RT-Linux의 개선된 스케줄링 방안

An Improved Scheduler of RT-Linux

요 약

실시간 시스템은 논리적 정확성은 물론 시간적 정확성을 요구한다. 실시간 시스템을 위한 운영체제는 실시간 시스템의 지원을 위한 전용 실시간 운영체제와 범용 운영체제인 유닉스 계열 운영체제를 확장한 것으로 구분된다. 유닉스 계열 운영체제를 확장한 형태인 RT-Linux는 RMS와 EDFS 두 가지의 스케줄러가 구현되어 있지만, 많은 보완과 개선이 필요하다. 본 논문에서는 RT-Linux의 스케줄러가 갖고 있는 문제점을 개선해 보고자 두 가지 방안을 보여준다. RT-Linux는 RMS와 EDFS 두 가지 스케줄러 중에서 각각의 스케줄러의 특성을 고려하지 않고 두 가지 방법 중 하나를 임의로 선택하여 사용함으로써 종료시한 miss rate, 시스템 정지 현상 등의 문제점이 발생하게 된다. 이 문제점을 해결하기 위해, 먼저 스케줄 가능성 검사를 통해 스케줄링 하는 방법(SPC방법)을 제안하고, 특정 실시간 태스크의 시그널이 발생하였을 경우 시그널 처리에 있어 지연 시간을 늘리게 된다는 문제점을 해결하기 위해, ISBC방법을 이용한 좀 더 효율적인 시그널 함수의 구현을 제안한다. 이 두 방법을 사용함으로써 종료시한을 보장하고, 잘못된 스케줄로 인해 발생되는 시스템 정지 현상을 제거하였으며, 응답시간의 지연을 줄일 수 있었다.

키워드 : SPC방법, ISBC방법

Abstract

The real-time system demands not only logical correctness but also timely correctness. There are two types of operating systems for the real-time system. The first one is a dedicated RTOS for the real-time systems and the second one is an extension of Unix-like OS. The second type OS can use the various services and the already familar developing environment supplied by Unix. The RT-Linux, an extension of Unix-like OS, have been implemented by two scheduler such as RMS and EDFS, needed improvement. In this paper, we show two method to improve problem of RT-Linux scheduler. First, we propose SPC method through schedule possibility check to solve problem that characteristics of each scheduler have not been considered but used random selection. Second, we suggest more improved design of signal function through ISBC method to solve problem that latency time has been increased to handle signal by check signal bit of all task.

keywords : SPC method, ISBC method

1. 서론

실시간 시스템(Real-Time System)은 기존의 범용 시스템과는 달리 시스템 동작의 논리적인 정확성은 물론 시간적인 정확성을 중요시하는 시스템이다. 시간적 제약 조건은 종료시한(deadline)으로 주어지며, 시간적 제약 조건의 엄격성에 따라 시간적 제약이 지켜지지 않으면 치명적인 손상을 가져오는 경성(hard) 실시간 시스템과 제약이 지켜지지 않더라도 치명적인 영향은 발생하지 않고 다만 비효율적인 결과를 초래하게 되는 연성(soft) 실시간 시스템으로 구분된다.

실시간 시스템의 구현을 위해서 많은 실시간 운영체제(Real-Time Operation System, 이하 RTOS)가 제작되었는데, 실시간 시스템의 지원을 위해서 전용으로 제작된 것과 실시간 시스템의 지원을 위해서 범용 운영체제인 유닉스(UNIX)계열 운영체제를 확장한 것으로 구분할 수 있다. 유닉스(UNIX)계열 운영체제를 확장한 시스템에서는 유닉스 계열 운영체제에서 제공되는 대부분의 서비스를 그대로 사용할 수 있다는 장점이 있으며, 유닉스 계열 RTOS에는 Solaris, IRIX, MINIX, LynxOS, Linux 등이 있다. 이 중 POSIX.1b 중 일부만을 지원하기 때문에, 연성실시간 시스템의 처리에 적합한 형태인 Linux를 New Mexico 기술연구소에서 기존 Linux에 최소의 변경만을 가해서, 경성실시간 시스템을 지원해주는 RT-Linux를 개발하였다. RT-Linux에서는 실시간 태스크의 개발을 위한 인터페이스를 제공해주고는 있지만, 아직 다양한 스케줄러의 지원이 부족하다. RMS와 EDFS 두 가지의 스케줄러가 구현되어 있지만, 많은 보완과 개선이 필요하다.

