Journal of Korea Multimedia Society Vol. 13, No. 9, September 2010(pp. 1285-1298)

비선점 환경의 TinyOS에서 실시간성을 고려한 태스크 그룹 기반의 스케줄링 기법

손치원[†], 탁성우[††]

요 약

비선점형 태스크 스케줄링 정책을 사용하는 TinyOS는 선입선출 (FIFO: First-In First-Out) 방식의 태스크 스케줄링만 제공하기 때문에 최상위 우선순위를 가진 사용자 태스크가 즉시 실행이 필요한 태스크임에도 불구하고 우선순위가 낮은 태스크가 획득한 CPU 사용권한을 선점하지 못한다. 따라서 실시간 서비스를 요구하는 사용자 태스크 (User Task)의 마감시한 (Deadline)을 보장할 수 없다. 또한, 비선점 환경의 TinyOS에서 사용자 태스크가 요청한 실시간 서비스를 완료하기 위해서는 사용자 태스크의 마감시한을 보장함과 동시에 사용자 태스크에서 호출 및 실행되는 TinyOS 플랫폼 태스크들의 마감시한도 보장해야 한다. 이에 본 논문에서는 비선점형 태스크 스케줄링 정책을 사용하는 기존 TinyOS 환경에서 실시간성을 제공하는 태스크 그룹 기반의 스케줄링 기법을 제안하였다. 제안한 기법은 요청한 사용자 태스크의 마감시한을 보장하기 위하여 사용자 태스크와 함께 사용자 태스크가 완료되기 위하여 호출 및 실행이 필요한 다수의 TinyOS 플랫폼 태스크를 태스크 그룹으로 형성한 후, 해당 태스크 그룹을 하나의 가상적인 큰 태스크 단위로 스케줄링한다. 제안한 기법의 동작을 시험한 결과, 제안한 기법은 비선점형 태스크 스케줄링 정책을 사용하는 TinyOS 환경에서 사용자 태스크의 마감시한을 보장함과 동시에 사용자 태스크의 평균 응답시간을 줄이고 기존 TinyOS 플랫폼간의 호환성을 제공할 수 있었다.

A Task Group-based Real-Time Scheduling Technique in the Non-Preemptive TinyOS

Chiwon Son[†], Sungwoo Tak[††]

ABSTRACT

Since the TinyOS incorporating a non-preemptive task scheduling policy uses a FIFO (First-In First-Out) queue, a task with the highest priority cannot preempt a task with lower priority before the task with lower priority must run to completion. Therefore, the non-preemptive TinyOS cannot guarantee the completion of real-time user tasks within their deadlines. Additionally, the non-preemptive TinyOS needs to meet the deadlines of user tasks as well as those of TinyOS platform tasks called by user tasks in order to guarantee the deadlines of the real-time services requested by user tasks. In this paper, we present a group-based real-time scheduling technique that makes it possible to guarantee the deadlines of real-time user tasks in the TinyOS incorporating a non-preemptive task scheduling policy. The proposed technique groups together a given user task and TinyOS platform tasks called and activated by the user task, and then schedule them as a virtual big task. A case study shows that the proposed technique yields efficient performance in terms of guaranteeing the completion of user tasks within their deadlines and aiming to provide them with good average response time, while maintaining the compatibility of the existing non-preemptive TinyOS platform.

Key words: Non-preemptive Scheduling(비선점형 스케줄링), Sensor Node Platform(센서 노드 플랫폼), Real-time Scheduling(실시간 스케줄링), TinyOS

※ 교신저자(Corresponding Author) : 탁성우, 주소 : 부산시 금정구 장전동 산30번지 부산대학교 정보컴퓨터공학부(609-735), 전화 : 051)510-2387, FAX : 051)515-2208, E-mail : swtak@pusan.ac.kr

접수일 : 2010년 2월 2일, 수정일 : 2010년 6월 8일
완료일 : 2010년 6월 29일

[†] 준회원, 부산대학교 컴퓨터공학과
(E-mail: sonchiwon@gmail.com)
[††] 종신회원, 부산대학교 정보컴퓨터공학부 부교수

※ 본 연구는 부산대학교 자유과제 학술연구비(2년)에 의하여 연구되었음.

