이벤트를 고려한 효과적인 세마포어 구현 알고리듬

한기희*, 신봉식**, 피찬호*, 정정화*
*한양대학교 정보통신학과
**한양대학교 전자공학과
e-mail : [hanq73, sbs69]@korea.com

An Efficient Semaphore Implementation Scheme with Event

Ki-Hee Han*, Bong-Sik Sihn**, Chan-Ho Pi*, JongWha Chong*
*Dept. of Information & Communication, Han-Yang University
**Dept. of Electronic Engineering , Han-Yang University

Abstract

In this paper, we present a novel semaphore implementation scheme which shortens finish time of high priority tasks and improves reliability of a system. The real-time systems have time constraints. Especially, the task with hard real-time constraints must meet its deadline. Consequently, managing shared resources is considered guaranteeing mutual exclusion as well as meeting task's deadline under unfavorable condition. According to the number and sort of the locked semaphores under the event occurred, this paper presents the reduction of the finish time of high priority task by decision whether the context switched or not. The experimental results show that the proposed method gives performance improvements in finish time of high priority tasks of about 11% over zuberi[4] method.

1. 서론

실시간 시스템은 마감시간(deadline), 릴리스타임(release time), 실행주기(period)와 같은 태스크의 실행에 관계된 시간적 제약을 가지고 마감시간까지 정확한 결과값이 나와야 하는 시스템이다. 특히 하드 실시간 시스템에서는 어떠한 상황에서도 마감시간까지 정확한 결과값이 나와야 한다.[1] 실시간 시스템의 정확한 결과값은 언제 태스크를 수행시키고 중간결과를 언제 전달해야 하는가에 관계가 있다.

일반적으로 태스크들이 공유자원을 사용하고자 할 때 세마포어(semaphore)를 이용하여 상호배타적(mutual exclusive)으로 사용함으로써 태스크 간의 동기(synchronization)를 맞출 수가 있다[2][3][4][5][6][7].

Sha[2][3]는 태스크 우선순위 반전(unbounded priority inversion)문제에 대해 BPI(basic priority inheritance protocol)와 우선순위가 높은 태스크가 두 개 이상의 세마포어를 필요로 할 때 발생하는 연속 블록(chain blocking)문제에 대해 PCP(priority ceiling protocol)를 이용하여 해결되도록 하였다. Zuberi[4]은 세마포어를 사용함으로써 발생하는 문맥교환의 수를 줄임으로써 커널 오버헤드(kernel overhead)를 줄이는 방향을 제시했다. 이 방식은 내장형 실시간 시스템에 적합하게 Sha 의 방식 보다 적은 메모리를 사용하더라도 효과적으로 태스크 간의 상호배타적인 자원사용을 할 수 있도록 한다. 그러나, 이 방식들은 이벤트(event)를 고려한 상황에서 우선순위가 높은 태스크의 실행 마감시간이 길어지게 되는 단점이 있다.

본 논문에서는 이벤트를 고려한 상황에서 lock 상태에 있는 세마포어의 종류와 개수에 따라서 문맥교환 여부를 결정함으로써 우선순위가 높은 태스크에 대한 마감시간을 단축시킴으로서 우선순위가 높은 태스크에 대한 신뢰성을 높이게 하였다.

2 장 태스크 모델링, 3 장 Basic priority inheritance(BPI) 와 Zuberi[4]에 의한 스케줄링 방법, 4 장에서는 제안하는 새로운 스케줄링 방법, 5 장 제안한 방법에 대한 실험결과, 6 장 결론 순으로 구성된다.

