실시간 운영체제의 우선순위 역전현상 해결을 위한 프로토콜 설계 및 구현

강 성 구[+] · 경 계 현[+] · 고 광 선[++] · 엄 영 익[+++]

요 약

실시간 운영체제는 정해진 시간 내에 작업처리를 완료해야 하는 분야에 주로 사용되고 있으며, 최적의 실시간 운영체제를 설계하고 개발하기 위해서는 효과적인 스케줄링 정책, 인터럽트 지연 최소화, 우선순위 역전현상 해결 등의 조건을 만족시켜야한다. 이러한 조건들 중에서 우선순위 역전현상을 해결하기 위해 지금까지 basic priority inheritance 프로토콜, priority ceiling emulation 프로토콜 등이 제안되었으나, 하나의 프로세스가 동시에 다수의 자원을 소유하는 경우 또는 재귀적으로 자원을 소유하거나 요청하는 경우와 같이 대표적인 두 가지 복잡한 우선순위 역전현상에 대해서는 해결이 불가능하다. 이에 본 논문에서는 재귀적 자료구조를 기반으로 다양한 우선순위 역전현상을 효과적으로 해결할 수 있는 RPI(Recursive Priority Inheritance) 프로토콜을 설계하고, 이를 리눅스 커널에 구현하여 검증한 내용을 보인다.

키워드 : 실시간 운영체제, 우선순위 역전현상

Design and Implementation of a Protocol for Solving Priority Inversion Problems in Real-time OS

Seong-Goo Kang[†] · Gyehyeon Gyeong[†] · Kwangsun Ko[++] · Young Ik Eom[+++]

ABSTRACT

Real-time operating systems have been used in various computing environments, where a job must be completed in its deadline, with various conditions, such as effective scheduling policies, the minimum of an interrupt delay, and the solutions of priority inversion problems, that should be perfectly satisfied to design and develop optimal real-time operating systems. Up to now, in order to solve priority inversion problems among several those conditions. There have been two representative protocols: basic priority inheritance protocol and priority ceiling emulation protocol. However, these protocols cannot solve complicated priority inversion problems. In this paper, we design a protocol, called recursive priority inheritance (RPI), protocol that effectively solves the complicated priority inversion problems. Our proposed protocol is also implemented in the Linux kernel and is compared with other existing protocols in the aspect of qualitative analysis.

Key Words : Real-time Operating Systems, Priority Inversion Problem

1. 서 론

현재 우주탐사선, 미사일 등과 같이 정해진 시간 내에 작업처리를 완료해야하는 분야에 다방면으로 사용되고 있는 실시간 운영체제는 효과적인 스케줄링 정책, 인터럽트 처리 지연 최소화, 우선순위 역전현상 해결 등의 조건을 만족시켜야한다. 이 중에서 우선순위 역전현상이란 자원 공유 문제로 인하여 높은 우선순위를 가진 작업이 낮은 우선순위를 가진 작업에 의해 대기되는 현상을 뜻한다. 이 현상은 실행 시 자주 발생하는 문제는 아니지만, 우주탐사선이나 미사일과 같이 몇 달 또는 몇 십 년씩 재가동 없이 동작하는 시스템의 경우 단 한 번의 오류가 시스템에 중요한 영향을 끼치게 되므로 반드시 해결해야 하는 문제이다.

이러한 현상을 해결하기 위하여 지금까지 BPI(Basic Priority Inheritance) 프로토콜이나 PCE(Priority Ceiling Emulation) 프로토콜 등의 기법들이 제안되었지만 추가비용이 많이 들고, 자원을 동시에 여러 개 소유하거나 자원을 서로 재귀적으로 공유하는 상황 등에서는 우선순위 역전현상에 대해서는 해결이 되지 않는 등의 문제가 존재하기 때

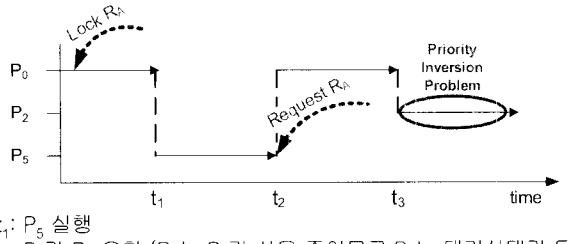
※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학IT 연구센터 육성지원 사업의 결과로 수행되었음(IITA-2006-C1090-0603-0027).
† 준 회 원 : 성균관대학교 대학원 컴퓨터공학과 석사과정
†† 준 회 원 : 성균관대학교 대학원 컴퓨터공학과 박사과정
††† 종신회원 : 성균관대학교 정보통신공학부 교수
논문접수 : 2006년 6월 23일, 심사완료 : 2006년 9월 8일

문에 일반적으로 다른 기법들과 혼용하여 우회적으로 해결하곤 한다. 이에 본 논문에서는 재귀적 자료구조를 이용하여 기존의 기법보다 더 효과적으로 우선순위 역전현상을 해결할 수 있는 프로토콜인 RPI(Recursive Priority Inheritance) 프로토콜을 설계하고, 이를 리눅스 커널에 구현하여 검증한 내용을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 우선순위 역전현상과 기존에 제안된 프로토콜들에 대해 설명하고, 3장에서는 기존 프로토콜들의 문제점들과 복잡한 우선순위 역전현상을 해결하기 위하여 재귀적 자료구조를 기반으로 한 RPI 프로토콜의 설계내용을 보인다. 4장에서는 제안 기법에 대해 리눅스 커널에 구현한 내용을 보이고, 5장에서는 구현한 내용에 대해 실험 및 검증을 한다. 마지막 6장에서는 결론 및 향후 연구계획을 설명한다.