1. 서 론

유비쿼터스 센서 네트워크 환경에서의 초소형 센서 노드는 PC나 PDA에 비해 매우 낮은 하드웨어 사양을 가지고 있음에도 불구하고 센싱 및 센싱된 데이터의 실시간 처리 작업, 그리고 노드간의 데이터 송수신 작업을 동시에 병행(Concurrency)해야 한다. 따라서 센서 노드에서는 효율적인 다중 작업의 제어 및 관리와 실시간성을 제공할 수 있는 경량 운영체제가 필요하다. 또한 무선 센서 네트워크에 대한 관심이 높아짐에 따라 무선 센서 네트워크를 구현하는 핵심 기술 중에서 센서 노드의 효율적인 운용 기술 개발이 진행되고 있다. 센서 노드는 운용 중 전력 공급이 어려우며, 매우 적은 용량의 메모리가 탑재된다는 하드웨어 제약을 가진다. 센서 노드의 이러한 제약 사항을 고려하여 제한된 하드웨어 자원 환경에서 센서 노드를 효율적으로 운용하기 위한 다수의 센서 노드 운영체제가 개발되었다. 대표적인 센서 노드용 운영체제로는 TinyOS [1], MANTIS [2], CONTIKI [3], 그리고 AvrX [4]가 있다. 이중에서 TinyOS는 다양한 센서 노드 하드웨어 플랫폼에 대한 이식 완료, 소스 공개 정책, 그리고 개발자들의 활발한 참여와 같은 강점을 바탕으로 전 세계적으로 가장 널리 사용되는 센서 노드용 운영체제로 자리 잡고 있다. 그리고 TinyOS에서 사용되는 컴포넌트 기반 언어인 NesC는 이벤트 구동 방식에 따른 동시적 작업 처리와 메모리 효율적인 코드 생성을 지원하도록 설계되었다[5]. NesC의 이와 같은 특징을 기반으로 TinyOS는 이벤트 기반 비선점형 스케줄링 정책을 구현하여 제한된 하드웨어 자원을 가지는 센서노드에 효율적인 성능을 제공한다. 최근 무선 센서 네트워크의 응용 서비스는 단순히 센서 노드들이 환경 데이터를 수집하고 전송하는 수준을 넘어서 센서 노드 스스로 상황 인지 서비스 및 위치 기반 서비스 등을 제공하는 수준으로 확대되고 있다. 또한, 센서 노드 운영체제는 공기 및 물의 오염 측정과 같은 환경오염 모니터링, 집안의 조명 및 창문의 원격 제어와 같은 디지털 홈, 그리고 군사 감시와 같은 실시간성이 요구되는 분야에 다양하게 적용될 수 있어야 한다. 이와 같은 고수준 서비스들은 공통적으로 센서 노드에서 수행되는 작업 처리의 실시간성을 요구한다. 그러나 TinyOS는 선입선출 (FIFO: First-In First-Out) 방식의 비선점형 태스크 스케줄링만 제공하기 때문에 즉시 실행이 필요한 태스크임에도 불구하고 자신의 우선순위보다 낮은 태스크가 획득한 CPU 사용권한을 선점할 수 없다. 따라서 TinyOS의 비실시간 스케줄링 정책은 현재 활발히 진행되고 있는 고부가가치 상용 무선 센서 네트워크의 실시간 서비스 개발에 중요한 걸림돌이 된다.

TinyOS에게 실시간 서비스를 제공하기 위한 실시간 스케줄링 연구는 공통적으로 TinyOS가 가지는 컴포넌트 기반의 구조적 특징과 호환성을 유지하면서 TinyOS의 스케줄러 컴포넌트를 보완하는데 초점을 두고 있다. 따라서 비선점 환경의 TinyOS에서 제안된 기존의 실시간 스케줄링 기법은 이벤트 기반의 센서 구동 방식을 유지하며, 스케줄러에 의한 문맥전환 (Context Switching) 오버헤드를 최소화하여 센서 노드의 초경량화를 유지하였다.

