 할 때, 커널은 Task 1과 Task 3의 우선순위를 원래 상태로 되돌려 놓는다(8).

(그림 4)에서는 TPSP이 적용된 후, 우선순위 역전현상이 중첩적으로 발생했을 경우에 태스크들이 어떤 순서로 실행되는지를 보여준다.

(그림 4)에서처럼, Task 3은 이미 Task 2와 우선순위가 서로 교환된 상태이다(6). 그 후 Task 1이 프로세서를 선점하고 실행되다가(9) 뮤텍스를 요청한다(10).

하지만 뮤텍스는 이미 Task 3이 소유한 상태이기 때문에 Task 1은 블록상태가 된다. 이 경우에도 마찬가지로 Task 3과 Task 1간에 TPSP를 적용해야 하지만, 이미 Task 3이 TPSP를 적용하였다. 즉, Task 3이 Task 2의 우선순위를



(그림 4) 중첩된 우선순위 역전현상에 대한 예

가지고 있으므로 Task 3과 Task 2의 우선순위를 원래대로 복귀한다. 그 후에 Task 3과 Task 1의 우선순위를 교환한다. 마지막으로 Task 3이 뮤텍스 사용 을 완료하고 커널에 반납하려고 할 때, 커널은 Task 3과 Task 1의 우선순위를 원래 상태로 되돌려 놓는다(12).

3.3 함수의 설계

본 논문에서 제안하는 TPSP는 기존의 우선순위 예약을 사용하는 BPI방식에서 사용한 함수들을 그대로 사용해도 구현이 가능하다. 즉, 뮤텍스를 사용하여 우선순위 역전현상을 해결하는 것 자체는 동일하기 때문에 함수의 이름들을 동일하게 사용한다.

다만, 우선순위 역전현상이 발생 되었을 경우에 실제 처리 하는 방식이 다르기 때문에 그에 대한 일부 함수들 의 내부 구현은 수정되어야 할 것이다.

TPSP를 적용하기 위해 ECB(Event Control Block으로 Mutex정보를 저장하는 구조체)의 변수인 OSEventCnt를 사용한다. OSEventCnt는 Mutex를 소유한 태스크의 우선순위 값을 저장하기 위한 16비트 int형 변수이다. 하위 8비트는 해당 Mutex가 사용 가능한지를 나타낸다.

OS_MUTEX_AVAILABLE(0xFF)값인 경우 해당 Mutex를 소유한 태스크가 없다는 뜻이고, 그 외의 경우는 해당 Mutex를 소유한 태스크의 우선순위 값을 나타낸다.

상위 8비트는 해당 Mutex가 TPSP를 적용했는지를 나타낸다. 초기 값은 0인 OS_ASCII_NUL를 가진다. TPSP에 의해 해당 Mutex를 소유한 태스크의 우선순위 값이 변경될 경우에 변경되기 전 원래의 우선순위 값을 저장한다. 이때는 추후 원래의 우선순위로 복귀하는데 사용된다.

3.4 함수 변경

3.4.1 OSMutexCreate() 함수

<코드 1>의 OSMutexCreate 함수는 태스크 동기화를 위한 Mutex를 생성하는 함수이다. 기존의 OSMutexCreate 함수의 경우 매개변수로 예약하려는 우선순위 값을 받아서 새로 생성하는 ECB에 할당해야 한다.

하지만 TPSP에서는 우선순위를 매개 변수로 받을 필요가 없으므로 해당 코드를 함수에서 삭제한다.

<코드 2>는 OSMutexCreate 함수에서 ECB의 멤버변수인 OSEventCnt에 대한 초기화 코드가 변경되었음을 보여준다.