2. 관련 연구

우선순위 역전현상이란 실시간 운영체제에서 우선순위가 낮은 프로세스가 사용 중인 자원을 우선순위가 높은 프로세스가 요청하였으나, 해당 자원을 할당받지 못하고 대기하게 되는 현상을 의미한다. 이에 대한 구체적인 동작과정을 (그림 1)에서 보인다. 편의상 프로세스와 자원은 각각 P과 R로 표시하며, 프로세스들 간에는 프로세스 번호가 낮을수록 우선순위가 낮다고 가정한다. 즉, P_0가 P_1보다 우선순위가 낮다.



t_1: P_5 실행
t_2: P_5가 R_A 요청 (R_A는 P_0가 사용 중이므로 P_5는 대기상태가 됨)
t_3: P_2 실행

(그림 1) 기본적인 우선순위 역전현상

(그림 1)에서 보이는 바와 같이, P_0이 R_A를 소유한 상태에서 t_2 시점에 P_5가 실행하여 R_A를 요청하면, P_5는 R_A가 반환될 때까지 대기 상태가 된다. t_3 시점에 P_2가 실행되면, P_0가 선점되기 때문에 P_5는 P_2에 의해 실행되지 못한다. 따라서 P_5는 P_2 때문에 실행되지 못하며, R_A를 얻기 위한 대기시간이 부정확하기 때문에 실시간 운영체제에서는 치명적인 문제가 된다[1].

2.1 BPI 프로토콜

BPI 프로토콜은 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 방식으로 동작함으로써 우선순위 역전현상을 해결하는 프로토콜이다[2-5]. 하지만 이 프로토콜은 단순한 우선순위 역전현상만 해결이 가능하기 때문에 단독으로 사용될 수 없는 단점이 있다[2, 4, 5]. 이에 대한 자세한 설명은 3장에서 보이도록 한다.

2.2 PCE 프로토콜

PCE 프로토콜은 실시간 운영체제가 초기화될 때, 임의의 자원의 우선순위를 해당 자원을 사용할 프로세스 중에서 가장 높은 우선순위를 가지는 프로세스보다 높게 설정하고, 해당 자원을 소유하는 프로세스는 자원의 우선순위를 상속받게 함으로써, 다른 프로세스에 의해 선점되지 않도록 하는 방식으로 동작함으로써 우선순위 역전현상을 해결하는 프로토콜이다[2, 5]. 하지만 이 프로토콜은 프로세스가 자원을 요구하지 않는 상황에서도 항상 가장 높은 우선순위를 할당받기 때문에 불필요한 성능저하가 발생한다[6-8].

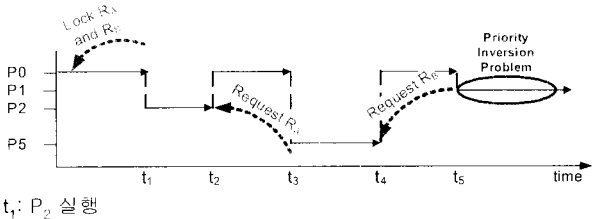
3. RPI 프로토콜 설계

본 장에서는 기존 기법들이 해결하지 못하는 대표적인 두 가지 형태의 복잡한 우선순위 역전현상을 보이고, 이를 해결하기 위한 RPI 프로토콜의 설계 내용을 보인다.

3.1 기존 기법의 문제점

기존에 제시된 기법에서는 (그림 2)와 (그림 3)이 보이는 바와 같이 대표적인 두 가지 형태의 복잡한 우선순위 역전현상을 해결할 수 없다. (그림 2)는 P_0이 두 개의 R_A와 R_B를 소유한 상태에서 우선순위가 높은 P_2와 P_5가 각각 R_A와 R_B를 요청하는 경우를 보이며, (그림 3)은 P_0와 P_2가 각각 R_A와 R_B를 소유한 상태에서 P_2가 R_A를 요청하여 대기 상태가 되고, P_5가 R_B를 요청하는 경우를 보인다.

BPI 프로토콜은 두 개 이상의 임계 영역에 대해서는 고려하지 않기 때문에 (그림 2)와 (그림 3)에서 보이는 우선순위 역전현상을 해결하지 못한다[4]. BPI 프로토콜을 (그림 2)의 상황에 적용할 경우, R_A를 소유한 P_0은 P_2가 R_A를 요청할 때, P_2의 우선순위를 상속받는다. 하지만 그 후에 P_5



t_1: P_2 실행
t_2: P_2가 R_A 요청 (R_A는 P_0이 사용 중이므로 P_2는 대기상태가 됨)
t_3: P_5 실행
t_4: P_5가 R_B 요청 (R_B는 P_0이 사용 중이므로 P_5는 대기상태가 됨)
t_5: P_1 실행

(그림 2) 자원을 동시에 여러 개 소유한 경우의 우선순위 역전현상

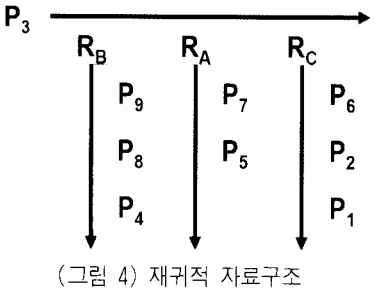
가 R_B를 요청했을 때, P_3의 권한을 상속받을 수 없다. 만약 P_3의 우선순위를 상속받을 경우, 이후에 P_0이 R_B를 먼저 반환할 때 P_2의 권한을 다시 상속받아야 하지만 BPI 프로토콜은 이러한 경우를 처리할 수 없다. 그리고 BPI 프로토콜을 (그림 3)의 상황에 적용할 경우, P_0은 P_2가 R_A를 요청하면서 P_2의 권한을 상속받고, P_2는 P_3가 R_B를 요청하면서 P_3의 권한을 상속받는다. 따라서 P_3는 P_2에 의해 대기되고, P_2는 P_0에 의해 대기되고 P_0은 P_2의 권한을 상속받은 상태이므로 우선순위 역전현상이 발생한다.