그러나 비선점 환경의 TinyOS에서 사용자 태스크 (User Task)가 요청한 실시간 서비스를 완료하기 위해서는 개별적인 사용자 태스크의 마감시한 (Deadline)을 보장함과 동시에 사용자 태스크를 완료하기 위하여 TinyOS 플랫폼에서 요구되는 태스크들의 마감시한을 동시에 보장해야 한다. 그러나 TinyOS 플랫폼 내부에서 운영되는 다수의 태스크들이 비실시간 및 독립적으로 호출 및 실행되기 때문에 요청된 사용자 태스크의 마감시한을 보장할 수 없다. 이에 본 논문에서는 이벤트 기반 비선점형 태스크 스케줄링 정책을 사용하는 TinyOS 환경에 실시간성을 제공하는 태스크 그룹 기반의 스케줄링 기법을 제안하였다. 태스크 그룹 기반의 실시간 스케줄링은 요청한 사용자 태스크와 함께 사용자 태스크의 마감시한 완료에 필요한 다수의 TinyOS 플랫폼 태스크를 한 개의 태스크 그룹으로 형성하여 그룹 내의 태스크들을 일괄적으로 스케줄링하는 효과를 준다. 제안한 기법은 비선점형 환경의 기존 TinyOS 시스템과 호환성을 유지하면서 사용자 태스크의 마감시한을 보장함과 동시에 평균 응답시간을 최소화하고자 한다. 또한, 제안한 기법은 기존 TinyOS 스케줄러 컴포넌트의 인터페이스 유지 및 확장 기능을 제공하기 때문에 이미 개발된 사용자 레벨의 응용 태스크를 수정 없이 동작할 수 있도록 지원한다. 본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 기술한다. 3장에서는 비선점 환경의 TinyOS에서 실시간성을 고려한

태스크 그룹 기반의 실시간 스케줄링 기법에 대해 기술하였다. 그리고 4장과 5장에서는 제안한 실시간 스케줄링 기법의 성능 평가를 위한 모델링 및 성능 분석을 기술하였다. 마지막으로 6장에서는 결론을 기술하였다.

2. 관련 연구

이 장에서는 TinyOS에 실시간성을 부여하기 위한 기존의 관련 연구에 대하여 기술한다. 먼저 TinyOS의 스케줄링 정책을 간략히 살펴보면 다음과 같다. 이벤트 기반 스케줄링 정책에 따라 TinyOS는 실행 중인 태스크가 없을 경우 프로세서를 수면 상태(Sleep State)로 천이하고, 인터럽트 혹은 이벤트가 발생하면 태스크를 다시 실행시켜 센서 보드의 전력 사용을 효율적으로 관리하고자 한다. 그리고 TinyOS의 비선점형 스케줄링 정책에 따라 태스크들은 서로 선점하지 않기 때문에 태스크간 문맥전환 (Context Switch)이 없다. 따라서 현재 수행되고 있는 태스크의 문맥을 저장하는 메모리와 바로 다음에 수행할 태스크의 문맥 복구에 필요한 메모리 공간이 필요 없게 되어 센서 노드의 메모리 사용량을 최소화 할 수 있다. 그림 1은 이벤트 기반 비선점형 스케줄링 기법을 사용하는 TinyOS의 정책 효과 및 목적을 보여준다.

그러나 TinyOS에서 사용하는 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법은 태스크들의 생성 순서대로 실행하기 때문에 실시간 서비스를 제공하기가 어렵다. TinyOS에 실시간성을 제공하는 기법과 관련된 연구로는 비선점형 우선순위 기반의 사용자 태스크 스케줄링 기법[6], 비선점형 EDF

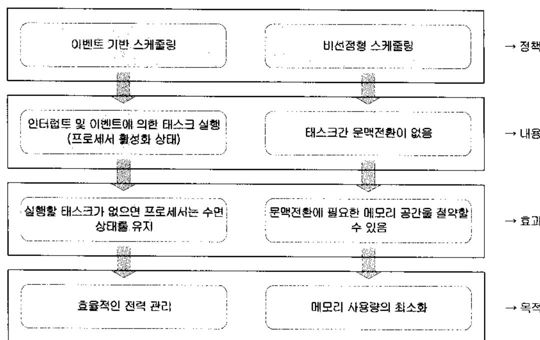

그림 1 TinyOS의 스케줄링 정책

(Earliest Deadline First) 기반의 사용자 태스크 스케줄링 기법 [7], 선점형 5-단계 우선순위 기반의 사용자 태스크 스케줄링 기법 [8] 등이 있다. 그리고 비선점 환경의 TinyOS를 선점형 및 태스크간 문맥전환을 제공할 수 있도록 수정하여 기존의 실시간 운영체제에서 사용하고 있는 선점형 EDF 사용자 태스크 스케줄링 기법을 적용시킬 수 있다. 표 1은 기존 TinyOS에 적용된 실시간 사용자 태스크 스케줄링 기법을 기술하였다. 본 논문에서 제안한 태스크 그룹 기반의 실시간 스케줄링 기법은 사용자 태스크의 마감시한 완료에 필요한 일련의 다수 태스크를 한 개의 태스크 그룹으로 형성하여 그룹 내의 태스크들을 일괄적으로 스케줄링하는 기법이다.

