시간 결정성을 보장하는 실시간 태스크 스케줄링

Deterministic Real-Time Task Scheduling

조문행, 이숭열, 이원용, 정근재, 김용희, 이철훈
충남대학교 컴퓨터공학과

Moon-Haeng Cho(root4567@cnu.ac.kr), Soong-Yeol Lee(sy-lee@cnu.ac.kr),
Won-Yong Lee(nisskr@cnu.ac.kr), Geun-Jae Jeong(gjjeong@cnu.ac.kr),
Yong-Hee Kim(yonghee@cnu.ac.kr), Cheol-Hoon Lee(clee@cnu.ac.kr)

요약

오늘날의 내장형 시스템은 군사 무기체계, 로봇, 인공위성 등과 같이 전통적인 내장형 시스템에서 휴대폰, 디지털 캠코더, PMP, MP3플레이어와 같은 보다 복잡한 응용프로그램 구동을 필요로 하는 휴대용 시스템으로 그 영역을 넓혀가고 있다. 이런 내장형 실시간 시스템은 내장형 시스템의 한정된 자원을 효율적으로 관리하고 시간적 논리적 정확성을 보장하기 위해 실시간 운영체제를 사용한다. 실시간 운영체제의 서비스를 통해 응용프로그래머는 응용프로그램을 구성하는 각 태스크가 시간 결정성에 위배되지 않도록 응용프로그램을 구현할 수 있다. 더욱이, 실시간 운영체제는 시간 결정성 보장을 위해 스케줄링과 문맥교환에 사용되는 시간을 예측할 수 있어야 한다. 본 논문에서는 추가적인 메모리 오버헤드 없이 22r 레벨의 우선순위를 갖는 시스템에서 고정 상수 시간 내에 가장 높은 우선순위를 갖는 태스크를 결정할 수 있는 알고리즘에 대해 기술한다.

■ 중심어 : | 실시간 운영체제 | 시간 결정성 | 태스크 스케줄링 |

Abstract

In recent years, embedded systems have been expanding their application domains from traditional applications (such as defense, robots, and artificial satellites) to portable devices which execute more complicated applications such as cellular phones, digital camcoders, PMPs, and MP3 players. So as to manage restricted hardware resources efficiently and to guarantee both temporal and logical correctness, every embedded system use a real-time operating system (RTOS). Only when the RTOS makes kernel services deterministic in time by specifying how long each service call will take to execute, application programers can write predictable applications. Moreover, so as for an RTOS to be deterministic, its scheduling and context switch overhead should also be predictable. In this paper, we present the complete generalized algorithm to determine the highest priority in the ready list with 22r levels of priorities in a constant time without additional memory overhead.

■ keyword : | Real-Time Operating Systems | Time Determinism | Task Scheduling |

I. 서 론

1. 실시간 시스템

실시간 시스템은 시스템 동작의 정확성이 논리적 정확성뿐만 아니라 시간적 정확성에서 좌우되는 시스템으로 정의한다. 또한, 시스템의 수행 결과가 기능적으로 정확해야 할 뿐만 아니라, 결과가 도출되는 시간 역시 주어진

* 본 연구는 정보통신부의 선도기반기술개발사업의 지원으로 수행되었습니다.
접수번호 : #061101-002
접수일자 : 2006년 11월 01일
심사완료일 : 2006년 12월 21일

교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

제약 조건을 만족해야 한다. 실시간 시스템은 주어진 시간의 엄격성에 따라 크게 3가지로 분류할 수 있다[1][2].

- **경성(hard) 실시간 시스템** : 시스템이 주어진 종료시한을 만족시키지 못한 경우에 막대한 재산적 손실이나 인명의 피해를 주는 시스템.
- **연성(soft) 실시간 시스템** : 온라인 시스템과 같이 시간제약 조건을 만족시키지 못하더라도 경성의 경우처럼 치명적이지 않고 종료시한을 넘겨 수행을 마쳐도 계산의 결과가 의미가 있는 시스템.
- **준경성(firm) 실시간 시스템** : 경성과 연성의 중간 형태로 종료시한을 넘겨 수행을 마치는 것은 무의미한 경우이지만 시간초과에 대한 손실이 치명적이지 않은 시스템.

2. 실시간 운영체제의 시간 결정성

실시간 시스템은 어떠한 상황에서도 예측 가능해야 한다. 즉 시간 결정성을 보장해야 한다. 이런 예측성은 실시간 운영체제가 제공하는 시스템 수준에서의 커널 서비스를 통해 보장할 수 있다. 또한 실시간 운영체제는 모든 실시간 태스크에 대해 그 종료 시한을 만족할 수 있도록 해야 하며, 실행 준비 상태의 높은 우선순위의 태스크가 존재하는 데도 낮은 우선순위의 태스크가 CPU를 점유하는 우선순위 역전현상이 발생하지 않도록 보장해야 한다. 또한, 주어진 종료시한을 만족하기 위해 스케줄러와 태스크관리는 시간 결정성을 보장해야 한다.