2. 태스크 모델링

1) 태스크 집합은 다음과 같이 정의한다.
T = { T1,...,Tn}.
- 태스크 번호가 빠를수록 우선순위는 높다.
Ex) 우선순위 : T1 > T2>...>Tn
- Rn : Tn 이 실행가능한 시간

2) 세마포어 집합은 다음과 같이 정의한다.
S = {S1,...,Si}
- Si^{Tn} : Si 이 Tn 에 Lock 되어 있는 상태
- Si_{free} : Si 에 대해 Tn 이 사용가능한 상태

3. 기존의 스케줄링 방법

3-1. BPI(Basic priority inheritance protocol)

낮은 우선순위의 태스크(T_L)가 Si^n 인 상태에서 높은 우선순위를 가진 태스크(T_H)가 발생해(release) T_L 이 가지고 있는 Si^n 을 요구할 때 태스크의 우선순위와 관계없이 T_L 이 Si^n 을 넘겨 줄(unlock) 때까지 T_H 는 대기 상태에 있게 된다. 이 때 Si^n 와 관련이 없는 중간 우선순위를 갖는 태스크(T_M)가 발생했다면 T_H 는 T_M 이 다 수행될 때까지도 기다려야 한다. 이와 같이 T_H 가 우선순위가 낮은 태스크(TM, TL)에 의해 실행이 중지되는(block)되는 현상을 우선순위반전(unbounded priority inversion)이라 하며, 블록이 되는 기간을 최소로 하기 위해 일반적으로 BPI 를 사용한다.

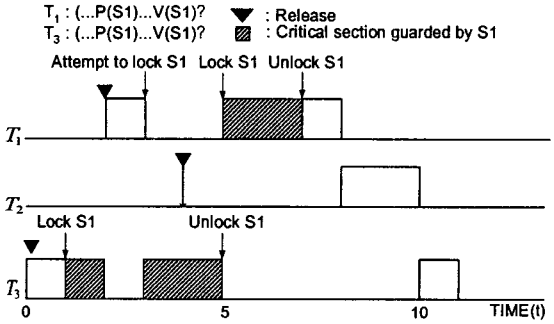

그림 1 BPI (basic priority inheritance protocol)

그림 1 은 BPI 에 대한 그림으로써 태스크의 우선순위는 T1>T2>T3 의 순서이며 높은 우선순위를 갖는 태스크는 발생(release) 즉시 선점할 수 있다(Preemptive).

그림 1 으로 BPI(basic priority inheritance protocol)에 의한 태스크의 수행 과정을 보면 다음과 같다.

1. T3 가 $S1^{T3}$ 인 상황에서 t_2 에 T1 이 발생한다(R1=t_2).
2. t_3 에서 T1 이 S1 을 요구한다. 이때 자원을 갖고 있는 T3 의 우선순위를 T1 만큼 올린다(inheritance). 즉 T3 는 T1 만큼의 우선순위를 가지고 수행된다.
3. t_4 에서 R2 가 된다. 그러나 T1, T3 보다 우선순위가 낮음으로 대기상태가 된다.
4. t_5 에서 $S1^{T3}$ 가 Unlock 상태($S1_{free}$)가 되는 동시에 T3 는 원래의 우선순위로 돌아가고 $S1^{T1}$ 상태가 된다. T1 이 수행된다
5. t_8 에서 T1 은 모두 수행되고 그 다음 우선순위를 갖는 T2 가 실행되고 t_{10} 에서 T3 가 수행된다.

3-2 Zuberi[4]에 의한 스케줄링 방법

Zuberi[4]는 우선순위계승을 사용하여 높은 우선순위를 가진 태스크의 수행이 블록 되는 것을 최소로 하는 동시에 여분의 문맥교환의 수를 줄여 커널 오버헤드를 줄이도록 하였다. 즉, T_L가 Si^n인 상태에서 T_H가 발생됐을 때 Si^n을 필요로 하는 부분이 있는지 없는지를 미리 판단하여 Si^n을 필요로 하는 부분이 있다면 T_H 을 일시적으로 지연을 시켜 여분의 문맥교환을 줄인다.

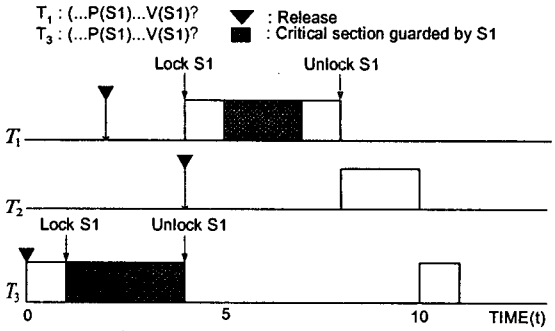

그림 2 Zuberi[4]의 방식에 의한 스케줄링(1)

그림 2 으로 Zubreri[4]에 의한 태스크 수행 과정을 보면 다음과 같다.