이벤트에 의해 비동기적으로 실행되는 이벤트 핸들러에서 호출 및 실행되는 센서 노드 태스크에게 실시간성을 제공하기 위해서는 미리 지정된 정적 우선순위를 태스크에 부여하는 것보다 마감시한 기반의 동적 우선순위를 태스크에 부여하는 것이 더욱 적합하다. 따라서 제안한 태스크 그룹 기반의 실시간 스케줄링 기법도 EDF 기반의 동적 우선순위 개념을 사용하여 사용자 태스크의 마감시한을 보장하고자 한다. 그리고 본 논문에서 제안한 실시간 스케줄링 기법의 성능을 검증하기 위하여 기존의 TinyOS에서 사용하고 있는 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법과 표 1에서 제시한 동적 우선순위 기반의 비선점형 EDF 기반 및 선점형 EDF 기반의 사용자 태스크 스케줄링 기법을 고려하였다.

사용자 태스크 스케줄링 기법들의 동작 분석에 사용되는 용어는 다음과 같다. n개의 태스크 집합 $\{T_1, T_2, \ldots, T_n\}$이 주어질 때 각 태스크 T_i의 도착시각 및 실행시간과 마감시각을 각각 R_i, C_i, 그리고 D_i라고 하면, 태스크 T_i는 $T_i:(R_i, C_i, D_i)$로 표현된다. 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법과 비선점형 및 선점형 EDF 기반의 사용자 태스크 스케

표 1. TinyOS에 적용된 실시간 사용자 태스크 스케줄링 기법

	비선점형	선점형
정적 우선순위	우선순위 기반의 사용자 태스크 스케줄링	5-단계 우선순위 기반의 사용자 태스크 스케줄링
동적 우선순위	EDF기반의 사용자 태스크 스케줄링	EDF기반의 사용자 태스크 스케줄링

줄링 기법을 평가하기 위하여 3개의 태스크 T_1:(1, 17, 41), T_2:(4, 13, 29), 그리고 T_3:(8, 9, 21)를 사용하였다. 그림 2는 3개의 태스크 T_1, T_2, 그리고 T_3를 비선점형 선입선출 및 EDF 기반, 그리고 선점형 EDF 기반의 사용자 태스크 스케줄링 기법에 적용할 때 예상되는 동작 결과를 보여준다. EDF에서는 태스크의 마감시한이 가장 촉박한 태스크에게 가장 높은 우선순위를 부여한다. 그림 2-(a)에서 보는 바와 같이 기존 TinyOS에서 사용하고 있는 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법은 사용자 태스크의 마감시한에 대한 고려 없이 태스크가 도착한 순서대로 스케줄링한다. 그림 2-(b)의 비선점형 EDF 기반 사용자 태스크 스케줄링 기법에서는 TinyOS의 제약 사항인 비선점형 태스크 스케줄링 정책 때문에 태스크 T_2는 자신보다 낮은 우선순위를 가지는 태스크 T_1의 실행시간만큼 지연 처리된다는 문제가 발생한다. 태스크 T_1의 지연 처리는 태스크 T_3에게도 영향을 주게 되어 결국 태스크 T_2와 태스크 T_3 모두 마감시한을 초과하게 된다. 그림 2-(c)의 예상 동작 결과에서 선점형 EDF 기반 사용자 태스크 스케줄링 기법은 그림 2-(a)와 그림 2-(b)에서 발생한 마감시한 초과 문제를 해결할 수 있다. 그러나 선점형 EDF 기반의 사용자 태스크 스케줄링 기법을 사용하는 경우에 문맥 전환에 필요한 메모리 사용량이 증가하게 되며, 비선점형 태스크 스케줄링 정책을 사용하는 TinyOS의 실행 환경을 선점형 환경으로 수정해야 하기 때문에 기존 TinyOS 플랫폼간의 호환성이 없어지게 된다.

그림 2에서 분석한 사용자 태스크 스케줄링 기법들의 동작 과정은 각 기법들의 개념적인 동작 과정이므로 실제 TinyOS 환경에서 동작하는 경우에는 그림 2에서 보여준 동작 결과와 매우 큰 차이가 발생한다. 이러한 차이가 발생하는 이유는, 요청한 사용자 태스크의 마감시한을 보장하기 위해 사용자 태스크뿐만 아니라 사용자 태스크에서 호출 및 실행되는 TinyOS 플랫폼 태스크들의 마감시한도 보장되어야 하기 때문이다. 그림 3은 사용자 태스크와 다수의 TinyOS 플랫폼 태스크간 논리적인 결합으로 구성된 센서 노드의 작업을 보여주고 있다. TinyOS에서 요청한 센서 노드의 작업은 최소 한 개 이상의 사용자 태스크와 최소 한 개 이상의 TinyOS 플랫폼 태스크로 구성된다. TinyOS에서 발생되는 이벤트는 주로 하드웨어 영역에서 생성된 인터럽트가 TinyOS 플랫폼 내의 이벤트 핸들러에 전달되는 신호 형식의 인터페이스이며, 이벤트 기반의 태스크 구동방식을 사용하는 TinyOS에서는 요청한 센서 노드의 작업이 이벤트에 의해서 시작된다.