기존의 우선순위 예약을 사용하는 방법에서는 OSEventCnt

〈코드 1〉 변경된 OSMutexCreate 함수

```
OS_EVENT  *OSMutexCreate (INT8U *perr){
/*
1. 우선순위를 받은 매개변수 제거
2. 우선순위를 할당 받아 ECB에 삽입하는 부분 제거
3. OSEventCnt의 초기화, <코드 2>참고
*/
}
```

〈코드 2〉 OSEventCnt 초기화

```
pevent->OSEventCnt    =    (INT16U)((INT16U)OS_ASCII_NUL << 8) | OS_MUTEX_AVAILABLE;
```

상위 8비트에 예약된 우선순위를 저장했지만, TPSP에서는 필요가 없다. 대신 상위 8비트 공간을 우선순위 복구의 용도로 사용한다. 예를 들어, 우선순위 역전현상이 발생하면 뮤텍스를 소유한 태스크의 원래 우선순위를 상위 8비트에 옮겨 놓고 하위 8비트의 우선순위 값을 변경한다. 그 후

뮤텍스를 소유한 태스크가 뮤텍스를 다 사용하게 되면 상위 8비트에 저장된 우선순위 값으로 자신의 우선순위를 되돌려 놓는다.

3.4.2 OSMutexDel() 함수

<코드 3>의 OSMutexDel 함수는 공유자원을 사용한 후, ECB를 반환하는 기존의 함수를 일부 수정하였다. OSMutexCreate 함수와 마찬가지로, TPSP는 우선순위 예약을 사용하지 않으므로 기존의 함수 안에 있는 우선순위 예약에 관련된 코드들을 제거한다.

〈코드 3〉 변경된 OSMutexDel 함수

```
OS_EVENT  *OSMutexDel (OS_EVENT *pevent,
                       INT8U opt, INT8U *perr){
/*
미리 할당받은 우선순위를 반환하는 부분 제거
*/
}
```

3.4.3 OSMutex_RdyAtPrio() 함수

<코드 4>의 OSMutex_RdyAtPrio 함수는 TPSP를 적용한 후 원래대로 두 우선순위를 원래 태스크들에게 되돌리기 위해 호출된다. 그 경우에는 두 가지 경우가 존재하는데,

첫 번째, 공유자원을 소유한 태스크가 공유자원에 대한 소유를 반환하였을 경우에 함수가 호출된다.

〈코드 4〉 변경된 OSMutex_RdyAtPrio 함수

```
static  void  OSMutex_RdyAtPrio (OS_TCB *ptcb,
                                 INT8U prio,
                                 OS_TCB* bptcb,
                                 INT8U bprio,
                                 OS_EVENT* pevent){
/*
  1. 매개변수로 받은 두 TCB의 우선순위를 원래의
  우선순위 값으로 복귀한다.
2. OSTCBPrioTbl 값을 서로 교환
*/
}
```

두 번째, 우선순위 역전현상이 중첩하여 발생될 경우에 함수가 호출된다.

이 함수의 경우 변경된 주요 코드는 <코드 5>와 같다.

<코드 5>에서 ptcb와 bptcb는 TCB의 포인터 매개 변수이며, bprio와 prio는 교환되어질 우선순위 값을 나타내는 매개 변수이다. 우선순위 값을 교환하게 되면, 각 우선순위를 가진 TCB의 메모리 주소도 변경되어야 하므로 OSTCBPrioTbl의 주소 값도 변경해야 한다.

〈코드 5〉 변경된 OSMutex_RdyAtPrio 함수

```
ptcb->OSTCBPrio          = bprio;
OSTCBPrioTbl[bprio]      = ptcb;
bptcb->OSTCBPrio         = prio;
OSTCBPrioTbl[prio]       = bptcb;
```

3.4.4 OSMutexPost() 함수

<코드 6>의 OSMutexPost 함수는 공유자원을 소유한 태스크가 그 소유를 반환하기 위해 호출되는 함수이다. 이 함수는 공유자원을 사용가능하도록 설정하는데, 그전에 먼저 소유했던 태스크가 TPSP를 적용했는지 여부를 확인해야 한다. TPSP를 적용해서 우선순위 값이 바뀐 경우라면 OSMutex_RdyAtPrio 함수를 호출하여 원래의 우선순위 값으로 복귀하도록 한다.

<코드 7>은 OSMutex_RdyAtPrio함수 호출 후 ECB의 멤버변수인 OSEventCnt의 상위 8비트 prio 값을 원래대로 하위 8비트로 복귀하는 실제 코드이다.