1. T3 가 $S1^{T3}$ 인 상황에서 t_2 에 T1 이 발생한다. 이때 T1 이 $S1^{T3}$ 을 요구하는지 검사를 한다. 세마포어를 요구하지 않거나 $S1^{T3}$ 와 다른 세마포어를 요구한다면 우선순위 방식에 따라 T1 을 그대로 수행하고 $S1^{T3}$ 을 요구한다면 T1 을 일시적으로 지연(delay)시키는 동시에 T3 의 우선순위를 T1 만큼 올린다.(priority inheritance)
2. t_4 에서 $S1_{free}$ 상태로 되자마자 T3 의 우선순위는 원래대로 돌아간다. T2 가 발생되지만 우선순위가 낮기에 수행하지 않고 우선순위가 높은 T1 이 수행된다.
3. t_8 에서 T1 이 모든 실행을 마치고 T2 가 수행되고 t_{10} 에서 T3 의 나머지 부분이 실행된다.

그림 1 과 그림 2 를 통해 Zuberi[4]의 방식으로 스케줄링 할 때 모든 태스크가 BPI 와 동일한 끝나는 시간을 갖고도 BPI 에서는 3 개의 태스크를 실행하는데 5 개의 문맥교환이 발생하는데 비해 Zuberi[4]의 방식에서는 단지 3 번의 문맥교환이 발생한다. 이는 T_H 가 $Si^{Tn}(n > H)$ 인 상태에서 세마포어를 요구할 때 세마포어와 관계가 없는 부분을 먼저 수행시키지 않고 세마포어 이용이 가능 할 때에 태스크를 한번에 수행시킴으로써 가능하게 된다. 그러나 이벤트를 고려한 상황에서 세마포어 사용시 높은 우선순위를 가진 태스크가 계속 세마포어를 기다리게 되는 상태가 되어 T_H 가 블록되는 시간이 길어지는 현상이 발생할 수 있게 된다. 이에 본 논문에서는 이러한 문제점을 해결할 수 있는 새로운 세마포어 구현 방식을 제안하고자 한다.

4. 제안하는 이벤트 상황을 고려한 스케줄링 방법

본 논문에서 제시하는 방식은 선점형 커널(preemptive kernel)의 사용과 우선순위계승방식(priority inheritance)의 사용 등 기본적 자원관리 정책과 이벤트를 고려한 스케줄링 방식이다.

이벤트에 의한 다양한 블록상황이 발생할 수 있다. [4]

- 내부적인 이벤트 : 태스크 수행의 중간의 값을 요구하거나 태스크 수행자체에서 어느 일정시간까지의 지연(delay)를 요구
- 외부적인 이벤트 : 비주기적인 태스크(aperiodic task)나 주기적인 태스크(periodic task)가 된다.

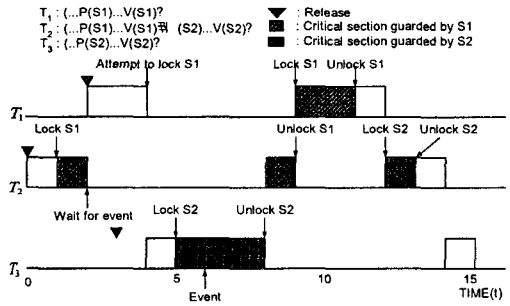

그림 3. Zuberi[4]의 방식에 의한 스케줄링(2)

그림 3 는 2 개 이상의 세마포어를 사용하고 이벤트를 고려한 상황에서 Zuberi[4]의 방법을 사용했을 때의 스케줄 상태를 나타낸 그림이다.