비선점형 선입선출 방식의 사용자 태스크 스케줄링 및 실행 환경을 사용하는 TinyOS에서 실행 시간이 긴 사용자 태스크에 의해 나머지 사용자 태스크들의 실행이 지연되는 문제를 해결하고자 지연 처리 호출 (DPC: Deferred Procedure Call) 기법을 권고하고 있다. TinyOS의 지연 처리 호출 기법은 실행시간이 긴 태스크를 여러 개의 서브태스크로 나누어 프로세서의 선점을 양보하여 대기하고 있는 태스크들의 지연 대기 시간을 단축시키는 협업 기법이다. TinyOS에서 태스크의 실행을 지연 처리 호출하는 방식은 NesC 키워드인 post를 사용하여 task 키워드로 정의된 태스크 함수를 바로 실행시키지 않고 TinyOS의 실행 준비 큐에 먼저 등록시켜 태스크의 실행을 지연 처리한 후에 비선점형 선입선출 방식의 사용자

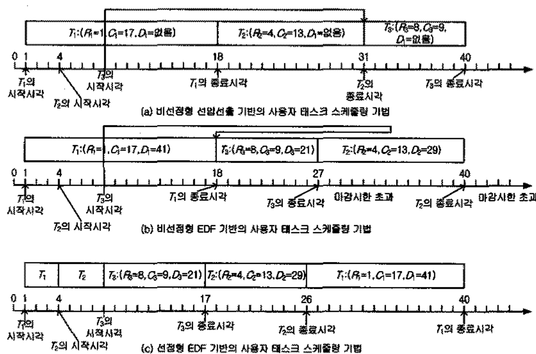

그림 2. TinyOS에서 사용자 태스크 스케줄링 기법들의 예상 동작 결과

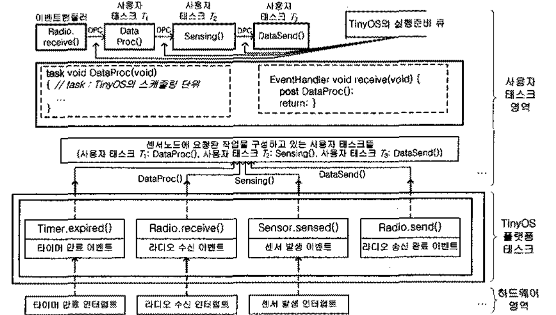

그림 3. TinyOS에서 이벤트 기반 구동 방식에 따른 센서 노드 작업의 동작 기법

태스크 스케줄링에 의하여 실행된다. 그림 3에서 센서 노드에 요청한 작업을 구성하고 있는 사용자 태스크는 3개의 사용자 태스크 (사용자 태스크 T_1, T_2, 그리고 T_3)로 나눈 후에 센서 노드의 RF (Radio Frequency) 통신 모듈에서 센서 데이터를 수신하면, 하드웨어 인터럽트가 발생한다. 하드웨어 인터럽트는 TinyOS 플랫폼에서 라디오 수신 이벤트로 전환되며 이벤트 핸들러 Radio.receive()에 의해 사용자 태스크 T_1인 DataProc()가 지연 처리 호출 및 실행된다. 사용자 태스크 T_1인 DataProc()는 수신한 데이터를 처리한 후, 사용자 태스크 T_2인 Sensing()은 센싱 기능을 수행하며, 사용자 태스크 T_3인 DataSend()는 센싱 데이터를 이웃 노드에게 전달하여 준다. 사용자 태스크 T_2와 사용자 태스크 T_3는 지연 처리 호출 방식으로 실행되는 과정을 보여준다. 그림 4는 그림 2에 기술한 사용자 태스크를 지연 처리 호출 기법으로 재구성한 사용자 태스크의 형태를 보여준다.