〈코드 6〉 변경된 OSMutexPost 함수

```
INT8U OSMutexPost( OS_EVENT* pevent){
/*
if(OSEventCnt의 상위 8비트가 0이 아님)
    OSMutex_RdyAtPrio 함수를 호출하여 우선순위를 복귀
    OSEventCnt의 하위 8비트의 값을 상위 8비트의 값으로 설정
    OSEventCnt의 상위 8비트의 값을 OS_ASCII_NUL 으로 설정
*/
}
```

3.4.5. OSMutexPend() 함수

<코드 8>의 OSMutextPend 함수는 태스크가 공유자원을 요청할 때 호출하는 함수이다. 먼저, 함수는 요청한 공유자원을 다른 태스크가 소유하고 있는지를 검사한다.

<코드 9>는 실제 검사 코드로서 OSEventCnt의 하위 8

〈코드 7〉 OSEventCnt 복귀

```
pevent->OSEventCnt    =    (INT16U)((INT16U)OS_ASCII_NUL << 8) | prio;
```

〈코드 8〉 변경된 OSMutexPend 함수

```
void OSMutexPend(OS_EVENT* pevent,
                 INT16U timeout,
                 INT8U* err){
/*
if(뮤텍스가 사용 가능 하다면, <코드 9>참고)
    1. 뮤텍스를 할당
    2. OSEventCnt의 하위 8비트 = 호출 태스크의 우선
       순위로 설정
else if(우선순위 역전현상이 발생 <코드 10> 참고)
    if(OSEventCnt의 값이 0이 아님)
        a. OSMutex_RdyAtPrio 함수를 호출하여
           원래의 우선순위로 복귀
        b. OSEventCnt의 하위 8비트의 값을 상위
           8비트의 값으로 설정
        c. OSEventCnt의 상위 8비트의 값을
           OS_ASCII_NUL 으로 설정
    1. 우선순위를 서로 교환
    2. OSEventCnt의 상위 8비트를 하위 8비트의
       값으로 설정
    3. OSEventCnt의 하위 8비트의 값을 호출한 태
       스크의 새로운 우선순위 값으로 변경
    4. 각 OSTCBPrioTbl 값을 서로 교환

OS_EventTaskWait 함수 호출
*/
}
```

〈코드 9〉 뮤텍스가 사용가능한지 검사

```
if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8)
   == OS_MUTEX_AVAILABLE) {
    pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
    pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
    ...
```

비트를 추출해서 그 값이 0x00FF 값 즉, OS_MUTEX_AVAILABLE이라면 소유하고 있지 않은 경우이므로 공유자원을 할당하고 함수를 빠져 나온다. 하지만 이미 다른 태스크가 공유자원을 소유한 경우, 요청한 태스크의 우선순위와 공유자원을 소유한 태스크의 우선순위를 비교해야 한다.

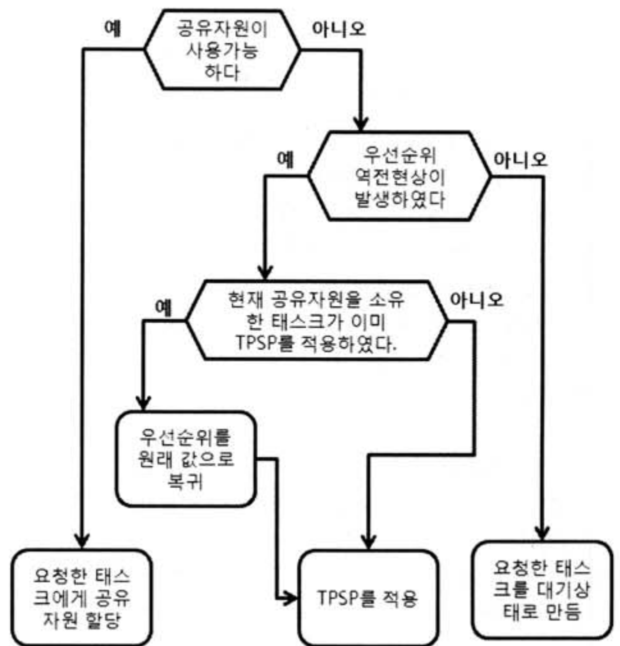
<코드 10>의 첫 if문장은 뮤텍스를 요청한 태스크와 소유한 태스크간의 우선순위를 서로 비교하는 코드이다.