그리고 PCE 프로토콜은 (그림 2)가 보이는 형태의 우선순위 역전현상은 해결이 가능하지만, 한 프로세스가 직접적으로 요청하고 사용하는 자원의 권한만을 상속받기 때문에 (그림 3)이 보이는 바와 같은 재귀적인 형태의 우선순위 역전현상은 해결이 불가능하다. PCE 프로토콜을 (그림 3)의 상황에 적용할 경우, R_A를 요청하는 P_2와 P_3는 최상위 권한인 P_3의 권한을 가지게 되지만 P_3는 P_2에 의해 대기되고 P_2는 P_0에 의해 대기된다. 하지만 P_0은 R_B만을 사용하므로 P_2의 권한을 가지게 돼서 우선순위 역전현상이 발생한다. 그리고 불필요한 경우에도 우선순위 상속이 발생해서 실행 시 성능이 떨어지는 단점이 있다[9]. 이러한 이유는 기본적으로 우선순위 역전현상은 실시간 운영체제의 모든 프로세스와 자원들 사이에서 재귀적인 형태로 발생하지만, 기존에 제시된 기법들은 이러한 재귀적인 관계를 고려하지 않았기 때문이다.

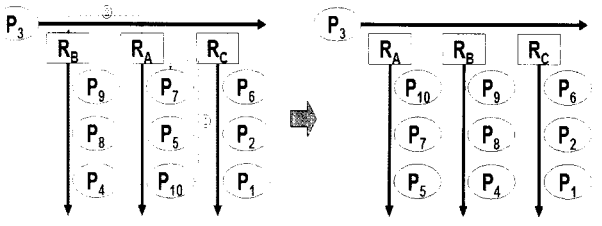
3.2 재귀적 자료구조와 RPI 프로토콜

재귀적 자료구조란 각 프로세스가 획득한 자원 및 그 자원할당을 대기 중인 프로세스에 대한 정보를 유지하는 자료구조로 (그림 4)에서 보인다. 본 논문에서 제안하는 RPI 프로토콜은 각 프로세스마다 이러한 재귀적 자료구조를 유지함으로써 다양한 우선순위 역전현상을 해결하는 프로토콜이다.

(그림 4)에서 보이는 바와 같이, P_3의 재귀적 자료구조는 R_A, R_B, 그리고 R_C를 소유하고 있으며, 각 자원들은 해당 자원을 요청한 프로세스 대기 큐를 가진다. 또한 각 자원별로 대기 중인 프로세스는 우선순위에 의해 정렬이 되어있고, 재귀적 자료구조를 가지는 P_3은 소유 중인 자원들을 각 자원의 첫 번째 대기 프로세스의 우선순위에 의해 정렬한다. 따라서 첫 번째 자원의 첫 번째 대기 프로세스가 항상 가장 높은 우선순위를 갖도록 정렬하고, 이 우선순위를 재귀적



(그림 4) 재귀적 자료구조



(그림 5) R_A를 요청한 P_{10}이 추가되고 재정렬된 재귀적 자료구조

자료구조를 갖는 프로세스가 상속받도록 한다. 즉, P_3은 R_B의 첫 번째 대기 프로세스인 P_9의 우선순위를 상속받게 된다. (그림 4)와 같은 상황에서 임의의 높은 우선순위를 가지는 P_{10}이 R_A를 요청하여 대기 큐에 추가되면, 이후의 자료구조는 (그림 5)와 같이 변경된다.

먼저 P_{10}이 요청한 자원의 대기 큐에 추가되어 프로세스간 재정렬(①)된 후, 첫 번째 프로세스의 우선순위에 의해 자원들 간 재정렬(②)이 이루어진다. 이 경우 시스템 내의 전체 프로세스와 자원의 수를 각각 n과 m으로 가정할 경우, 전체 자료구조는 이미 정렬되어있기 때문에 P_{10}이 추가됨에 따라 P_3의 재귀적 자료구조에서 재정렬에 소요되는 시간 복잡도는 $O(n+m)$이 된다. 만일 (그림 5)의 우측 그림에서 P_3이 소유 중인 자원을 해제할 경우 또는 대기 중인 프로세스의 우선순위가 변경되는 경우에도 (그림 5)에서 보이는 것과 같은 방법으로 재정렬되고, P_3은 새로운 우선순위를 상속받는다.

4. RPI 프로토콜 구현

본 장에서는 3장에서 제안한 RPI 프로토콜을 실제 리눅스 커널에 구현한 내용을 보이며, 이를 위한 구현 환경, 재귀적 자료구조, 그리고 자원의 획득/반환/대기를 위한 함수를 구체적으로 보인다.

4.1 구현 환경

실시간 운영체제를 구현하기 위하여 리눅스 커널 버전 2.6.12에 uCLinux 패치를 적용하였다[10]. 프로세스가 요청하는 자원을 세마포어로 정의하고, 우선순위 역전현상에서 자원 요청 및 해제 작업을 세마포어의 $P()$ 연산과 $V()$ 연산으로 정의한다. 또한 우선순위 조정은 리눅스 커널 내부 함수인 *set_user_nice()* 함수를 이용하여 나이스 값(이하 우선순위)을 조절하는 방식으로 실시한다. 실제 프로세스가 가지는 우선순위를 직접 수정할 수 있지만, 리눅스 커널의 스케줄링이나 여타 정책에 의해 우선순위가 변할 수 있기 때문에, 커널이 제공하는 함수를 수정하여 우선순위를 조절하도록 한다.