1. t_0 에서 S1 를 요구하는 T2 가 발생하고 S1 을 얻고 수행하게 된다.($S1^{T2}$)
2. t_2 에서 T2 는 내부적인 이벤트를 기다리는 상태가 된다. 이때 S1 을 요구하는 T1 이 발생한다. Zuberi[4]의 방식에서 T2 가 이벤트에 의해 기다리는 상황이 아니고 단지 S1 을 기다리는 높은 우선순위를 가진 T1 이 나타났을 때 강제로 T1 을 지연시키겠지만 T2 가 이벤트를 기다리는 상태에서 수행가능 한 태스크는 T1 이 되므로 T1 을 수행시킨다.
3. t_4 에서 T1 은 S1 을 기다리는 상태가 되고, T3 를 수행시킨다. T3 는 S2 를 가져야 할 때 S2 는 사용가능 함으로 S2 를 가지고 실행한다.($S2^{T3}$)
4. t_6 에서 이벤트가 발생한다. 그러나 S2 사용도중 발생하였으므로, (T2 가 S2 를 필요로 하는 부분이 있다.) T3 의 S2 를 사용하는 부분을 마저 수행을 시킨 후에 T2 로 문맥교환을 함으로써 문맥교환 횟수가 줄어든다.
5. t_8 에서 $S1_{free}$ 상태가 되면 S1 을 기다리는 T1 이 수행되고 T2, T3 의 순으로 마저 실행된다.

위의 태스크 수행과정은 Zuberi[4]의 방식으로 event 를 적용하고 2 개 이상의 세마포어를 사용할 때 우선순위가 높은 태스크의 수행이 늦어 질 수 있음을 보여준다. 즉 T2 에서 원하는 이벤트가 T3 수행도중 발생했을 때, T3 의 수행을 중지 시키고 바로 문맥교환을 시켜, T2 를 수행시킨다면 T1 도 빨리 수행시킬 수 있었으나, 이벤트 발생 당시 세마포어(S2)를 기다리고 있는 태스크(T2)가 있기에 문맥교환을 시키지 않고 그냥 수행해버려 발생한 결과이다.

이에 본 논문은 이벤트 발생을 기다리는 태스크가 있고, 그 이벤트가 발생했을 때의 태스크의 부분에 자원이 포함되어 있다면, 다음과 같은 검사를 하도록 하였다.

```
If (include S2)
   If ( semaphore number > 1 )
      Context switch
   Else
      Continue
Else
   Context switch
```

그림 4 Check routine 의 의사코드

앞으로 수행할 태스크(이벤트가 발생할 당시 태스크를 제외한 이벤트 발생 당시 wait queue 에 저장되어 있는 태스크로서 그림 3 에서는 T1, T2 가 된다.) 중에서 현재 포함되어 있는 세마포어(S2)를 가지고 수행할 예정이거나 현재 세마포어(S2)를 기다리는 task 가 있는지 검사한다. (lock 상태에 있는 세마포어의 종류 검사)

기다리는 태스크가 있다면 현재 lock 상태에 있는 세마포어의 개수를 검사한다. 다음 그림 5 는 그림 4 에 대한 흐름도이다.

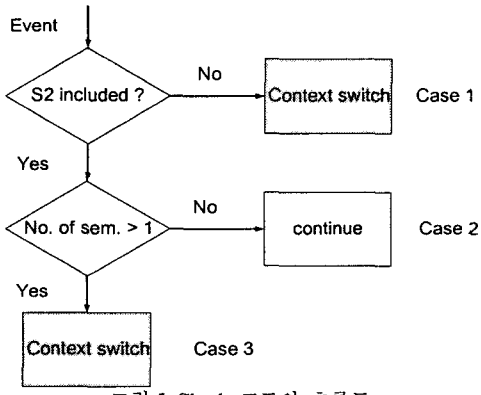

그림 5. Check 코드의 흐름도

위 그림에서 Case 1 과 2, 3 을 볼 수 있으며, 각각은 다음과 같다.

Case 1: 앞으로 수행할 태스크에 세마포어(S2)를 가지고 수행할 예정이거나 현재 세마포어(S2)를 기다리는 태스크가 없다면 이벤트가 발생한 현재 수행하고 있는 우선순위가 낮은 태스크(T3)를 수행 할 필요가 없다. 우선순위가 높은 태스크들이 기다리고 있기 때문이다.