만약 요청 태스크 OSTCBCur의 우선순위 OSTCBPrio가 공유 자원을 소유한 태스크의 우선순위 mprio보다 작다면, 우선순위 역전현상이 아니므로 요청한 태스크를 대기상태로 만든 후 함수를 빠져 나온다.

그 반대의 경우라면, TPSP를 적용하여 우선순위 역전현상을 방지해야 한다. 하지만, TPSP를 적용하기 전에 먼저 검사해야 할 것이 있다. 우선순위 역전 현상이 중첩되어 발생했는지 여부를 검사하는 것인데, 이를 위해 <코드 10>의 두 번째 if 문장처럼 공유 자원을 소유한 태스크가 이미 TPSP를 적용하여 우선순위 값이 바뀌었는지를 보면 된다. pip 값은 OSEventCnt의 상위 8비트를 나타내며, 초기값은 OS_ASCII_NUL이다.

〈코드 10〉 우선순위 비교 및 중첩여부 비교

```
if (mprio > OSTCBCur->OSTCBPrio) {
    if(pip != OS_ASCII_NUL && pip != mprio){
```



(그림 5) 변경된 OSMutexPend 함수의 순서도

만약 TPSP로 인해 우선순위가 변경될 경우, 변경되기 전에 원래 우선순위 값을 일시적으로 저장한다.

pip값이 OS_ASCII_NUL이 아니라면, 즉 이미 TPSP가 적용된 경우 OSMutex_RdyAtPrio함수를 호출하여 원래의 우선순위로 복귀해야 한다. 그 후 새로 뮤텍스를 요청한 태스크와 TPSP를 적용 한다.

마지막으로, TCB주소를 저장하는 OSTCBPrioTbl에서 두 태스크의 TCB주소 값을 교환한다.

(그림 5)는 OSMutextPend 함수 내에서의 처리 과정을 도식화한 그림이다.

4. 동작 실험 및 결과 분석

4.1 실험 환경

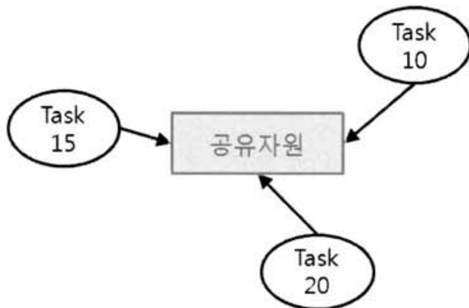
본 실험은 TPSP가 우선순위 예약을 하지 않고도 올바른 동작을 수행하는지 테스트한다. 커널 상에서 변화된 부분만을 테스트하는 것이 순수 목적이므로, 실제 임베디드 장비에 포팅하지 않고 Windows XP 운영 체제상에서 가상 머신으로 동작되도록 프로그램으로 시뮬레이션 하였다.

- 커널 버전 : MicroC/OS-II V2.85
- 개발 도구 : Visual Studio 6.0

- 실행 환경 :
  CPU : Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz
  RAM : 1GB
  MicroC/OS-II and Microsoft Windows XP

4.2. 실험 상황

(그림 6)에서와 같이 3개의 태스크를 생성하고, 이들 태스크들이 우선순위에 의해 실행되도록 한다. 여기서 Task 10은 우선순위가 10인 태스크를 나타내며 가장 높은 우선순위를 가지는 태스크이다. 실험에서는 우선순위 역전 현상이 발생되도록 하기 위해, Task 20이 가장 먼저 뮤텍스를 요청하고 할당 받는다. 그 후 Task 15가 뮤텍스를 요청하고, 마지막으로 Task 10이 뮤텍스를 요청한다.



(그림 6) 공유자원에 접근하는 3개의 태스크

4.3 소스 코드의 작성

4.3.1 태스크와 뮤텍스의 생성

<코드 11>에서와 같이 OSMutexCreate 함수를 호출하여 뮤텍스를 할당한다. ResourceMutex는 ECB 구조체의 포인터 변수로서 할당된 번지를 저장한다. 그 후, 3개의 OSTaskCreateExt 함수를 호출하여 태스크를 생성한다.