4.2 재귀적 자료구조

RPI 프로토콜에서 사용하는 재귀적 자료구조는 리눅스

커널의 *task_struct* 구조체에 세 개의 변수를 추가하고, 세 개의 구조체를 추가함으로써 구현한다. 이러한 자료구조들을 (그림 6)과 (그림 7)에서 보인다.

```
struct task_struct {
    ...
    semaphore_t *resources;
    int restore_prio;
    int blocked_semid;
    ...
};
```

(그림 6) task_struct 구조체에 추가된 변수들 (include/linux/sched.h)

```
struct owned_sem_list {
    pid_t pid;
    int semid;
    struct owned_sem_list *prev, *next;
};

struct process_list {
    struct process_list *prev, *next;
    pid_t pid;
};

struct semaphore_list {
    struct semaphore_list *prev, *next;
    struct process_list *processes;
    int semid;
};

typedef struct process_list process_t;
typedef struct semaphore_list semaphore_t;
```

(그림 7) 우선순위 역전현상 해결을 위해 정의된 재귀적 자료구조 (include/linux/sem.h)

(그림 6)에서 보이는 자료구조는 리눅스에서 프로세스를 생성할 때 호출되는 *copy_process()* 함수에서 초기화된다. *resources* 구조체 변수는 RPI 프로토콜의 핵심인 재귀적 자료구조이며, 해당 프로세스가 할당받은 자원들과 자원할당을 대기하고 있는 프로세스들의 정보가 저장된다. *semaphore_t* 구조체는 (그림 7)에 정의되어 있다. *restore_prio* 변수는 RPI 프로토콜에 의해 프로세스의 우선순위가 변경된 후, 원래의 우선순위로 복귀할 때를 위해 사용되고, *blocked_semid* 변수는 해당 프로세스가 어떠한 자원을 요청하고 대기 중인지를 나타낸다. *blocked_semid* 변수가 필요한 이유는 자원할당을 대기 중인 프로세스의 권한이 변경될 경우, 그 자원을 가진 프로세스의 재귀적 자료구조 내에서 재정렬이 요구되는데, 이 때 그 자원을 가진 프로세스를 찾기 위해서이다. *owned_sem_list* 구조체는 현재 어떤 프로세스가 어떤 자원을 획득하고 있는지에 대한 정보를 저장하는데, 리눅스 커널에서는 자원을 요청하고 대기하고 있는 프로세스를 알아내는 것은 쉽지만, 그 자원을 획득하고 있는 프로세스를 알아내는 것은 어렵기 때문이다. *semaphore_list* 구조체는 *task_struct* 구조체에 들어가 있는 해당 프로세스가 소유한 자원들을 나타내는 자료구조이고, *process_list* 구조체는 해당 자원을 요청하고 대기 중인 프로세스들을 나타내고 있다.