태스크 스케줄링 오버헤드(Δt)는 스케줄러 코드에 의해 실행되는 시간이며, 스케줄러 코드는 실행이 중단되거나 재개될 때 실행 준비 상태의 태스크들 중 가장 우선순위가 높은 태스크를 선택하고 관리할 수 있도록 하는 실행 코드이다. 태스크의 실행이 중단되었을 경우, 실시간 운영체제는 실행 중단될 태스크를 관리하기 위한 일부 데이터 정보를 변경하고 새로 실행할 태스크를 선택해야 하는데, 태스크를 중단하기 위해 소요될 시간을 블록킹 오버헤드(Δtb)로, 새로 실행할 태스크를 선택하기 위해 소요되는 시간을 선택 오버헤드(Δts)로 구분한다. 이와 유사하게, 실행 중단된 태스크를 실행 재개하기 위해서 소모되는 시간을 언블록킹 오버헤드(Δtu)로 표현하고 언블록킹 시에도 현재 수행중인 태스크가 실행 준비 상태의 태스크보다 우선순위가 낮은 경우 문맥교환이 발생하게 되는데, 이때에도 실행 준비 상태의 태스크들 중 가장 우선순위가 높은 태스크를 찾기 위한 선택 오버헤드가 존재한다.

오늘날의 상용 실시간 운영체제는 일반적으로 우선순위에 따라 정렬된 우선순위 큐에 의해 태스크를 관리하며, 정렬된 우선순위 큐의 가장 앞에 있는 태스크가 가장 높은 우선순위를 갖는다. 포인터 변수 HighestP가 이를 가리키게 되어, 선택 오버헤드(Δts)는 실행 준비 상태의 태스크 수에 상관없이 항상 O(1)이 된다. 또한, 태스크 실행이 중단될 경우의 오버헤드는 태스크 제어 블록(TCB) 필드의 값을 갱신하는 시간 O(1)과 가장 높은 우선순위를 갖는 태스크를 가리키는 포인터 변수 HighestP를 정렬된 큐의 다음 태스크를 가리키도록 변경하는 시간 O(1)의 합이 된다. 하지만, 태스크 실행이 재개될 경우의 오버헤드는 실행 재개될 태스크를 그 우선순위에 따라 정렬 큐에 삽입할 위치를 찾기 위한 시간은 그 태스크의 수(n)에 따라 달라진다. 즉, 언블록킹 오버헤드(Δtu)는 O(n) 시간이 된다. 이에 따라 최악의 경우 스케줄링 오버헤드는 n까지 증가할 수 있다. 이런 스케줄링 오버헤드는 실시간 운영체제의 시간 결정성에 치명적 영향을 미치며, 태스크의 종료 시한을 보장하지 못할 수 있다[4][5].

µC/OS는 독창적인 데이터 구조체를 사용하여 시간 결정적인 스케줄러를 제안했다. µC/OS의 스케줄링 오버헤드(Δt)는 응용 프로그램에서 생성하는 태스크의 수에 상관없이 상수의 시간을 갖는다. 하지만, µC/OS는8-bit 마이크로 컨트롤러를 위해 만들어졌기 때문에, 유일한 우선순위로 64개의 태스크에 우선순위를 할당할 수 있다. 즉, 동일 우선순위는 지원하지 않기 때문에 64개의 태스크만 생성이 가능하다. 이런 µC/OS의 생성 태스크 수의 제한은 근래의 다수의 태스크로 이루어진 복잡한 응용프로그램을 구동하기 위해서는 적합하지 않다. 우리는 이미 추가적인 메모리 오버헤드 없이 µC/OS의 제약사항을 제거한 일반화된 알고리즘을 제안하였다[6]. 본 논문에서는 임의의 수 r에 대해 22r 레벨의 우선순위 상에서 실행 준비 상태의 태스크들 중 가장 높은 우선순위를 결정하는 완전히 일반화된 시간 결정성을 갖는 스케

줄링 알고리즘을 제안한다.

본 논문의 구성은, 본론에서 우선순위 선택을 위한 일반적인 탐색 알고리즘의 스케줄링 오버헤드와 μC/OS에서 제안한 시간 결정성을 갖는 스케줄링 알고리즘에 대해 간략히 소개하고[3], 생성 가능한 태스크 수에 제한이 있는 μC/OS의 제약사항을 해결한 본 논문의 22r 레벨의 우선순위 상에서 실행 준비 상태의 태스크들 중 가장 높은 우선순위를 결정하는 완전히 일반화된 시간 결정적인 스케줄링 알고리즘에 대해 소개한다. 마지막으로, 결론에서는 본 논문의 완전히 일반화된 시간 결정성을 갖는 스케줄링 알고리즘에 대해 소개한다.

II. 본 론

1. 우선순위 선택을 위한 일반적인 탐색 알고리즘

실행 큐에 있는 태스크들 중에서 가장 우선순위가 높은 것을 검색하는 방법은 선형탐색(linear search), 이진탐색(binary search), 선택정렬(selection sort) 등 이 있을 수 있으며, 그 중 가장 간단한 선형 검색의 코드는 [표 1]과 같다.

표 1. 선형 검색 코드

```
task_t select_task() // Linear Search algorithm
{
    int p;
    for(p = MAX_PRIORITY; p > 0; p—) {
        if(!ready_q_empty(&task_ready_q[p])) {
            return ready_q_first(&task_ready_q[p]);
        }
    }
    return 0;
}
```

표 2. 검색 알고리즘에 따른 시간 공간 복잡도

Algorithms	Time Complexity			Space Complexity
	Min.	Max.	Average	
Linear Search	O(1)	O(n)	O(n)	O(n)
Binary Search	O(1)	O(logn)	O(logn)	O(n)
Selection Sort	O(n2)	O(n2)	O(n2)	O(n)

탐색 알고리즘을 통해 가장 높은 우선순위를 찾는 방법은 최악 수행시간이 매우 길 뿐만 아니라 태스크의 수에 따라 수행시간의 차이가 많이 나게 된다. 표2는 일반적인 가장 높은 우선순위의 태스크를 찾기 위한 탐색 알고리즘의 시간과 공간 복잡도를 나타낸 것으로 최상의 우선순위를 가진 태스크를 찾기 위하여 발생하는 비교횟수가 대부분 O(n)의 시간 복잡도를 가진다. 이것은 실시간 시스템에 있어서 치명적인 제약이 될 수 있다. 따라서 일반적인 탐색이나 정렬에 의한 방법을 사용하지 않고 일정한 연산을 통해 우선순위를 찾아내는 방법을 사용하게 된다.