사용자 태스크 T_1과 T_2, 그리고 T_3은 그림 3에서 기술한 이벤트 기반 구동 방식에 의하여 재구성되었다. 각 태스크는 이벤트 핸들러에 의해서 호출되며, 이벤트 핸들러에서 소비되는 실행시간은 1로 가정하였다. 각 태스크는 지연 처리 호출 방식을 적용하여 균등하게 실행시간 4로 구성되는 서브태스크로 재구성하였다. 예를 들어, 사용자 태스크 T_1은 4개의 서브태스크 T_{1_A}, T_{1_B}, T_{1_C}, 그리고 T_{1_D}로 재구성된다. 각 서브태스크 중에서 일부는 사용자 태스크에서 호출 및 실행되는 TinyOS 플랫폼 태스크를 나타낸다. 그림 5는 서브태스크로 재구성한 사용자 태스크 T_1과 T_2, 그리고 T_3이 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법과 비선점형 및 선점형 EDF 기반의 사용자 태스크 스케줄링 기법에 따라 동작되는 예상 결과를 준다. 본 논문에서 분석한 바에 따르면, TinyOS의 스케줄링 단위는 일련의 다수 태스크로 구성된 센서 노드의 작업이 아닌, 센서 노드 작업을 구성하는 개별 태스크이기 때문에 그림 5에서 보

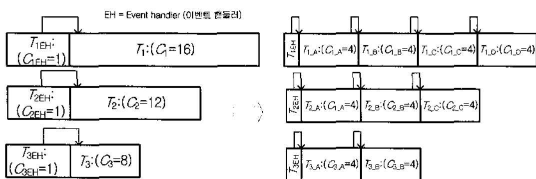

그림 4 지연 처리 호출 기법으로 재구성한 사용자 태스크

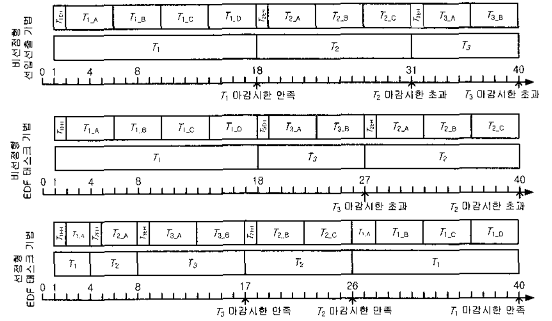

그림 5 재구성한 사용자 태스크의 예상 스케줄링 동작 결과

여준 예상 스케줄링 동작 결과와 TinyOS의 실제 동작 결과간의차이가 있다. 먼저 TinyOS의 기본 스케줄링 기법인 비선점형 선입선출 방식의 사용자 태스크 스케줄링기법을 사용한 실제 동작 결과의 분석은 그림 6과 같다. TinyOS에서는 인터럽트와 이벤트에 의해 호출되는 핸들러가 태스크보다 높은 실행 우선순위를 가진다.

그림 6에서 보는 바와 같이, 지연 처리 호출 기법을 적용한 TinyOS의 실제 동작 환경에서 다수의 서브태스크로 구성된 센서 노드 작업의 응답 시간이 예상 스케줄링의 동작 과정보다 증가된다. 지연 처리 호출 기법이 비선점형 EDF 사용자 태스크 스케줄링과 선점형 EDF 사용자 태스크 스케줄링 기법에 적용되는 경우에도 실시간으로 사용자 태스크를 스케줄

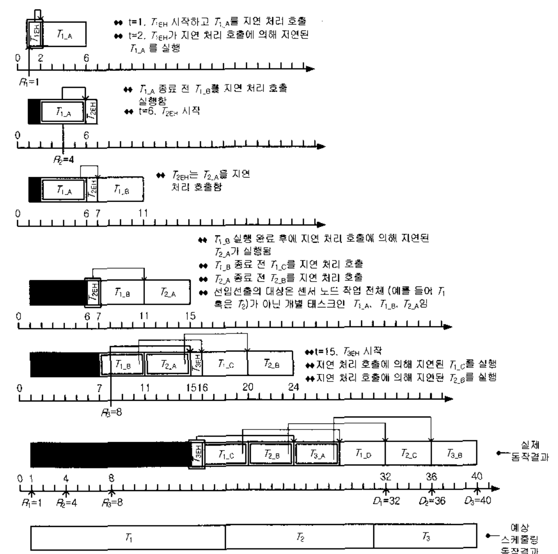

그림 6. 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법에 따른 실제 동작 결과

링하는데 문제가 발생된다. 이에 대한 이유를 살펴보면 다음과 같다.