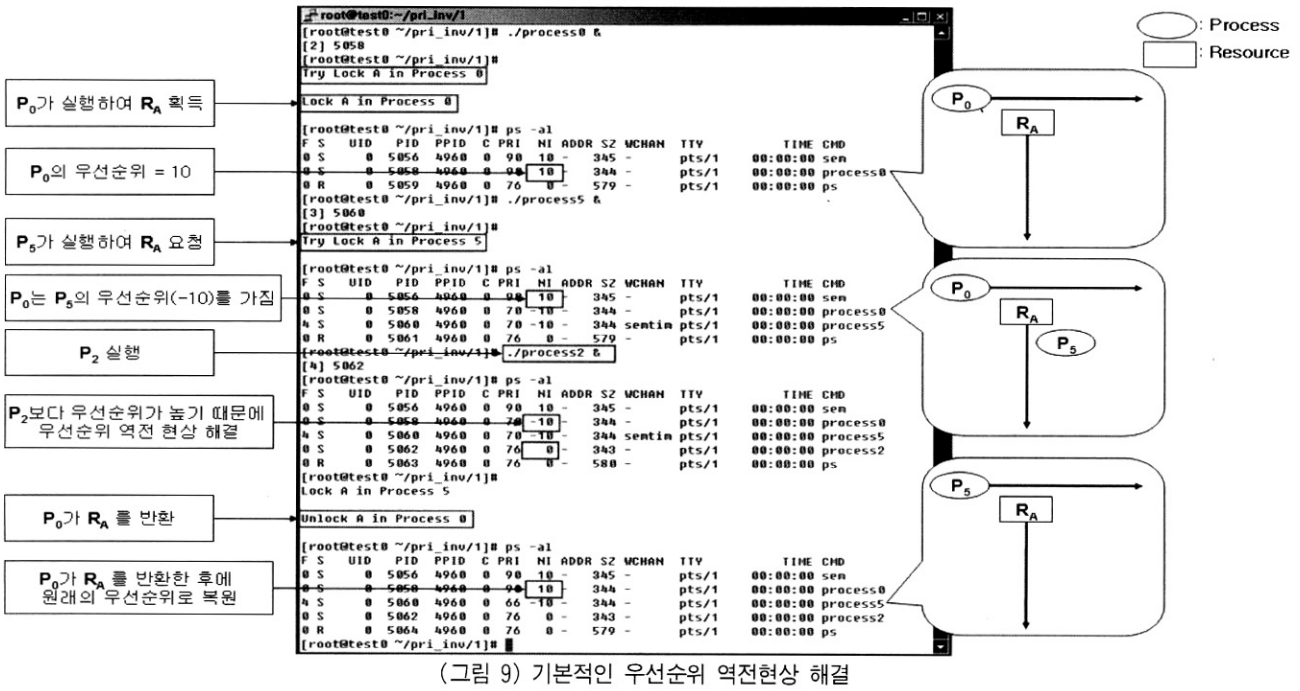
4.3 자원의 획득/반환/대기

리눅스를 포함한 POSIX 지원 유닉스 계열의 운영체제에서는 세마포어의 *P()* 연산과 *V()* 연산을 할 때 *semop()* 시스템 콜을 사용한다. 리눅스 커널에서는 *semop()* 시스템 콜은 *sys_semtimedop()* 함수를 호출하게 되는데 이 함수를 수정하여 *P()* 연산과 *V()* 연산 시에 원하는 작업을 수행하도록 한다. (그림 8)이 수정된 *sys_semtimedop()* 함수이고, 기존 함수에 추가되거나 삭제된 부분은 주석으로 명시하였다.

(그림 8)에서 보이는 바와 같이, 자원의 획득은 *P()* 연산이 수행되어 자원을 획득하여 임계영역에 들어가게 되면, *get_semaphore()* 함수를 통해 해당 프로세스가 해당 자원을 가졌다는 것을 갱신하도록 하고, *insert_semaphore()* 함수를 통해 해당 프로세스의 재귀적 자료구조에 해당 자원을

```
asmlinkage long sys_semtimedop(int semid, struct sembuf __user *tsops,
            unsigned nsops, const struct timespec __user *timeout)
{
...
    error = try_atomic_semop (sma, sops, nsops, un, current->tgid);
    if (error <= 0) {
        if (alter && error == 0) {
            /* 추가된 코드: 자원의 획득과 반환 */
            if (p == 1) { /* P(), 자원의 획득  */
                get_semaphore(current->tgid, semid);
                insert_semaphore(semid);
            }
            if (v == 1) { /* V(), 자원의 반환 */
                release_semaphore(semid);
            }
            update_queue (sma);
        }
        goto out_unlock_free;
    }
    /* 추가된 코드: 자원 요청을 위해 대기하는 함수 */
    insert_process(semid);
    current->blocked_semid = semid;
    queue.sma = sma;
    queue.sops = sops;
    queue.nsops = nsops;
    queue.undo = un;
    queue.pid = current->tgid;
    queue.id = semid;
    queue.alter = alter;
    /* 삭제된 기존 코드
    if (alter)
        append_to_queue(sma ,&queue);
    else
        prepend_to_queue(sma ,&queue);
    */
    queue.status = -EINTR;
    queue.sleeper = current;
    /* 추가된 코드: append_toqueue()를 대체하는 함수 */
    insert_to_queue(sma, &queue);
    current->state = TASK_INTERRUPTIBLE;
...
}
```

(그림 8) 수정된 sys_semtimedop() 함수 (ipc/sem.c)



(그림 9) 기본적인 우선순위 역전현상 해결

추가하게 된다. *get_semaphore()* 역할을 수행하는 함수가 필요한 이유는 임계영역에 들어가기 위해 대기하고 있는 프로세스를 알아내는 것은 쉽지만, 반대로 현재 임계영역 내에 진입해 있는 프로세스를 식별하는 것이 어렵기 때문에, 추가적인 자료구조(*owned_sem_list*)를 이용하여 현재 임계영역 내에 있는 프로세스들의 정보를 유지하도록 하였다. 자원의 반환은 *V()* 연산을 통해 임계영역을 나오게 되면 *release_semaphore()* 함수를 통해 자원의 점유를 해제하게 된다. 이 때 자원을 해제하는 프로세스의 재귀적 자료구조 내에서 재정렬이 일어나고 우선순위가 조정된다. 그리고 해당 자원을 요청하는 프로세스가 있을 경우, 자신이 가지고 있었던 자원의 자료구조인 *semaphore_list* 구조체를 넘기게 된다. 이렇게 함으로써 해당 자원을 요청하여 대기하고 있는 프로세스들의 순서가 그대로 유지된다. 자원의 획득을 위한 대기는 *P()* 연산을 수행하여 임계영역에 들어가지 못하고 대기하게 될 경우, 해당 프로세스는 블록되어 대기 상태가 된다. 그리고 *insert_process()* 함수를 통해서 해당 자원을 소유하고 있는 프로세스의 큐에 삽입이 되고, 그 프로세스 내에서 재정렬 후에 우선순위가 조정된다. 그리고 리눅스 커널 내 해당 자원의 대기 큐에 삽입된다. 기존 리눅스 커널에서는 *P()* 연산 시 *append_to_queue()* 함수를 이용하여 무조건 대기 큐의 마지막에 삽입되고, *P()* 연산 또는 *V()* 연산이 아닌 경우에는 *prepend_to_queue()* 함수를 통해 대기 큐의 맨 앞에 추가된다. 그리고 *V()* 연산 후 *update_queue()* 함수가 대기 큐에서 가장 앞에 있는 프로세스가 임계영역으로 진입할 수 있도록 해당 프로세스를 깨우게 된다. 추가적으로, 본 논문에서 제안한 프로토콜을 구현하기 위해서는 기존의 FIFO 방식의 큐는 수정이 되어야하는데, 재귀적 자료구조 내에서 자원 요청을 하고 대기하는 프로세스들이 우선순위에 따라 정렬되기 때문에 리눅스 커널에서 세마포어 연산을 위해 대기하는 큐의 프로세스 역시 똑같이 우선순위로 정렬이 되어야 한다. 그래서 기존의 *append_to_queue()* 함수 대신에 *insert_to_queue()* 함수를 이용하여 대기 큐 내에 프로세스들이 우선순위로 정렬되어 삽입되도록 한다.