함수의 인자들 중, 첫 번째 인자는 함수 포인터로서, 실제 태스크가 수행될 함수를 지정한다. 네 번째 인자는 태스크의 우선순위를 가리킨다. 각 태스크는 10, 15, 20의 우선순위를 갖게 된다. 각 태스크는 생성되는 순간 준비리스트에 등록되어 실행을 기다리게 된다.

4.3.2 태스크의 수행

각 태스크의 함수 내부 코드는 OSMutexPend 함수를 호출하여 뮤텍스를 요청하고, OSMutexPost 함수를 호출하여 할당받은 뮤텍스를 반납하는 과정이 동일하다. 하지만, OSTimeDlyHMSM 함수를 사용하여 뮤텍스를 할당받는 시점은 서로 다르게 작성하였다. 설명의 편의상, 태스크 생성 시 우선순위가 10, 15, 20인 태스크를 각각 태스크 10, 태스크 15, 태스크 20으로 명명하였다.

<코드 12>는 20의 우선순위를 가진 태스크를 생성할 때 첫 번째 인자로 지정한 TaskPrio20함수의 내부 코드이다.

<코드 12>에서처럼, 우선순위가 가장 낮은 태스크 20이 실행되자마자 뮤텍스를 요청하고, 사용하게 된다. 그 후 30초의 시간을 지연시킨 후, OSMutexPost 함수를 통해 뮤텍스를 반환한다.

<코드 13>은 15의 우선순위를 가진 태스크를 생성할 때, 첫 번째 인자로 지정한 TaskPrio15함수의 내부 코드이다.

<코드 13>에서처럼 태스크 15는 시작과 동시에 0.1초간 대기한다. 그 후, 뮤텍스를 요청하는데 이미 태스크 20이 태스크를 소유하였기 때문에 TPSP를 적용한다. TPSP는 태스크 20과 태스크 15의 우선순위를 교환한다.

<코드 14>는 10의 우선순위를 가진 태스크를 생성할 때, 첫 번째 인자로 지정한 TaskPrio10함수의 내부 코드이다.

〈코드 12〉 태스크 20의 수행

```
for ( ; ; ){
        OSMutexPend(ResourceMutex, 0, &err);
        OSTimeDlyHMSM(0, 0, 30, 0);
        OSMutexPost(ResourceMutex);
}
```

〈코드 13〉 태스크 15의 수행

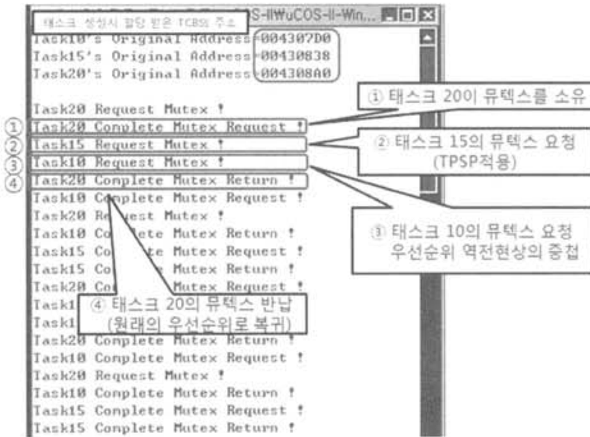
```
for ( ; ; ){
        OSTimeDlyHMSM(0, 0, 0, 100);
        OSMutexPend(ResourceMutex, 0, &err);
        OSTimeDlyHMSM(0, 0, 2, 0);
        OSMutexPost(ResourceMutex);
}
```