2. μ C/OS의 시간 결정성을 갖는 알고리즘

다중 실시간 시스템에서 여러 태스크 중 CPU점유는 하나의 태스크 밖에 할 수 없으므로 수행 준비 테이블에 등록을 한 후 그 중에서 가장 우선순위가 높은 태스크만이 CPU를 점유하여 수행하게 된다.

μC/OS의 시간 결정성을 갖는 알고리즘에 대해 살펴보자.

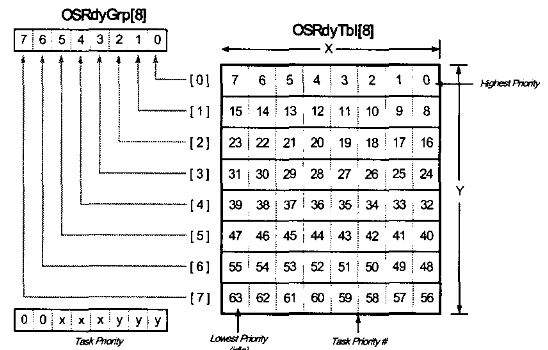


그림 1. 준비 그룹과 준비 테이블

시스템이 64개의 우선순위를 가질 경우 준비테이블은 [그림 1]과 같이 나타낼 수 있다. 각각의 태스크는 0과 63 사이의 고유한 우선순위로 지정되며, 가장 낮은 우선순위인 63은 초기화될 때 항상 유휴상태(Idle State)로 지정된다. 수행 준비상태의 태스크는 준비테이블(OSRdyTbl[])과 준비그룹(OSRdyGrp)으로 구성된 준비

리스트에 위치한다. 준비그룹은 8개의 태스크를 하나의 그룹으로 지정하며, 각 비트는 해당 그룹에 수행 준비상태의 태스크가 있는지를 나타낸다. 태스크가 수행 준비되면 그와 관련된 준비테이블의 8개의 변수 중 우선순위에 대응하는 수행대기 비트를 설정한다. 다음 수행할 태스크는 스케줄러가 OSRdyTbl[8]에 설정된 가장 낮은 위치의 비트를 선택하여 결정한다. 실행 중지된 태스크가 해제될 때 또는 수행 중에 보다 더 높은 우선순위의 태스크에 의해 CPU 점유권을 빼앗긴 경우에, 다음과 같은 몇 개의 코드 연산에 의해서 준비리스트에 수행준비 상태임을 표시한다. 즉 수행준비가 된 우선순위p인 태스크를 준비리스트에 삽입하는 경우에는, 우선순위 p에 대응하는 준비테이블 및 준비그룹의 비트를 '1'로 설정하고, 우선순위 p에 대응하는 준비 큐[p]에 삽입한다.

표 3. 맵 테이블 생성

index	bit mask	
	binary	hexa
0	00000001	0x01
1	00000010	0x02
2	00000100	0x04
3	00001000	0x08
4	00010000	0x10
5	00100000	0x20
6	01000000	0x40
7	10000000	0x80

[표 3]의 OSMapTbl은 0과 7사이의 색인을 비트 마스크(Bit Mask)로 변환하는 것으로, 이 테이블은 미리 여러 개의 태스크들이 존재할 때 각각에서 가장 우선순위가 높은 우선순위의 위치를 정해놓은 일종의 우선순위 변환 테이블이다. 즉 k번째 이진수는 k번째 비트만이 '1'이고 나머지 비트는 '0'인 8개의 비트를 가진 8개의 이진수로 구성된 맵 테이블을 이용하여 다음 식의 연산에 따라서 태스크의 우선순위(p)에 대응하는 준비그룹과 준비테이블을 '1'로 설정하여 삽입한다.

```
OSRdyGrp |= OSMapTbl[p >> 3];
OSRdyTbl[p>>3] |= OSMapTbl[p & 0x07];
```

p는 태스크의 우선순위이다. 이러한 연산에 의한 방법은 중단된 태스크의 해제 동작이 고정된 수의 명령어 처리에 의해 수행됨을 의미한다.

[그림 1]에서 태스크 우선순위의 하위 3비트(Task Priority의 xxx 표시)는 준비테이블의 비트 위치를 결정하며, 다음 상위 3비트(Task Priority의 yyy 표시)는 준비그룹의 색인(Index)을 나타낸다.

수행이 중단되거나 더 높은 우선권의 태스크가 준비상태가 되는 등의 사건이 발생하여 우선순위가 p인 태스크를 준비리스트에서 제거하는 경우에는 태스크 p를 준비 큐에서 제거하고, 다음과 같은 코드의 연산에 의해 준비리스트에서 제거된다.

```
if((OSRdyTbl[p>>3] &= ~OSMapTbl[p&0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[p >> 3];
```

위의 코드를 살펴보면, OSRdyTbl[p>>3]에서 중지된 태스크의 수행준비 비트를 지우고 해당 그룹의 모든 태스크들이 수행대기상태가 아니면(즉, OSRdyTbl[p>>3]이 0일 때), 준비그룹의 해당되는 그룹 비트를 지운다.