RT-Linux의 스케줄러가 갖고 있는 문제점 중 두 가지를 개선해 보고자 두 가지 방안을 제안한다. 첫째, RT-Linux가 RMS와 EDFS 두 가지 스케줄러 중에서 각각의 스케줄러의 특성을 고려하지 않고 두 가지 방법을 임의로 선택하여 사용하게 함으로써 발생하는 문제점을 해결하기 위해, 스케줄링 가능성 검사를 통해 스케줄링 하는 방법(SPC방법)을 제안한다. 둘째, RT-Linux는 특정 실시간 태스크의 시그널이 발생하였을 경우 이 시그널에 대한 핸들러를 수행시키기 위해서는 현재 태스크 목록에 추가되어 있는 모든 태스크들의 시그널 비트를 검사하게 되어, 시그널 처리에 있어 지연 시간을 늘리게 된다는 문제점을 해결하기 위해, ISBC방법을 이용한 더 효율적인 시그널 함수의 구현을 제안한다. 이 두 방법을 사용함으로써 종료시한을 보장하고, 잘못된 스케줄로 인해 발생되는 시스템 정지 현상을 제거할 수 있으며, 응답시간의 지연을 줄일 수 있다.

2. 관련연구

현재, RT-Linux에는 RMS와 EDFS 두 가지의 스케줄러가 구현되어 있으며, 이 두 가지 스케줄러 중에서 사용자가 각각의 스케줄링 알고리즘의 특성을 고려하지 않고 두 가지 방법을 임의로 선택하여 사용하도록 하고 있다. 이로 인해 시스템의 스케줄링 가능성 검사가 이루어지지 않아 종료시한 miss rate를 증가시키게 되고, 스케줄 불가능한 태스크에 대한 스케줄로 인해 시스템이 정지되는 치명적인 현상이 발생하게 된다. 이런 현상을 방지하기 위해서 본 논문에서는 스케줄 가능성 검사(Schdule Posibility Check : SPC)방법을 통해 스케줄러를 사용하는 방안을 제안한다. 또한, RMS에서는 각 태스크들을 독립적이라고 가정하여 각 태스크에게 정적(static)으로 우선순위를 부여한다. 이때, 주기가 짧은 태스크일수록 높은 우선순위가 주어지게 되는데, 이 알고리즘은 우선순위가 고정적으로 주어진 태스크 집합들에 대한 스케줄링 알고리즘들 중 최적의 알고리즘으로 알려져 있다. 단일 프로세서 환경에서 RMS를 사용하는 경우 태스크 집합의 스케줄링가능성은 주어진 주기적인 태스크들의 집합의 프로세서이용률(processor utilization)이 다음을 만족하면 스케줄링이 가능하다 간주한다.

$$U = \sum_{i=1}^{n} \frac{Ci}{Ti} \leq n(2^{\frac{1}{n}} - 1) \quad ; \text{n은 태스크 수}$$

위의 식으로 확인되지 못하는 경우는 동시에 시작된 각 태스크의 첫 번째 job의 완료시간을 구한 값이 종료시한보다 작은 가의 여부로 확인될 수 있다. 그러나, 태스크 집합의 프로세서 이용률이 $n(2^{\frac{1}{n}} - 1)$보다 크고 1보다 작으면 RMS방식으로는 태스크 집합이 스케줄링 가능한지 판단을 할 수 없다.