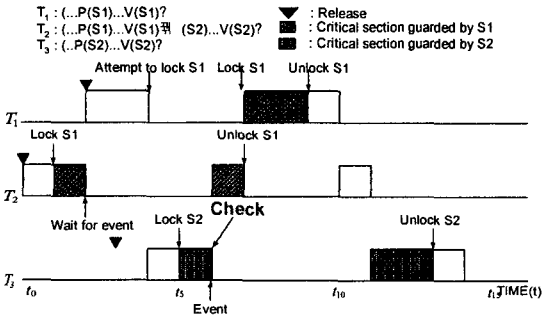

그림 6. Case 1

Case 2: 앞으로 수행할 태스크에 세마포어(S2)를 가지고 수행할 예정이거나 현재 세마포어(S2)를 기다리는 태스크가(T2) 있지만 현재 Lock 상태에 있는 세마포어가 단 한 개라면(이 한 개는 S2 일 것이다) Zuberi[4]의 방식를 사용하여 문맥교환을 하지 않고 T3 를 수행한다. (T2 를 먼저 수행시키더라도 세마포어(S2)를 기다려야 하기 때문에 마감시간은 T3 를 먼저 수행시키고 나중에 수행시키는 것과 똑같다)

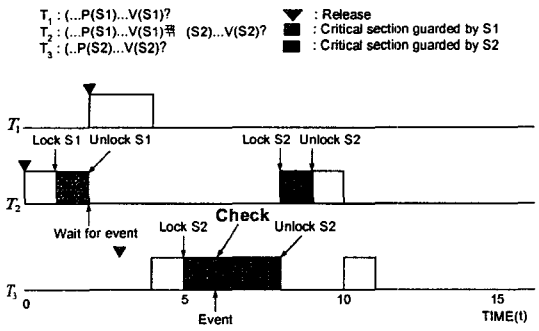

그림 6 Case 2

Case 3 : 앞으로 수행할 태스크에 세마포어(S2)를 가지고 수행할 예정이거나 현재 세마포어 S2 를 기다리는 태스크가 있고 현재 Lock 상태에 있는 세마포어가 한 개 이상이라면 문맥교환의 수가 많아 지더라도 다른 태스크들의 빠른 수행을 위해 문맥교환을 시킨다.

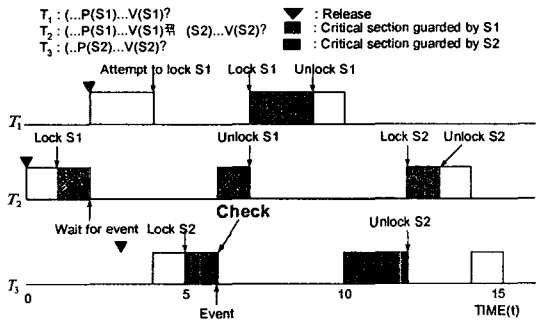

그림 7 Case 3

5. 모의실험결과

본 논문에서는 Case 3 의 경우에 제안하는 방법에서의 높은 우선순위를 가진 태스크의 수행이 마치는 시간과 Zuberi[4]가 제안한 방식를 사용하여 나중에 문맥교환을

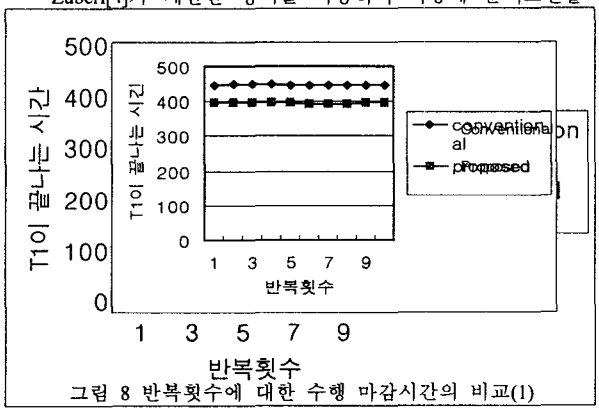

그림 8 반복횟수에 대한 수행 마감시간의 비교(1)

그림 8 은 Case 3 에서 제안된 방법을 사용했을 때와 Zuberi[4]의 방법을 사용했을 때, 높은 우선순위를 가진 태스크의 마치는 시간을 비교한 것이다. 100 번부터 1000 번까지 임의의(random) 실행시간으로 반복수행 시켰다.