준비리스트에 있는 태스크들 중에 가장 우선순위가 높은 태스크를 찾기 위하여 준비리스트에 있는 모든 태스크를 비교하는 방법을 사용하지 않고 연산에 의해서 최고의 우선순위 태스크를 찾는다. 이러한 연산에 사용되는 우선순위 변환용 언맵 테이블은 미리 여러 개의 태스크들이 존재할 때 각각에서 가장 우선순위가 높은 우선순위의 위치를 정해 놓은 일종의 우선순위 변환 테이블이다. 먼저 사용 가능한 우선순위 수를 [그림 2]와같이 정방형의 테이블을 갖는 형태로 표시하고 각각의 열을 하나의 그룹(ReadyGrp)으로 정의한다. 그리고 각각의 그룹을 주어진 우선순위를 갖는 태스크가 존재하는 우선순위 비트를 '1'로 설정한 후 그들 중에서 가장 우측에 있는 비트가 '1'인 곳을 찾아내는 방법을 사용한다.

```
/* priority resolution table */
uint const OSUnMapTbl[] = {
        0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
        4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
};
```

그림 2. 256 크기의 언맵 테이블

[그림 2]의 OSUnMapTbl[256]은 우선순위 분해 테이블(Priority Resolution Table)로써 8비트는 한 그룹 내의 어떤 태스크가 수행대기 상태에 있는지를 표시하며, 최하위 비트가 가장 높은 우선순위를 나타낸다. 이 테이블은 0에서 7까지의 수를 사용하여 1회의 연산으로 비트 내에 가장 우선순위가 높은 수행대기 상태에 설정된 비트의 위치를 전달한다. 다음은 수행대기상태에 있는 태스크들 중 가장 우선순위가 높은 태스크를 결정하는 연산코드이다.

```
y  =  OSUnMapTbl[OSRdyGrp];
x  =  OSUnMapTbl[OSRdyTbl[y]];
p  =  (y << 3) + x;
```

여기서 p는 수행준비 큐(Ready Queue)에서 수행하기 위해 대기 중인 태스크들 중에 가장 높은 우선순위를 갖는 태스크의 우선순위를 가리키며, 해당되는 태스크의 TCB 포인터(즉, higestP)를 얻는 것은 각각의 우선순위가 고유한 태스크와 연결되므로 단순한 연산만으로 가능하다. 그러므로 수행준비 중인 태스크들 중 가장 높은 우선순위 태스크를 결정하는데 태스크의 수에 따라 변하는 탐색방법이 아닌 고정된 몇 개의 명령어 처리를 수행하는 것만으로 가능하다. 즉 이러한 연산을 이용한 태스크 스케줄링 방법은 응용 프로그램에서 생성된 태스크의 수에 관계없이 일정하다. 따라서 스케줄러가 시간결정성을 갖는다. 그러나 언맵 테이블을 이용한 방법은 최우선순위를 준비상태에 있는 태스크들 수에 상관없이 일정 시간 내에 찾을 수 있는 장점이 있지만, 사용가능한 우선순위의 수가 늘어나면 언맵 테이블의 수가 더 커지므로 메모리 사용에 불이익을 가져오게 된다. 그 예로 우선순위 수가 64개일 경우의 언맵 테이블 수는 256(28)개 이지만 사용 가능한 우선순위의 수가 256개로 늘어난다면 언맵 테이블의 수는 65,536(216)개로 늘어나게 되므로 메모리 사용에 제한이 있는 내장형 시스템에서는 문제가 발생할 수 있다.

수행준비 상태의 태스크들 중에 가장 높은 우선순위를 계산하기 위한 언맵 테이블(OSUnMapTbl[])은 단순하게 확장하면 2^{2^x}개의 이진수 배열이 된다. 예를 들어 앞 장에서 처럼 x=3인 경우에는 언맵 테이블(OSUnMapTbl[])이 28(256)개이지만, x=4인 경우는 216(65,536) 개로 x=3인 경우에 비해 256배 증가하고, x=5인 경우는 x=3인 경우에 비해 무려 224(16,777,216) 배의 메모리를 요구한다. 따라서 우선순위의 수를 증가시키면서 메모리 양이 적은 소형 내장형 시스템에는 실제로 사용이 불가능한 방식이 되며 따라서 실제로 이용하기 위해서는 언맵 테이블의 크기를 줄여야 한다. 이러한 제한은 최근의 다양한 응용프로그램을 지원하기에는 여러 가지로 제약이 있을 수 있다. 따라서 본 논문에서는 기존의 비트맵을 이용한 연산이 갖는 시간결정성을 갖는 태스크 스케줄링의 장점을 유지하면서 태스크가 가질 수 있는 우선순위의 수가 증가할 경우 발생하는 기하급수적인 메모리 증가의 문제점을 보완한 일반화된 시간결정성을 갖는 스케줄링 방법을 제안한다.

3. 일반화된 시간 결정성을 갖는 스케줄링 알고리즘

앞 장에서 시간 결정성은 보장하지만 최대 우선순위가 64로 제한된 μC/OS의 스케줄링 기법과 이런 제한 사항을 제거하기 위한 스케줄링 기법에 대해서 살펴 보았다. 하지만, 태스크가 가질 수 있는 우선순위의 수를 증가시켰을 때, 단순한 계산만으로 최고의 우선순위를 찾는데 사용되었던 언맵 테이블의 크기가 기하급수적으로 증가하여 메모리 크기가 많이 요구되는 문제점이 있었다. 이러한 문제를 해결하기 위해서는 실시간 시스템의 기본조건이면서 장점인 시간 결정성을 갖는 스케줄링 알고리즘

과 기하급수적인 메모리의 증가를 요구하지 않는 스케줄링 알고리즘이 필요하다.