그림 7에서 보는 바와 같이 일반적으로 TinyOS에 실시간 태스크 스케줄링을 제공하기 위하여 TinyOS의 사용자 태스크 생성 인터페이스를 수정하여 태스크의 마감시한과 같은 실시간 속성을 매개변수로 입력받아서 EDF 기반의 태스크 스케줄링을 할 수 있도록 한다. 그림 7에서 nescC 표현인 *post* T_1()은 *DPC*(*void*)로 간략하게 표현하였고, *post* T_1(*Deadline*)은 *DPC*(*Deadline*)으로 표현하였다. 이와 같이 수정된 사용자 태스크 생성 인터페이스의 매개변수를 사용하여 사용자 태스크의 마감시한을 입력할 수 있다. 그러나 TinyOS에서 센서 노드의 I/O 장치들을 제어하는 기존의 TinyOS 플랫폼 태스크들은 이미 매개변수가 없는 void형 기반의 태스크 생성 인터페이스를 사용하여 구현되어 있기 때문에 TinyOS 플랫폼 태스크의 마감시한을 설정할 수가 없다. 따라서 비선점형 EDF 기반의 사용자 태스크 스케줄링 기법과 선점형 EDF 기반의 사용자 태스크 스케줄링 기법에서는 사용자 영역에서 지연 처리 호출되는 사용자 태스크들의 마감시한만 비교 가능하다. 그리고 센서 노드의 I/O 작업을 수행하는 TinyOS 플랫폼 태스크들은 비선점형 기반의 선입선출 방식으로 스케줄링되며 사용자 태스크의 마감시한을 고려하지 않고 독립적으로 실행되기 때문에 요청한 센서 노드 작업의 실시간 속성이 상실되는 현상이 발생한다.

센서 노드 작업의 실시간 속성이 상실되는 현상이 나머지 실시간 스케줄링 기법인 비선점형 EDF 및 선점형 EDF 기반의 사용자 태스크 스케줄링에 미치는 영향을 분석한 결과는 그림 8과 같다. 그림 8에서 공통적으로 마감시한이 가장 짧은 태스크 T_3는 지연 처리 호출 과정을 거치며 최상위 우선순위를 상실하게 되어 예상 스케줄링 동작 결과와는 다르게 마감시

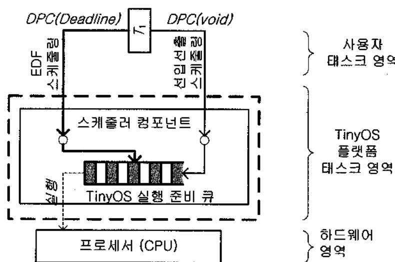

그림 7. 태스크 생성 인터페이스

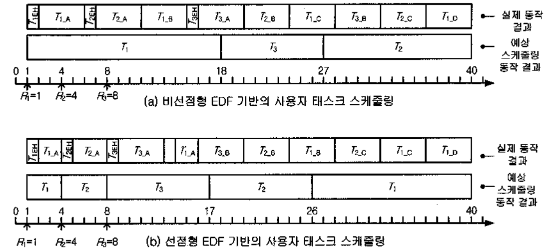

그림 8. 비선점형 및 선점형 EDF 사용자 스케줄링 기법의 실제 동작 결과

한을 초과하게 된다. 비선점형 EDF 및 선점형 EDF 기반의 사용자 태스크 스케줄링 기법을 사용하더라도 사용자 태스크를 제외한 TinyOS 플랫폼 태스크의 순차적 호출 및 실행으로 인하여 실시간 속성의 상실 현상이 발생하게 되어 요청한 센서 노드 작업의 마감시한이 초과하게 된다.

다음 장에서는 사용자 태스크 및 TinyOS 플랫폼 태스크들이 지연 처리 호출되는 과정에서 발생되는 실시간 속성의 상실 현상 문제를 해결하는 태스크 그룹 기반의 스케줄링 기법에 대하여 기술하였다.