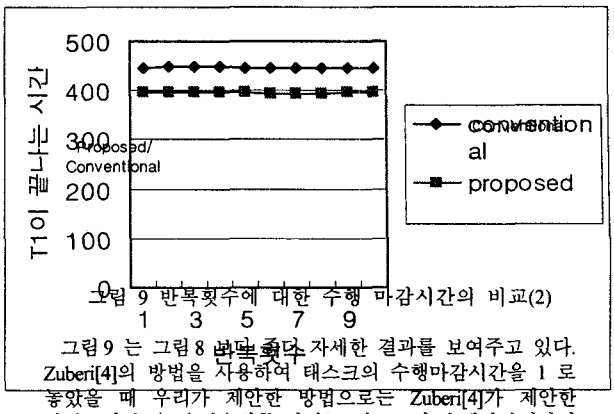

그림 9 반복횟수에 대한 수행 마감시간의 비교(2)

그림 9 는 그림 8 의 좀 더 자세한 결과를 보여주고 있다. Zuberi[4]의 방법을 사용하여 태스크의 수행마감시간을 1 로 놓았을 때 우리가 제안한 방법으로는 Zuberi[4]가 제안한 방법보다 높은 우선순위를 가지는 태스크의 수행마감시간이 평균 11%정도 줄어든 것을 알 수 있다.

6. 결론

본 논문에서는 이벤트 발생시 우선순위가 높은 태스크의 마감시간을 단축시키기 위한 새로운 스케줄링 방법을 제안 하였다. 기존의 자원관리 방식들은 태스크의 수행도중의 이벤트를 고려하지 않는 상황에서의 자원관리 방식이어서 자원을 요구하는 우선순위가 높은 태스크가 지연 됨으로써 마감시간이 길어지는 단점이 있었다. 본 논문에서는 어떠한 이벤트가 발생했을 때의 상황에 맞게 우선순위가 높은 태스크에 대한 마감시간이 길어질 수 있는 경우에서는 문맥교환을 함으로써 우선순위가 높은 태스크에 대한 마감시간을 단축시켰다. 모의실험결과 Zuberi[4]가 제안한 방식보다 높은 우선순위의 태스크의 마감시간을 11% 정도 단축시켰다.

참고문헌

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, 1973

[2] L. Sha, R. Rajkumar and J. Lehoczky, "Priority inheritance protocols : an approach to real-time synchronization, *IEEE Trans. on Computer*, vol. 39, no.3, pp.1175-1198, 1990

[3] J. B. Goodenough and L. Sha, "The priority ceiling protocol : A method for minimizing the blocking of high priority Ada tasks," in *Proc. 2 nd ACM Int. Workshop Real-Time Ada Issues*, 1988

[4] K. M. Zuberi and K. G. Shin, "An efficient semaphore implementation scheme for small-memory embedded systems," *IEEE Trans. on Computer*, pp.25-34, 1997

[5] C. D. Wang, H. Takada and K. Sakamura, "Priority inheritance spin locks for multiprocessor real-time systems," in *2 nd International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 70-76, 1996

[6] H. Takada and K. Sakamura, "Experimental implementations of priority inheritance semaphore on ITRON-specification kernel," in *11th TRON Project International Symposium*, pp. 106-113, 1994

[7] A. Terrasa, A. Garcia-Fornes, "Real-Time Synchronization between Hard and Soft Tasks in RT-Linux," *IEEE Trans. On Real-time Computing Systems and Applications*, pp. 303-310, 1999

[8] C. L. Liu, "Real-Time Systems," *Prentice Hall*, 2000

[9] S. H. Son, "Advances in real-time systems," *Prentice Hall*, 1995