이에 앞서 스케줄러에 대해 다시 일반화해 본다. 스케줄러는 태스크 리스트를 관리하고 수행중인 태스크가 중단되거나 태스크 중단이 해제될 때 마다 수행준비상태에 있는 가장 높은 우선순위 태스크를 선택하고 준비리스트를 갱신한다. 스케줄러의 동작은 준비리스트에 수행 준비된 태스크를 표시(추가)하는 동작, 준비리스트에서 중단되는 태스크를 제거하는 동작과 수행준비 상태인 태스크들 중에 가장 우선순위가 높은 태스크를 선택하는 동작으로 구분할 수 있다. 준비리스트에서 가장 우선순위가 높은 태스크를 선택하는 동작은 가장 우선순위가 높은 태스크가 있는 그룹을 먼저 찾은 후, 해당 그룹의 준비테이블에서 수행할 태스크를 선택하는 동작으로 구분한다. 우선순위는 준비테이블의 2r개 비트수와 2r그룹수로 분해할 수 있다. [그림 3]에서와 같이 준비테이블(OSRdyTbl[2r])은 2r개의 비트를 가진 2r개의 이진수로 구성된 배열을 갖는 구조로써 각 각의 비트는 우선순위를 나타내고, 준비그룹은 각 준비테이블의 이진수에 대응하는 비트로 구성되며, 준비 리스트는 준비테이블과 준비그룹 이외에 [그림 3]과 같이 준비리스트 각 비트에 대응하는 22r개의 준비 큐(queue)를 포함하여 여러 개의 태스크들이 동일한 우선권을 가질 수 있도록 한다. 태스크의 우선순위(p)는 2r개의 비트로 구성되어 좌측 r개 비트(y)는 준비그룹에서의 위치를 나타내고 우측 r개 비트(x)는 준비테이블의 위치를 나타낸다. 태스크의 우선순위(p)는 그 숫자가 낮을수록 높은 것이어서 준비그룹에서 우측의 비트가 우선순위가 높으며 준비테이블에서도 우측의 비트가 우선순위가 높다.

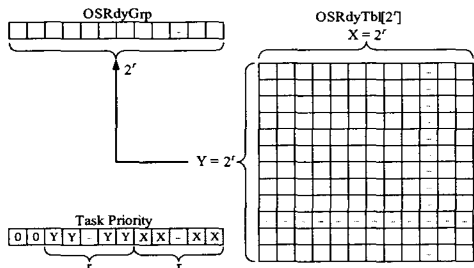


그림 3. 일반화된 준비 테이블

[그림 3]의 준비그룹과 준비테이블의 관계를 살펴보면 우선순위의 수(number_P)는 준비그룹의 비트수(x)와 그룹에 대응하는 준비테이블의 비트 수(y)의 곱으로 나타내어진다.

표 4. 우선순위에 따른 언맵(unmap) 테이블 크기

priority	ready group(y)	ready table(x)	unmap table
64	8	8	256(=28)
128	16	8	65536(=216)
	8	16	
256	16	16	65536(=216)
512	16	32	4,294,967,296(=232)
	32	16	4,294,967,296(=232)
1,024	32	32	

우선순위를 64, 128, 192, 256, …, 512, …, 1024, … 등으로 확장하는 경우에 준비그룹의 비트수와 준비테이블의 비트수를 조합을 보면, 64개의 우선순위 사용에 있어서 8비트 값을 가진 준비그룹과, 크기가 8이고, 8비트를 표현할 수 있는 배열이 준비테이블로 사용되었다. 그러나 태스크가 가질 수 있는 우선순위의 확장을 위하여 128개의 우선순위 사용을 가정 한다면 priority = readygroup(y) X readytable(x)의 조합에 의하여 16비트의 준비그룹과 8비트의 준비테이블 또는 8비트 준비그룹과 16비트 준비테이블의 조합으로 확장할 수 있다. 64개의 우선순위를 예를 들어 설명한 것과 같이 준비그룹과 준비테이블의 비트 중에서 가장 우선권이 높은 비트를 찾는 문제, 즉 가장 왼쪽 비트를 찾는 문제는 256개의 기존의 언맵 테이블을 이용하였다. 그러나 128개의 우선순위에 대해서는 128=16 X 8 로 인하여, 준비그룹을 위한 216개의 언맵 테이블과 준비테이블을 위한 28개의 언맵 테이블이 각각 필요하다. 이는 메모리 관리 관점에서 효율적이지 못하다. 또한 192개의 우선순위를 가진 시스템은 12비트의 준비그룹과 16비트의 준비테이블 또는 16비트의 준비그룹과 12비트의 준비테이블 조합으로 확장할

수 있다. 여기서, 192개의 우선순위를 구현하는 관점에서 살펴보면, 12비트의 준비그룹을 위한 언맵 테이블과 16 비트의 준비테이블을 위한 언맵 테이블 사용은 언맵 테이블의 중복 사용을 고려할 때 효과적이지 못하다. 이런 경우 256개의 우선권을 가진 것과 비교할 때 공간 복잡도(Space Complexity)면에서 오히려 비효율적이다. 아래 [그림 4]는 우선순위의 수(n)에 따른 언맵 테이블의 공간 복잡도를 나태내고 있다.