〈코드 11〉 main 함수 내의 태스크와 뮤텍스 생성

```
ResourceMutex= OSMutexCreate(&err);

OSTaskCreateExt(TaskPrio10, (void *)0, (OS_STK *)&TaskPrio10Stk[TASK_STK_SIZE-1], 10, 10, (OS_STK *)&TaskPrio10Stk[0],
        TASK_STK_SIZE, (void *)0, OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

OSTaskCreateExt(TaskPrio15, (void *)0, (OS_STK *)&TaskPrio15Stk[TASK_STK_SIZE-1], 15, 15, (OS_STK *)&TaskPrio15Stk[0],
        TASK_STK_SIZE, (void *)0, OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

OSTaskCreateExt(TaskPrio20, (void *)0, (OS_STK *)&TaskPrio20Stk[TASK_STK_SIZE-1], 20, 20, (OS_STK *)&TaskPrio20Stk[0],
        TASK_STK_SIZE, (void *)0, OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
```

〈코드 14〉 태스크 10의 수행

```
for ( ; ; ){
        OSTimeDlyHMSM(0, 0, 3, 0);
        OSMutexPend(ResourceMutex, 0, &err);
        OSMutexPost(ResourceMutex);
}
```



(그림 7) TPSP 방식의 수행 결과

<코드 14>에서처럼, TPSP가 적용된 태스크 20(우선순위는 15)이 뮤텍스를 반환하기 전에 태스크 10이 뮤텍스를 요청한다. 이때에도 태스크 20과 태스크 10의 우선순위가 교환되어야 한다. 하지만, 이미 태스크 20의 우선순위가 TPSP에 의해 변경되었으므로 태스크 20은 원래의 우선순위로 복귀해야 한다. 즉 태스크 15와 태스크 20의 우선순위가 원래 상태로 복귀된 후, 태스크 10과 태스크 20의 우선순위가 TPSP에 의해서 교환된다.

구현된 TPSP 방식으로, (그림 7)은 위의 응용프로그램을 수행한 결과를 보여준다.

4.4 성능 평가 및 분석

4.3절의 응용프로그램을 포함한 전체 소스를 컴파일하고, 그 결과로 생성된 Object 파일과 Binary 파일을 비교하고자 한다.

<표 1>에서처럼 뮤텍스 서비스 Object 파일은 기존의 BPI를 적용한 것보다 약 3KB가, 응용프로그램을 포함한 전체 Binary 파일은 약 12KB가 감소되었다. 크기 감소의 주된 원인은 우선순위 예약을 구현하기위해 사용된 모든 변수와 명령들이 제거되었기 때문이다.

<표 2>는 우선순위 예약을 사용한 기존 코드와 제안 프로토콜인 TPSP의 수정 코드의 길이를 보여준다.

<표 2>에서처럼 OSMutex_RdyAtPrio 함수를 제외한 나머지 함수의 코드 수가 감소되었다. 기존 함수들의 경우, 뮤텍스에 할당된 우선순위 값에 대한 사용가능 여부 체크와 할당에 대한 코드가 제거되었기 때문이다. 반면, OSMutex_RdyAtPrio 함수의 경우는 코드가 증가 하였다.

〈표 1〉 코드 크기에 대한 프로토콜 비교

크기 \ 프로토콜	기존 프로토콜 (BPI)	제안 프로토콜 (TPSP)
뮤텍스 서비스 (Object 파일)	20,305바이트 (19.8KB)	17,221바이트 (16.8KB)
응용 프로그램 포함 전체 크기 (Binary 파일)	209,047바이트 (204KB)	196,740바이트 (192KB)

〈표 2〉 기존 코드와 수정 코드 간 비교

함수 \ 코드 길이	기존 프로토콜 (BPI)	제안 프로토콜 (TPSP)
OSMutexCreate	39	32
OSMutexDel	89	87
OSMutexPend	116	114
OSMutexPost	47	40
OSMutex_RdyAtPrio	20	40
전체	311	313

OS_Mutex_RdyAtPrio 함수는 원래의 우선순위로 복귀할 때 호출되는 함수로, 위의 두 프로토콜에서의 역할이 동일하다. 다만, 기존의 함수에서는 상속받은 태스크 하나에 대해서만 우선순위 복귀가 이루어지고 수정된 함수에서는 우선순위가 교환된 두 태스크간에 복귀가 이루어지기 때문에 코드가 증가하는 것이다.

즉, 우선순위 역전현상이 중첩적으로 발생했을 때의 두 프로토콜간의 시간 복잡도를 비교 분석할 필요가 있다.