EDFS에서는 각 태스크에게 우선순위가 동적으로 부여되는데, 매 스케줄링 시점에서 볼 때 종료시한까지의 거리가 가장 짧은 태스크에게 높은 우선순위가 주어지는 선점 가능 스케줄링 알고리즘이다. 실시간 시스템에서 태스크들이 종료시한 안에 수행된다고 보장받는다면 그 태스크 집합은 스케줄링 가능하다고 한다. EDFS에서는 다음의 조건이 만족되면 태스크들을 스케줄링 가능하다고 할 수 있다.

$$U = \sum_{i=1}^{n} \frac{Ci}{Ti} \leq 1$$

또한 태스크가 주기 이전에 종료시한을 가진다면 Ci ≤ Ti - Di 또는 Ci + Di ≤ Ti의 조건을 만족하게 된다.

3. 제안하는 방법

알고리즘1을 이용하여 스케줄링 가능성 검사를 행한 다음 RMS와 EDFS 중 수행하고자 하는 태스크의 특성에 맞는 스케줄러를 선택하여 태스크들을 수행시킴으로써 태스크들의 종료시한을 보장하고, 스케줄이 불가능한 경우 태스크 스케줄로 인해 발생되는 시스템 정지 현상을 방지하게 된다.
RT-Linux에서 사용하고 있는 스케줄링 방법은 우선순위를 기반으로 한 선점 가능한 스케줄러를 사용한다. 여기서 우선순위는 실시간 태스크가 생 성될 때, 각 태스크는 고정된 우선순위를 할당받는다. 만약 현재 수행중인 태스크의 우선순위보다 높은 우선순위의 태스크가 있을 경우 바로 선점이 가능하다. 또한, 기존 Linux 커널은 가장 낮은 우선순위를 가지게 되고 수행될 실시간 태스크가 없을 경우에만 수행될 수 있다.
현재 RT-Linnux에서 사용하고 있는 스케줄러의 함수 부분은 생성된 모든 실시간 태스크들과 기존의 Linux 태스크들을 모두 하나의 목록에 추가시키고 RT-Linux가 사용하는 timer가 발생할 때마다 그 목록을 모두 검사하여 가장 높은 우선순위를 가지는 태스크를 찾는 방법을 사용하고 있다.

알고리즘2. SPC방법의 기본알고리즘

```
bool SchedulableCheck(int SetOfTasks)
{       for(i=0;i<SetOfTasks;i++)
        {
                ProcessorUtilization+=(TaskValue[i].computetime/TaskValue[i].period);
        }
        bound=SetOfTasks*(POW(2,(1/SetOfTasks))-1);
        if(ProcessorUtilization)>=0&&
                ProcessorUtilization<=bound)
                schedulingOK = true;
        else if(ProcessorUtilization>=0&&
                ProcessorUtilization <=1)
                schedulingOK = true;
        else
        {
                schedulingOK = false
                clean_module();
        }
        return SchedulingOK;
        }
```

목록에 포함되어 있는 태스크의 개수가 적을 경우에는 문제가 없지만, 태스크의 개수가 많아지면 가장 큰 우선순위를 찾는데 걸리는 시간이 O(n)의 시간으로 커지게 되므로 종료시한을 놓치는 태스크들이 생길 수 있다. 또한, 특정 실시간 태스크의 시그널이 발생하였을 경우 시그널 핸들러를 수행시키기 위해서는 현재 태스크 목록에 추가되어 있는 모든 태스크들의 시그널 비트를 검사해야 한다. 물론 특정 시그널이 발생하지 않더라도 이 작업은 수행된다. 따라서, 이 작업은 시그널이 발생하고 처리되는 데까지 걸리는 지연 시간을 늘리게 되며, 실시간 시스템에서는 바람직하지 않다.
실시간 태스크는 일정한 주기를 단위로 ready와 delayed 상태를 반복하며 실행한다. suspend상태가 된 후에는 wakeup 시그널이 발생되면 깨어난다. RT-Linux는 태스크 목록에서 ready 상태의 태스크들 중에서 가장 우선순위가 높은 태스크를 수행한다.
RT-Linux의 스케줄링 함수는 실시간 태스크가 생성될 때 우선순위가 부여되고 태스크의 주기가 부여