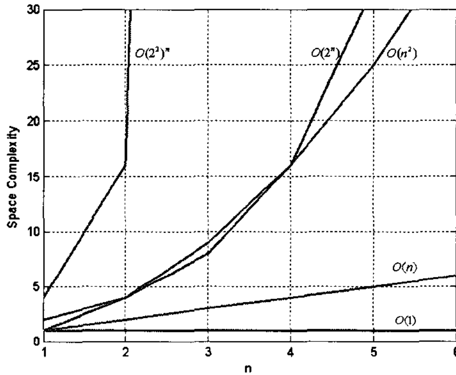


그림 4. 일반화된 준비 테이블

따라서, 구현 측면에서 우선순위의 확장은 준비그룹과 준비테이블의 비트수가 8의 배수와 16의 배수조합으로 유리하다. 그러나 이러한 조합도 216개의 언맵 테이블을 고려할 때 제한적인 메모리사용을 요구하는 내장형 시스템에서는 문제가 될 수 있고 더 나아가서 512개 이상의 우선순위가 요구되는 시스템에서는 232개의 언맵 테이블이 필요하고 [표 4]에서 나타나 있듯이 그 이상의 우선순위에 대해서는 실제로 구현이 불가능한 방법이 된다. 따라서 우선순위의 수를 증가시키기 위해서 이렇게 단순히 기하급수적으로 늘어나는 업 맵 테이블의 사용은 불가능한 방식이 되며 따라서 실제로 이용하기 위해서는 언맵 테이블의 크기를 줄여야 한다.

본 논문에서는 언맵 테이블의 크기를 줄이는 방법으로 64개의 우선순위에 적용하였던 256개의 언맵 테이블을 이용하는 방법으로 임의의 수에 대하여서도 확장이 가능함을 예를 통해 보인다.

256개의 우선순위(22r 우선순위에서 r을 4로 설정)를 갖는 태스크 스케줄링을 살펴보자. 수행준비 상태가 된 우선순위가 p인 태스크를 준비리스트에 삽입하는 경우에는, 태스크의 우선순위에 대응하는 준비그룹(OSRdyGrp, 한 그룹당 16개의 태스크)과 준비테이블(OSRdyTbl[16])에 '1'로 세팅하고 준비 큐[p]에 삽입한다. k번째 이진수는 k번째 비트만이 '1'이고 나머지 비트는 '0'인 2r개의 비트를 가진 2r개의 이진수로 구성되어 맵 테이블(OSMapTbl[16])을 이용하여, 다음 식에 따라 태스크의 우선순위(p)에 대응하는 준비그룹 및 준비테이블 비트를 '1'로 세팅한다.

```
OSRdyGrp  |=  OSMapTbl[p>>4];
OSRdyTbl[p>>4]  |= OSMapTbl[p & 0x0F];
```

수행이 중단되거나 더 높은 우선순위의 태스크가 수행준비 상태가 되는 등의 사건이 발생하여 우선순위가 p인 태스크를 준비 리스트에서 제거하는 경우에는, 태스크를 준비 큐[p]에서 제거하고, 태스크가 속하였던 준비 큐[p]에 다른 태스크가 있는지를 검색 한 후, 다른 태스크가 없으면 준비 큐에 대응하는 준비 테이블의 비트를 '0'으로 세팅한다. 그리고 그 비트가 속하는 이진수의 모든 비트가 '0'인지를 검색한 후, 이진수의 모든 비트가 '0'이면 이진수에 대응하는 준비그룹의 비트를 '0'으로 세팅한다.

```
if((OSRdyTbl[p>>4]  &=  ~OSMapTbl[p&0x0F])== 0)
        OSRdyGrp  &=  ~OSMapTbl[p >> 4];
```

준비 리스트를 이용하여 최고 우선순위(p)를 결정하는 경우에는, 준비그룹에서 그 값이 '1'인 비트들 중 가장 우측에 위치한 비트를 구하고, 비트에 대응하는 준비테이블의 이진수에서 그 값이 '1'인 비트들 중 가장 우측에 위치한 비트를 구함으로써 우선순위가 가장 높은 태스크를 결정한다. 그러나 이를 위해서는 2^{2^r}개의 언맵 테이블이 필요하다. 따라서 본 논문 에서는 언맵 테이블의 크기가 태스크가 가질 수 있는 우선순위의 수인 준비리스트의 크기에 비례하여 $O(2^{2^n})$으로 늘어나는 방법 대신에 약간의 시간적인 부담을 지불하여 언맵 테이블의 크기를

$O(2^n)$로 줄이는 방법을 제안한다. 기존의 방법이 준비리스트의 크기에 따라서 언맵 테이블의 크기를 맞추었던 방식이라면 제안한 방식은 언맵 테이블의 크기에 맞추어 준비리스트의 크기를 분할하는 방식이다. 즉, 준비그룹의 비트들을 2등분하여 우반부가 '0'이면 좌반부를, '0'이 아니면 우반부를 택하는 단계를 루프(loop)로 형성하여, r보다 작은 임의의 양의 정수 k값에 따라 채택된 좌반부 또는 우반부를 다시 2등분하는 단계를 (r-k)번 반복하는 단계를 거쳐, 작은 크기의 언맵 테이블을 이용 할 수 있도록 [그림 5]의 흐름도와 같이 재구성하였다. [표 5]는 [그림 5]의 흐름도에 따라 최상위 우선순위 선택을 위해 준비 그룹과 준비 테이블의 값을 계산하는 알고리즘이다.