두 프로토콜간 분석을 하자면, 중첩상황에 대한 최악의 경우를 고려해야한다. MicroC/OS-II에서의 최악의 경우는, 아래의 상황이 모두 충족되었을 때 발생한다.

1. N개의 태스크중 가장 우선순위가 낮은 태스크가 먼저 뮤텍스를 소유한다.
2. N-1개의 태스크들이 우선순위가 낮은 순서로 순차적으로 뮤텍스를 요청한다.
3. 처음 뮤텍스를 소유한 태스크는 최소한 가장 높은 우선순위를 가지는 태스크가 뮤텍스를 요청할 때까지 뮤텍스를 소유한다.

이런 상황이라면 우선순위 역전현상은 N-1번 발생하게 된다. 기존 프로토콜에서의 우선순위 상속과 복귀의 시간 복잡도는 <표 3>과 같다. 뮤텍스를 소유한 태스크가 예약된 우선순위 값을 상속받고, 원래의 우선순위로 복귀할 때 걸리는 수행 시간을 각각 t라고 가정하였다.

예를 들어, 4개의 태스크 T1, T2, T3, T4가 있고, 각 태스크의 우선순위는 T1>T2>T3>T4일 때 최악의 시간 복잡

〈표 3〉 기존 프로토콜(BPI)

O(n) = {nt; n=태스크 개수, t=상속에 걸리는 시간}

〈표 4〉 제안 프로토콜(TPSP)

O(n) = {2(n-1)t; n=태스크 개수, t=교환시간}

〈표 5〉 기존 프로토콜과 제안 프로토콜 간 비교

프로토콜 / 비교 기준	기존 프로토콜 (BPI)	제안 프로토콜 (TPSP)
시간 복잡도	O(n)	O(n)
공간 복잡도	O(1)	O(1)

도를 분석해보겠다. T4가 뮤텍스를 소유하였다. 이 때는, 우선순위를 상속받고 복귀하는데 4t의 시간이 소요된다.

1. T3가 뮤텍스를 요청. BPI 적용. t시간소요
2. T2가 뮤텍스를 요청. BPI 적용. t시간소요
3. T1이 뮤텍스를 요청. BPI 적용. t시간소요
4. T4가 뮤텍스를 반납. T4 우선순위 복귀. t시간소요

TPSP에서의 우선순위 교환과 복귀의 시간 복잡도는 <표 4>와 같다. TPSP의 경우도 마찬가지로, 우선순위 교환, 복귀하는데 걸리는 시간을 각각 t로 가정하였다.

예를 들어, 4개의 태스크 T1, T2, T3, T4가 있고, 각 태스크의 우선순위는 T1>T2>T3>T4일 때 최악의 시간 복잡도를 분석해보겠다. T4가 뮤텍스를 소유하였다. 이 때는, 우선순위를 상속받고 복귀하는데 6t의 시간이 소요된다.

1. T3가 뮤텍스를 요청. T3, T4간 우선순위 교환
2. T2가 뮤텍스를 요청. T3, T4의 우선순위 복귀
3. T2와 T4간 우선순위 교환
4. T1이 뮤텍스를 요청. T2, T4간 우선순위 복귀
5. T1와 T4간 우선순위 교환
6. T4의 뮤텍스 반납. T1, T4간 우선순위 복귀

기존 프로토콜과 본 논문에서 제안한 프로토콜 간 비교한 내용을 <표 5>에서 보인다. 비교기준으로 n은 태스크의 개수를 의미한다. 시간 복잡도 및 공간 복잡도에 따른 추가 비용에 대해서 제안된 TPSP 프로토콜은 BPI 프로토콜과 동일한 복잡도를 나타낸다. 공간 복잡도의 경우 특별한 자료 구조나 메모리 공간을 소요하지 않음으로써, O(1)이다.

5. 결론 및 향후 과제

우선순위 역전현상을 해결하는 몇 가지 프로토콜을 살펴보았다. 그리고 실시간 운영체제인 MicroC/OS-II에서 우선순위 역전현상 해결을 하기위해 사용하는 BPI에 기초한 우선순위 예약 방식을 살펴보았다. 이 방식은 한 번에 한 개의 자원 소유가 가능한 MicroC/OS-II가 가지는 특성으로 인해 교착상태를 방지할 수 있다. 반면에, 공유자원의 수에 비례하는 우선순위의 낭비를 초래한다.