되어 고정적인 우선순위를 가지는 주기적인 태스크가 생성된다. 태스크가 생성된 이후에 스케줄링 함수는 타이머 인터럽트가 발생할 때마다. rtl_task 목록의 처음부터 끝까지 검사하면서, 우선순위가 가장 높은 태스크를 찾는다. 타이머 인터럽트가 발생할 때마다 스케줄링 함수가 수행되고 수행될 때마다 전체 태스크 목록을 검사한다는 것은 상당히 비효율적인 작업이다. 태스크의 개수가 많으면 많을수록 스케줄링 하는데 걸리는 오버헤드는 상당히 크다. RT-Linux 스케줄러가 가장 큰 우선순위를 가진 태스크를 찾는 작업을 줄일 수 있는 알고리즘을 제안하고자 한다.

알고리즘3. 기존 태스크 목록 검사 알고리즘

```
void rtl_schedule(void)
{
        new_task = sched ->rtl_current;
        for(t=sched->rtl_tasks;t!=NULL;t=t->next)
        {
                if(t->state==RTL_THREAD_READY &&
                 (t->sched_priority>new_task->sched_priority))
                {
                        new_task = t ;
                }
                if(new_task != sched->rtl_current)
                {
                        current switch
                        rtl_switch_to(&sched->rtl_current, new_task);
                }
          }
```

우선 실시간 태스크들의 우선순위는 5단계로 한정하고 각 우선순위 별로 별도의 태스크 목록을 유지한다. 이 단계는 필요에 따라 늘어날 수 있지만 실제 시스템에서는 많은 우선순위의 단계를 필요로 하지 않으며 대신 동일한 우선순위를 많은 태스크들에 부여할 수 있게 함으로써 태스크 개수의 제한이 없으면서도 스케줄 함수를 효율적으로 구현할 수 있도록 한다. 현재 가장 높은 우선순위를 찾을 수 있는 전역변수를 하나 둔다. 이 변수는 태스크가 생성될 때마다, 태스크의 우선순위에 해당하는 비트가 설정된다. 설정된 값은 스케줄러가 실행되면서 만들어 놓은 우선순위 테이블에서 전역변수가 가지는 값을 참고로 하여 현재의 ready 상태의 태스크들에 대해 가장 높은 우선순위를 찾는다. 스케줄러가 가장 높은 우선 순위를 찾게 되면 이 우선순위에 해당하는 태스크 list의 가장 처음에 있는 태스크가 현재 수행중인 태스크보다 우선순위가 높으면 선점한다. 태스크가 ready 상태에서 delayed 상태로 바뀌면 ready 목록에서 제거하고 delayed 목록인 delta_list 목록에 만료시간 별로 정렬되어 추가된다. 이렇게 함으로써, 만료시간이 된 태스크들은 한꺼번에 원래의 ready 목록에 추가될 수 있다. 또한 suspend 된 태스크들은 suspend 목록에 추가되고, wakeup 시그널이 있으면 suspend 목록에서 제거하고 ready 목록에 추가한다.

알고리즘4. ISBC 검사 기본 알고리즘

```
void add_to_task_list(pthread_t thread)
{
        task_priority = thread->sched_priority;
        bit_mask l = priority_bit[task_priority];
        insert_task(thread, rtl_tasks[task_priority]);
}

void pthread_wait_np(void)
{
        rtl_current->state=RTL_THREAD_DELAYED;
        wait_priority = rtl_current->sched_priority;
        bit_mask &= priority_bit[wait_priority];
        add_to_delta_list(rtl_task[wait_priority]);
}

void rtl_schedule(void)
{
        highest_priority = pri_table[bit_mask];
        new_task=sched -> rtl_task[highest_priority];
}
```

이 제안된 방법을 사용하여 우선순위 기반의 RT-Linux에서 가장 높은 우선순위를 찾는 데 소요되는 시간을 줄이고, 시그널 핸들러를 수행시키는데 소요되는 지연시간을 단축시킬 수 있다.