표 5. 최상위 우선순위 선택 알고리즘 코드

```
int i, j, a, b, c
i = 0, a = r - 1, b = OSRdyGrp
do {
   if ((b & mask[2^a]) == 0)
       i = i + 2^a, b = b >> 2^a
   else   b = b & mask[2^a]
   a = a - 1;
} while(a > 0)
y = i + OSUnMapTbl[b];

j = 0, a = r - 1, c = OSRdyTbl[y];
do {
   if ((c & mask[2^a]) == 0)
       j = j + 2^a, c = c >> 2^a
   else   c = c & mask[2^a]
   a = a - 1
} while (a > 0)
x = j + OSUnMapTbl[c];
p = (y << r) + x;
```

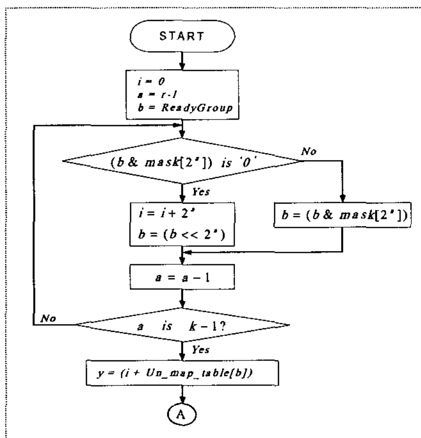


그림 5. 준비 그룹에서 최상위 우선순위 비트 찾기

이진수 '0'에서 2^{2k-1}까지의 값들에서 '1'이 처음 나타나는 비트 위치 값을 각 인덱스(index)의 값으로 하는 256개의 언맵 테이블을 이용하여, 다음 식에 의해 준비그룹에서 그 값이 '1'인 비트들 중 가장 우측에 위치한 비트의 위치(y)를 구한다.

$$y = i + \mathrm{OSUnMapTbl}[b];$$

(여기서, b는 준비그룹을 반복하여 이등분하는 단계를 통하여 최종 채택된 부분이고, i는 준비그룹에서 b의 우측에 있는 비트 수를 나타낸다.)

두 번째 단계로 [표 5]와 같이 선택된 준비그룹에 해당하는 준비테이블에서 최상위 우선순위를 가진 비트를 찾아내는 부분이다. y값에 대응하는 준비테이블[y]의 비트를 이등분하여 우반부가 '0'이면 좌반부를, '0'이 아니면 우반부를 채택하는 단계를 $(r-k)$번 반복하여 주어진 언맵 테이블 크기에 맞을 때까지 그 루프를 반복한다. 언맵 테이블을 이용하여, 다음 식에 의해 준비테이블[y]에서 그 값이 '1'인 비트들 중 가장 우측에 위치한 비트의 위치(x)를 구하고, 최종적으로는 최고의 우선순위(p)를 결정하는 단계로 수행한다.

$$x = j + \mathrm{OSUnMapTbl}[c];$$
$$p = (y << r) + x;$$

(여기서, c는 준비테이블[y]를 반복하여 이등분 하는 단계를 통해 최종 채택된 부분이고, j는 준비테이블[y]에서 c의 우측에 있는 비트 수이고, <<는 좌측으로 이동(shift)연산자이다)

k값은 r 보다 작은 임의의 값이며, k값을 크게 하면 루프를 돌며 반복하는 횟수가 줄어 시간을 단축할 수가 있으나 언맵 테이블의 크기가 증대하게 되고, k값을 작게 하면 언맵 테이블의 크기가 줄어들지만 루프를 도는 횟수가 증가해 시간이 지연된다. 또한 제안된 방법에서는 최고 우선순위 결정단계에서 k값을 적당히 조정함으로써 소요되는 시간과 메모리 공간사용 사이에서의 응용 시스템의 요구에 따라서 절충이 가능하다.

본 논문에서 제안한 알고리즘과 μC/OS의 알고리즘을 비교하기 위해 256, 1024, 4096단계로 우선순위를 확장했

을 때 언맵테이블 크기와 최상위 우선순위를 결정하는데 수행되는 명령어 수는 [표 6]과 같다.

표 6. 우선순위에 따른 언맵(unmap) 테이블 크기와 최상위 우선순위를 결정하기 위한 실행 명령어 수

알고리즘 / 우선순위	언맵테이블 크기		실행 명령어 수	
	μC/OS	제안 알고리즘	μC/OS	제안 알고리즘
64(26)	256(=28)	256	3	11
256(28)	65536(=216)	256	3	15
1,024(210)	4,294,967,296 (=232)	256	3	19
4,096(212)	264	256	3	23

[표 6]에서 처럼, μC/OS의 알고리즘을 사용할 경우 실행 명령어 수는 최상위 우선순위의 레벨에 상관없이 3개의 명령어만 실행하면 되지만, 우선순위가 증가함에 따른 언맵테이블 크기로 인한 메모리 오버헤드가 지수적으로 증가함을 알 수 있으며, 이를 통해 메모리 자원이 한정적인 임베디드 시스템에서 μC/OS의 알고리즘을 사용하여 시간결정성을 보장하는 것이 어렵다는 것을 알 수 있다.. 본 논문에서 제안한 알고리즘은 우선순위의 증가에 따른 실행 명령어 수는 증가하지만 이것 또한 시스템 상에서의 우선순위 수가 정해지면 명령어 수도 정해지므로 시간 결정성을 보장한다. 예를들어, 우선숭위의 수가 1024인 경우 제안한 알고리즘에서는 최상위 우선순위를 결정하기 위해서 19개의 명령어를 수행하면 된다. 또한, 우선순위 증가에 따른 언맵 테이블 증가로 인한 메모리 오버헤드가 존재하지 않으므로 임베디드 시스템에 적용하기에 적합하다.