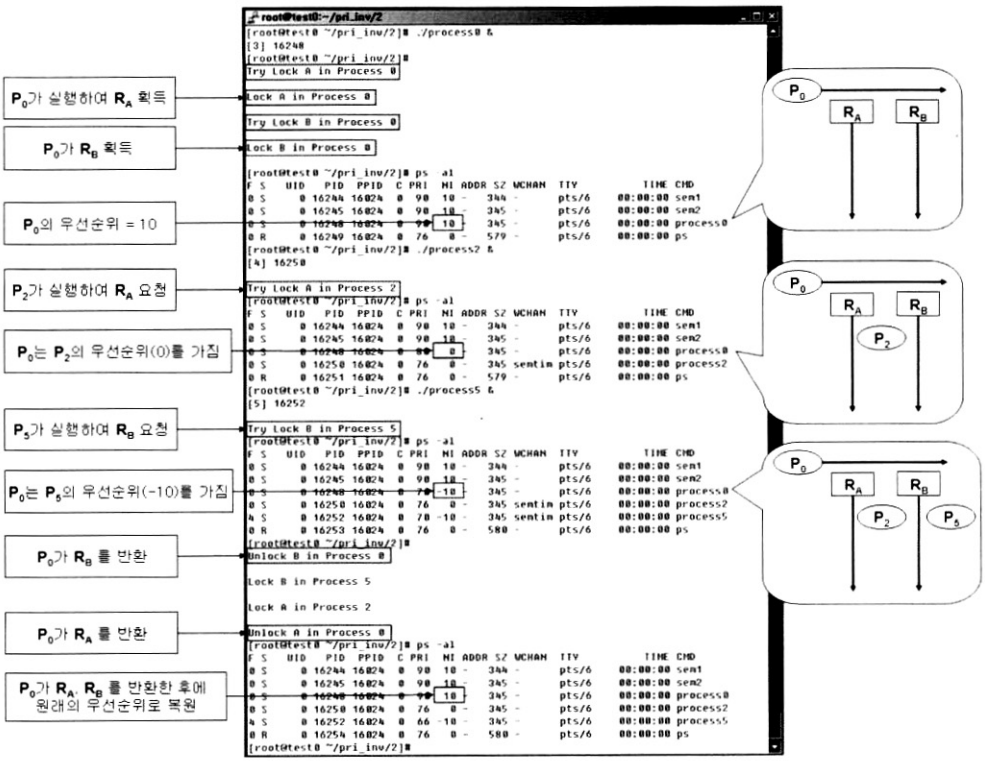
5. 실험 및 검증

본 절에서는 다양한 경우의 우선순위 역전현상에 대해서 본 논문에서 제안하고 구현한 프로토콜을 통해 해결한 내용을 보인다. 공통적으로 테스트 프로그램들은 리눅스에서 제공하는 *setpriority()* 시스템 콜로 우선순위를 조정하고, *semget()* 함수와 *semop()* 함수를 이용하여 세마포어 연산을 수행하게 된다. *setpriority()* 시스템 콜은 내부적으로 *set_user_nice()* 함수를 이용해서 우선순위를 조정한다. 모든 실험에서 각 프로세스의 우선순위는 P_0은 10, P_2는 0, 그리고 P_5는 -10이며, P_0의 우선순위가 가장 낮고 P_5의 우선순위가 가장 높다. 또한 (그림 9), (그림 10), (그림 11)에서 프로세스가 자원을 획득하는 메시지보다 자원을 반환하는 메시지가 늦게 출력되는 경우는 리눅스 운영체제 내에서 프로세스 스케줄링에 의해 출력 순서가 조정되었기 때문이다.