그러므로 MicroC/OS-II와 같은 특성을 가지는 경우에 적용될 수 있는 새로운 기법인 TPSP를 제안하게 되었다. TPSP를 적용함으로써 공유자원에 의한 교착상태를 방지할 수 있고, 추가의 우선순위를 미리 할당할 필요가 없다. 그러므로 기존의 BPI 프로토콜에서 발생되었던 우선순위 낭비를 제거할 수 있게 되었다.

향후 연구에는 우선순위 중복 불가능한 태스크의 문제 해결과 동시에 여러 자원의 소유 가능한 태스크 생성, 보다 향상되고 이식성 높은 우선순위 역전현상 해결 알고리즘을 적용하는 것이 필요할 것이다.

참 고 문 헌

[1] J. Stankovic, "Real-Time Computing", BYTE, invited paper, pp.155-160, August, 1992.

[2] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers Vol.39, No.9, pp.1175-1185, Sep., 1990.

[3] 도유환, 박명진, 오삼권, "실시간 커널 μC/OS의 최대 허용 태스크 개수의 확장", 한국정보과학회 추계 학술대회, pp.152-154, 1999.

[4] Jean J. Labrosse, "MicroC/OS-II 실시간 커널 2판", 에이콘, 2007, ISBN 89-89975-60-3.

[5] J. B. Goodenough and L. Sha, "The Priority Ceiling Protocol", Special Report CMU/SEI-88-SR-4, Mar. 1998

[6] Qing Li, "RTOS를 이용한 실시간 임베디드 시스템 디자인", 에이콘, 2005, ISBN 89-89975-42-5.

[7] 김주용, 고건, "개선된 우선순위 상한 프로토콜의 구현", 한국정보과학회 가을 학술발표논문집 제 20권 제2호 pp.565- 568, 1993.

[8] Min-Ih Chen, and Kwei-Jay Lin, "Dynamic Priority Ceilings: A concurrency Control Protocol for Real-Time Systems", The Journal of Real-Time Systems, 2, 1990, pp.325-346.

[9] 강성구, 경계현, 고광선, 엄영익. "실시간 운영체제의 우선순위 역전현상 해결을 위한 프로토콜 설계 및 구현". 한국정보처리학회, v.13A, No.5, pp.405-412, 2006.

[10] Jae-Ho Lee, Heung-Nam Kim, "Implementing priority inheritance semaphore on uC/OS real-time kernel," wstfes, pp.83, IEEE Workshop on Software Technologies for Future Embedded Systems, 2003.

[11] 김태호, 김창수, "우선순위 기반의 μC/OS-II 커널에서 확장된 R/R 스케줄링 연구", 한국멀티미디어학회, 2002.

전 영 식

e-mail : klocust@hanyang.ac.kr

2007년 한국방송통신대학교 컴퓨터과학과(학사)

2008년~현 재 한양대학교 컴퓨터공학과 석사과정

관심분야 : 시스템소프트웨어, 임베디드 시스템 등

김 병 곤

e-mail : bkkim@kict.re.kr

1991년 한양대학교 컴퓨터공학과(학사)

1993년 한양대학교 컴퓨터공학과(석사)

2003년 한양대학교 컴퓨터공학과(박사)

1993년~현 재 한국건설기술연구원 건설정보연구실 선임연구원

관심분야 : 운영체제, 무선센서네트워크(WSN), 건설정보화

허 신

e-mail : shinheu@hanyang.ac.kr

1973년 서울대학교 전기공학과(학사)

1979년 미국 University of Southern California 전산학(석사)

1986년 미국 University of South Florida 전산학(박사)

1980년~1986년 미국 University of South Florida 연구원보

1986년~1988년 미국 The Catholic University of America 조교수

1988년~현 재 한양대학교 컴퓨터공학과 교수

관심분야 : 분산컴퓨팅, 결함허용시스템, 실시간운영체제 등