Ⅲ. 결 론

내장형 실시간 시스템은 기존의 산업, 군사 분야에서 시스템의 소형화, 성능 증가와 발맞추어 많은 양의 정보처리 기능을 가진 다양한 응용 분야로 발전하고 있으며, 그 적용사례도 다기능의 정보가전, 휴대폰, PDA, PMP, MP3 등과 같은 휴대용 기기로 확대되고 있다. 이들 실시간 응용장치들이 복잡한 응용프로그램을 구동함에 따라 실시간 운영체제의 시스템 콜 수행의 정확한 시간과 각 태스크의 마감시간내의 수행을 요구하고 있다. 또한, 시스템의 업무수행 과정 중 스케줄링은 가장 빈번이 일어나며 이 특성에 따라서 시스템 성능에 영향을 줄 수 있다.

본 논문에서는 기존의 논문들이 가정을 통하여 무시하였던 스케줄링을 위한 부가소요시간이 미치는 영향을 분석하였고, 이 과정을 통하여 운영체제 구현 시 사용하였던 스케줄링을 위한 최우선 순위의 선택에 필요한 시간 결정성 관점에서 시간복잡도 $O(n)$ 으로 증가하는 것을 단순한 고정수의 연산만을 통하여 시간복잡도 $O(1)$의 시간에 우선순위를 결정하는 방법을 제안하였다. 또한, 태스크의 우선순위 수의 증가가 미치는 메모리의 공간 복잡도 $O(2^{2^n})$형태로 기하급수적인 증가를 가져오는 제한사항을 기존의 스케줄링 방법이 갖는 시간결정성의 장점을 그대로 유지하면서, 추가적인 메모리 사용 없이 기존의 최대 태스크 수에 의한 제약을 제거한 일반화된 시간결정성을 갖는 스케줄링 방법을 제안하였다.

참 고 문 헌

[1] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," Proc. of the IEEE, Vol.82, No.1, pp.6-24, 1994.

[2] K. M. Zuberi, P. Pillai, and K. G. Shin, "EMERALDS: A small-memory real-time microkernel," Proc. 17th ACM Symposium on Operating Systems Principles, 1999.

[3] J. J. Labrosse, C/OS: The Real-Time Kernel, R&D Pub, 1993.

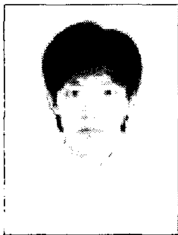
[4] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating system support for real-time systems," Proc. of IEEE, Vol.82, No.1, pp.55-67, 1996.

[5] C. M. Krishna and K. G. Shin, Real-Time Systems, McGraw-Hill Pub, 1997.

[6] S. J. Oh, J. N. Kim, Y. R. Seong, and C. H. Lee, "Deterministic Task Scheduling for Embedded Real-Time Operating Systems," IEICE Trans. Inf.&Syst., Vol.E87-D, No.2, pp.123-126, 2004.

저 자 소 개

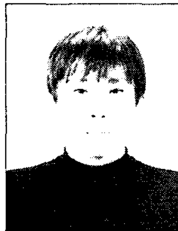
조 문 행(Moon-Haeng Cho) 정회원



- 2004년 2월 : 충남대학교 컴퓨터공학과 (공학사)
- 2006년 2월 : 충남대학교 컴퓨터공학과 (공학석사)
- 2006년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정 재학

<관심분야> : 실시간 컴퓨팅, 실시간 운영체제, 초소형 초절전 실시간 운영체제

이 숭 열(Soong-Yeol Lee) 준회원



- 2004년 2월 : 충남대학교 컴퓨터공학과 (공학사)
- 2005년 9월 ~ 현재 : 충남대학교 컴퓨터공학과 (석사과정) 재학 중

<관심분야> : 실시간 운영체제, 임베디드 시스템

이 원 용(Won-Yong Lee) 준회원

- 2004년 2월 : 원광대학교 컴퓨터정보통신공학과 (공학사)
- 2005년 9월 ~ 현재 : 충남대학교 컴퓨터공학과 (석사과정) 재학 중

<관심분야> : 실시간 운영체제, 임베디드 시스템

정 근 재(Geun-Jae Jeong) 준회원

- 2006년 2월 : 충남대학교 컴퓨터공학과 (공학사)
- 2006년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 (석사과정) 재학 중

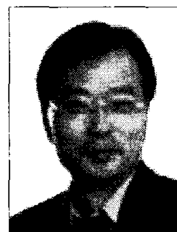
<관심분야> : 실시간 운영체제, 임베디드 시스템

김 용 희(Yong-Hee Kim) 정회원

- 2003년 2월 : 충남대학교 컴퓨터공학과 (공학석사)
- 2003년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정 재학

<관심분야> : 실시간 시스템, 실시간 운영체제, 고장허용 컴퓨팅

이 철 훈(Cheol-Hoon Lee) 정회원



- 1983년 2월 : 서울대학교 전자공학과 (공학사)
- 1988년 2월 : 한국과학기술원 전기및전자공학과 (공학석사)
- 1992년 2월 : 한국과학기술원 전기및전자공학과 (공학박사)
- 1983년 3월 ~ 1986년 2월 : 삼성전자 컴퓨터사업부 연구원
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터사업부 선임연구원
- 1994년 2월 ~ 1995년 2월 : Univ. of Michigan 객원연구원
- 1995년 2월 ~ 현재 : 충남대학교 컴퓨터공학과 교수
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 초빙연구원

<관심분야> : 실시간시스템, 운영체제, 고장허용 컴퓨팅