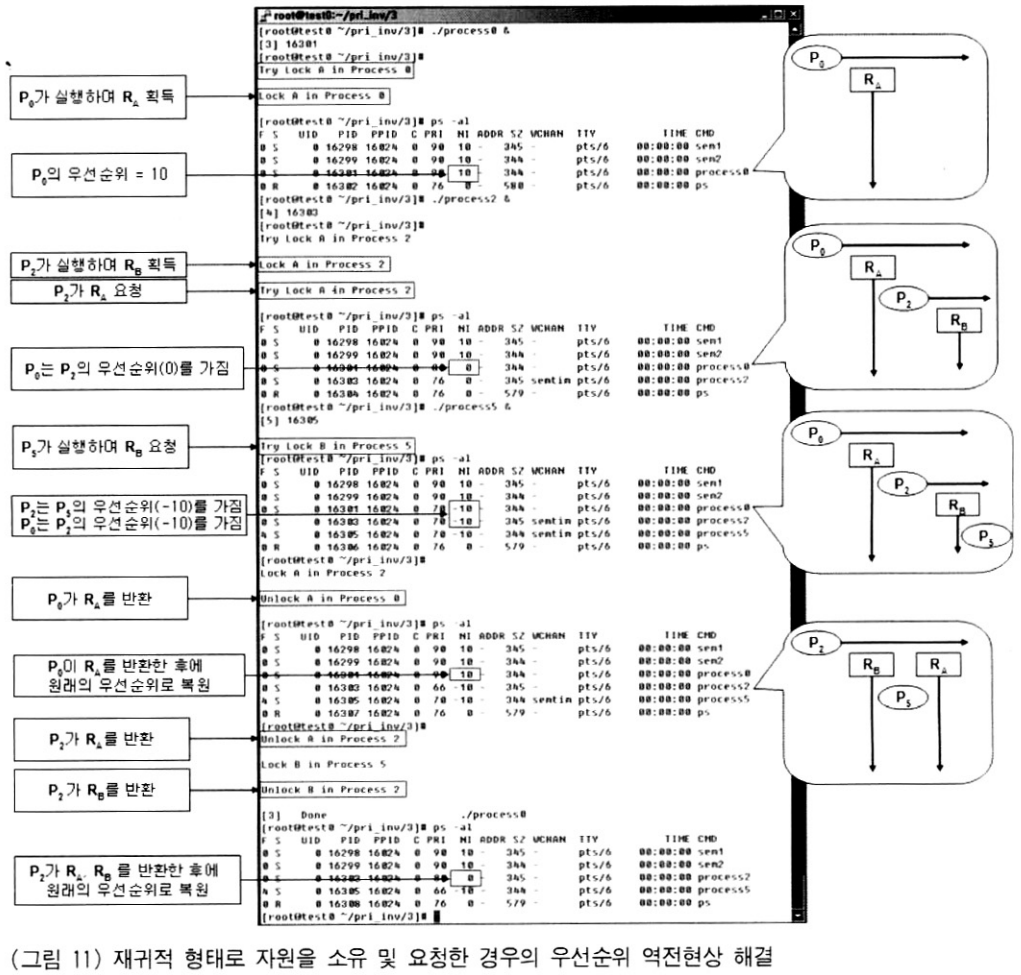
5.1 기본적인 우선순위 역전현상

(그림 9)는 기본적인 우선순위 역전현상((그림 1)의 경우)을 해결하는 것을 보여준다.

(그림 9)에서 우선순위가 10으로 가장 낮은 P_0이 자원의 임계영역에 진입한 상태에서 각각 우선순위 값이 -10인 높은 우선순위를 가진 프로세스인 P_5가 자원 획득을 요청했을



(그림 10) 자원을 동시에 여러 개 소유했을 경우의 우선순위 역전현상 해결



(그림 11) 재귀적 형태로 자원을 소유 및 요청한 경우의 우선순위 역전현상 해결

때 우선순위가 변하는 것을 보여주고 있다. 그리고 P_0이 자원 반환 후에 원래의 우선순위로 돌아온 것을 확인할 수 있다.

5.2 복잡한 우선순위 역전현상

(그림 10)과 (그림 11)에서 자원을 동시에 여러 개 소유했을 경우의 우선순위 역전현상((그림 2)의 경우)과 재귀적으로 자원을 소유 및 요청한 경우의 우선순위 역전현상((그림 3)의 경우)이 해결되는 것을 보인다.

(그림 10)에서 자원 2개를 획득한 P_0이 자원을 요청한 P_1와 P_2에 의해서 우선순위가 변한 후, 원래의 우선순위로 돌아온 것을 확인할 수 있다. 그리고 (그림 11)에서 R_1를 획득한 P_0이 R_B를 획득하고 R_1을 요청한 P_2에 의해 우선순위가 높아지고, 그 후에 R_B를 요청한 P_3에 의해 P_2의 우선순위와 P_0의 우선순위가 높아지게 된다. 또한 각각 자원을 반환한 후에는 원래의 우선순위로 돌아가게 됨을 확인할 수 있다.

5.3 기존 프로토콜과 제안 프로토콜 간 비교

기존에 제안된 기법들인 BPI 프로토콜 및 PCE 프로토콜과 제안 프로토콜 간 비교한 내용을 표 1에서 보인다. 비교 기준으로는 우선순위 역전현상을 해결할 수 있는 지에 대한 여부와 권한 상속에 따른 추가비용 발생여부를 시간복잡도 및 공간복잡도 측면에서 계산하였다. 표 1의 시간 복잡도와 공간 복잡도에서 n은 프로세스의 개수이고 m은 자원의 개수를 의미한다.

본 논문의 3장에서 설명한 바와 같이, BPI 프로토콜은 복잡한 우선순위 역전현상((그림 2)와 (그림 3)의 경우)을 해결하지 못하고, PCE 프로토콜은 재귀적 형태로 자원을 소유 및 요청하는 복잡한 우선순위 역전현상((그림 3)의 경우)을 해결하지 못한다. 그렇지만 본 논문에서 제안하는 RPI 프로토콜은 기본적인 우선순위 역전현상((그림 1)의 경우)과 복잡한 우선순위 역전현상((그림 2)와 (그림 3)의 경우)을 모두 해결할 수 있다.

RPI 프로토콜은 권한 상속을 위해 자원을 요청하거나 반환할 때마다 재귀적 자료구조 내에서 정렬이 발생하므로, 이로 인한 추가비용이 발생한다. BPI 프로토콜의 경우, 권한 상속을 위해 자원할당을 대기하고 있는 프로세스 중에서 가장 높은 우선순위를 가진 프로세스를 찾기 때문에 시간 복잡도가 $O(n)$이고, 특별한 자료구조나 메모리 공간을 소요하지 않으므로 공간 복잡도가 $O(1)$이다. 또한 PCE 프로토콜의 경우, 시스템 초기화 시에 모든 자원에 대해서 해당 자원을 사용하는 프로세스 중 가장 높은 우선순위를 가진 프로세스를 찾기 때문에 시간 복잡도가 $O(n\times m)$이지만, 이러한 작업은 시스템 초기화 시에만 일어난다. 그리고 각 자원이 상속할 우선순위를 저장하기 위해 시스템의 자원의 수만큼 메모리 공간이 요구되기 때문에 공간 복잡도는 $O(m)$가 된다.