4. 결론

실시간 운영체제에서 태스크 스케줄링은 매우 중요한 역할을 수행한다. 실시간 시스템에서는 각각의 태스크들마다 종료시한이 주어져 있고 이 태스크들이 종료시한 내에 수행되지 못하는 경우에는 큰 피해를 입을 수 있으므로 태스크의 종료시한은 지켜져야만 한다. 지금까지 종료시한을 보장하기 위한 여러 스케줄링 방법들이 연구되어 왔다.
RT-Linux는 이미 구현되어 있는 Linux를 기반으로 하여 경성 실시간 시스템을 구성한다.
현재까지 RT-Linux에는 스케줄러로써 RMS방식과 EDFS방법 두 가지가 구현되어 있다. 그러나, 이 구현된 스케줄러가 아직은 미흡한 점이 많다. 본 논문에서는 이 점을 보완하여 두 스케쥴러의 특성을 고려한 스케줄링 가능성 검사를 통한 스케줄러 방식인 SPC 방법과 우선순위 기반의 스케줄러의 가장 높은 우선순위를 찾기 위해 일일이 모든 태스크를 검사하는 비효율적인 면을 개선하기 위한 개선된 시그널 비트 검사 방식인 ISBC 방법을 제안하였다. 이 두 방법을 사용함으로써 종료시한을 보장하고, 잘못된 스케줄로 인해 발생되는 시스템 정지 현상을 제거하였다. 또한, 응답시간의 지연을 줄일 수 있었다.
앞으로의 연구 과제로서, 제안한 방법이 기존의 RT-Linux보다 어느 정도의 성능 개선이 있는지 다양한 환경에서의 평가가 필요하다.

참 고 문 헌

[1] Adelberg.B. Garcia-Molina.H. Kao.B, “Emulating soft Real-Time Scheduling Using Traditional Operating System Schedulers," *In Proceedings of IEEE Real-Time Systems Symposium*, 1994

[2] Barabanov.M, "A Linux-based Real Time Operating System," Jun 1997

[3] Corwin.B, "Adapting The Posix Standard To Real-Time," *Open Bus Systems'92*, Zurich, Oct.1992.

[4] Dhall.S.K. Liu.C.L, "On a Real-Time Scheduling Problem," *Operations Reasearch, Vol. 26, No. 1*, Jan-Feb. 1978.

[5] Epplin.Jerry, "Linux as an Embedded Operating System," *Embedded Systems Programming*, Oct. 1997

[6] Kleines.H. Zwoll.K, "Real Time Unix in Embedded Control - A Case Study within the Context of LynxOS," *IEEE Trans*. On Nuclear Science, vol. 43, No.1, Feb. 1996.

[7] Jean J. Labrosse, "MicroC/OS-II, The Real-Time Kernel," 1998

[8] Lehoczky, J.P., Sha, L., Y. Ding, "The Rate Monotonic Scheduling Algorithm : Exact Characterization and Average Case Behavior" *In Proceedings of the 10th Real-Time Systems Symposium*, 1989.

[9] Liu.C.L. Layland.J.W, "Scheduling algorithm for Multiprogramming in a hard Real-Time Environment," *JACM. vol.20,* 1973

[10] Mercer.C.W, "An Introduction to Real Time Operating System : Scheduling Theory," *School of Computer Science Carnegie Mellon University*, Nov. 1992.

[11] Sha.L. Rajkumar.R. Lehoczky.J.P, "Priority Inheritance Protocols : An approach to real-time synchronization," *IEEE transactions on Computers*, Sept. 1990.

[12] S-C Sheng, et al. "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey", Tutorial : Hard Real Time System, *IEEE Computer Society Press*, 1998

[13] Wainer.G.A, "Implementing Real-Time Services in MINIX," *Operating System Review, 29(3)*, July 1995.

[14] C.Warren, "Rate Monotonic Scheduling," *IEEE Micro*. Jun 1991.

[15] Yodaiken.V. Barabanov.M, "Real-Time Linux," *Linux Journal*, Mar. 1996

[16] Yodaiken.V, Barabanov.M, "RT-Linux Version Two," 1997.