〈표 1〉 기존 프로토콜과 제안 프로토콜 간 비교

비교기준 \ 해결방법		기존 프로토콜		제안 프로토콜 (RPI 프로토콜)
		BPI 프로토콜	PCE 프로토콜	
기본적인 우선순위 역전현상		O	O	O
복잡한 우선순위 역전현상	자원을 동시에 여러 개 소유했을 경우	X	O	O
	재귀적 형태로 자원을 소유 및 요청하는 경우	X	X	O
권한 상속에 따른 추가비용	시간 복잡도	$O(n)$	$O(n\times m)$	$O(n+m)$
	공간 복잡도	$O(1)$	$O(m)$	$O(n+m)$

본 논문에서 제안하는 RPI 프로토콜의 경우, 자원의 요청과 반환 시 재귀적 자료구조 내에서 프로세스와 자원의 재정렬이 발생하므로, 권한 상속을 위한 시간 복잡도는 $O(n+m)$이다. 그리고 모든 프로세스가 재귀적 자료구조를 유지하지만, 재귀적 자료구조 안에서 자원 혹은 프로세스가 중복되서 나타나는 경우는 없으므로 공간 복잡도는 $O(n+m)$이 된다. 제안된 RPI 프로토콜이 추가적인 자료구조를 이용하기 때문에 공간 복잡도 측면에서 BPI 프로토콜이나 PCE 프로토콜보다 추가 비용이 발생하지만, 시간 복잡도 측면에서는 PCE 프로토콜보다 효과적이다. 또한 기존 프로토콜이 해결하지 못하는 복잡한 우선순위 역전현상을 해결할 수 있기 때문에 추가비용이 상대적으로 낮다고 볼 수 있다.

6. 결 론

본 논문에서는 실시간 운영체제에서 해결해야 하는 조건 중의 하나인 우선순위 역전현상을 해결하기 위한 기법으로 재귀적 자료구조를 이용한 RPI 프로토콜을 설계하고, 실제 리눅스 커널에 구현 및 검증한 내용을 보였다. RPI 프로토콜의 장점으로는 기존의 제시된 기법으로는 해결이 어려운 복잡한 우선순위 역전현상을 해결할 수 있고, 비교적 간단하게 구현할 수 있으며, 실행 시 추가비용이 상대적으로 낮다.

향후 연구 계획으로는 보다 다양한 우선순위 역전현상에 대한 실험을 실시하고, 현재 구현된 것을 실시간 운영체제에 맞도록 최적화 작업을 실시하며, 세마포어 이외의 다른 자원에 대해 실험함으로써 RPI 프로토콜이 효율적임을 구체적으로 보이도록 한다.

참 고 문 헌

[1] TimeSys Inc., Priority Inversion: Why You Care and What to Do About It. A White Paper, 2004.

[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, Sep., 1990.

[3] D. Polock and D. Zöbel, "Conformance Testing of Priority Inheritance Protocols," Proc. of 7th International Conference

on Real-Time Computing Systems and Applications (RTCSA'00), pp.404-408, Dec., 2000.

[4] B. Akgul, V. Mooney, H. Thane, and P. Kuacharoen, "Hardware Support for Priority Inheritance," Proc. of the IEEE Real-Time Systems Symposium, pp.246-254, Dec., 2003.

[5] Dieter Zöbel, David Polock, Andreas van Arkel, "Testing for the Conformance of Real-time Protocols Implemented by Operating Systems," Electr. Notes Theor. Comput. Sci. Vol. 133, pp.315-332, 2005.

[6] J. B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks," Special Report CMU/SEI-88-SR-4, Mar., 1998.

[7] B. Dutertre, "Formal Analysis of the Priority Ceiling Protocol," Proc. of IEEE Real-Time Systems Symp., pp. 151-160, Nov., 2000.

[8] V. Yodaiken, Against Priority Inheritance, FSMLABS Technical Paper, Jul., 2003.

[9] V. Yodaiken, "The Dangers of Priority Inheritance," FSMLABS Technical Paper, 2001.

[10] Embedded Linux/Microcontroller Project, http://www.uclinux.org.

강 성 구

e-mail lived@ece.skku.ac.kr
2005년 성균관대학교 정보통신공학부 (학사)
2005년~현재 성균관대학교 대학원 컴퓨터공학과 석사과정
관심분야 : 유닉스, 리눅스, 실시간 시스템

경 계 현

e-mail gyehyeon@ece.skku.ac.kr
2006년 호서대학교 컴퓨터공학부 컴퓨터공학전공 (학사)
2006년~현재 성균관대학교 대학원 컴퓨터공학과 석사과정
관심분야 : 리눅스 커널, 시스템 보안, IPv6 네트워크 보안 등

고 광 선

e-mail : rilla91@ece.skku.ac.kr
1998년 성균관대학교 정보공학과 졸업
2004년 성균관대학교 대학원 전기전자및 컴퓨터공학부 (공학석사)
2004년~현재 성균관대학교 대학원 컴퓨터공학과 박사과정
관심분야 : 정보보호, 리눅스, 네트워크 등

엄 영 익

e-mail yieom@ece.skku.ac.kr
1983년 서울대학교 계산통계학과(학사)
1985년 서울대학교 전산과학과(이학석사)
1991년 서울대학교 전산과학과(이학박사)
2000년~2001년 Dept. of Info. and Comm. Science at UCI 방문교수
2005년 한국정보처리학회 학회지 편집위원장
1993년~현재 성균관대학교 정보통신공학부 교수
관심분야 : 분산 컴퓨팅, 이동 컴퓨팅, 이동 에이전트, 시스템 보안, 운영체제, 내장형 시스템 등