3. 태스크 그룹 기반의 스케줄링 기법

실시간 속성의 상실 현상이 발생되는 원인은 이벤트 핸들러에 의해 지연 처리 호출되는 사용자 태스크만이 마감시한 정보를 입력받기 때문이다. 따라서 TinyOS의 태스크 생성 인터페이스를 수정하여 사용자 태스크뿐만 아니라 TinyOS 플랫폼 태스크가 지연 처리 호출되는 경우에도 마감시한 정보를 입력받을 수 있게 해야 한다. 그러나 TinyOS에서 태스크 생성 인터페이스의 수정 없이는 TinyOS 플랫폼 태스크에게 마감시한을 부여하는 것이 불가능하다. 그러나 TinyOS 플랫폼에서 사용되는 태스크들은 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법으로 호출 및 실행되기 때문에 마감시한 정보를 입력받지 않는 태스크 생성인터페이스로 이미 구현되어 있다. 따라서 실시간 속성의 상실 현상을 해결하기 위해서는 TinyOS 플랫폼의 태스크 생성 인터페이스를 수정함과 동시에 선점형 기반의 플랫폼 환경으로 수정해야 한다. 그러나 이러한 수정 요구는 모든 TinyOS 플랫폼 태스크의 재작성과 그림 1에서 보여준 TinyOS의 주요 정책 효과 및 목적이 없어지게 된다. 요청된 센서 노드 작업을 마감시한 내에 완료

하기 위하여 사용자 태스크 및 TinyOS 플랫폼 태스크들의 마감시한을 동적으로 재계산하는 규칙을 결정하는 것은 해결하기 어려운 문제이다. 이에 본 논문에서는 TinyOS 플랫폼 태스크들을 재작성하는 비용 없이 센서 노드 작업을 구성하는 태스크들의 지연처리 호출하는 방식을 그대로 사용하면서, 동시에 사용자 태스크와 사용자 태스크에 의존적인 TinyOS 플랫폼 태스크에게 사용자 태스크의 실시간 속성을 상속하고 태스크 그룹으로 결합시키는, 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법을 제안하였다. 먼저 비선점형 TinyOS 환경에서 마감시한을 가지는 센서 노드 작업의 태스크 구성도는 그림 9와 같다. 그림 9는 이벤트 핸들러 T_{1EH}에서 호출되는 태스크 T_1의 실행 완료에 필요한 그러나 지연처리 호출되는 TinyOS 플랫폼 태스크들의 구성도이며, 이 전체가 1개의 센서 노드 작업이 된다.

그림 10은 그림 9의 태스크 구성도에서 보여준 태스크 $T_{I/O}$가 센서 데이터를 수신하여 발생시킨 라디오 수신 인터럽트 및 이벤트에 의해 실행되는 이벤트 핸들러 T_{1EH}로부터 실행되는 태스크 T_1의 동작 과정을 보여준다. 그림 10에서 이벤트 핸들러 T_{1EH}가 사용자 태스크 T_1을 지연 처리 호출할 때, 태스크 스케줄러는 마감시한을 매개변수로 입력받을 수 있도록 수정한 사용자 태스크 생성 인터페이스를 통해 센서 노드 작업의 마감시한 정보를 입력 받는다. 그리고 요청한 센서 노드 작업을 구성하는 사용자 태스크와 기존의 지연 처리 호출 인터페이스를 통해 생성되는

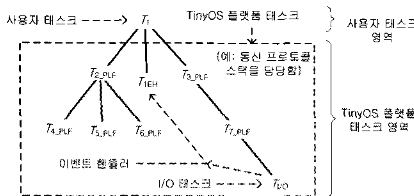

그림 9. 사용자 태스크의 실행 완료에 필요한 TinyOS 플랫폼 태스크들의 구성도

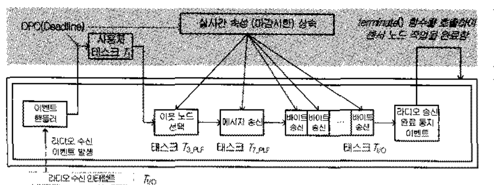

그림 10 센서 노드 작업의 동적 실행 과정

TinyOS 플랫폼 태스크들은 하나의 태스크 그룹으로 형성되고, 태스크 그룹은 사용자 태스크로부터 입력된 마감시한 기반의 실시간 속성을 상속받는다. 마감시한 기반의 실시간 속성을 상속하는 것은 센서 노드 작업이 요청한 라디오 송신이 완료된 후 라디오 송신 완료 통지 이벤트가 태스크 $T_{I/O}$로부터 발생할 때까지 유지된다. 그리고 마지막으로 사용자 태스크 T_1은 terminate() 함수를 호출하여 요청한 센서 노드 작업을 완료한다. 기존 TinyOS의 모든 I/O 플랫폼 태스크들은 완료 통지 이벤트를 발생시킨다. 이와 같이 태스크 그룹의 형성 및 태스크 그룹에게 실시간 속성을 상속하는 기법을 적용하여 비선점형 TinyOS에서 센서 노드 작업 단위의 실시간 스케줄링을 제공할 수 있도록 하였다. TinyOS에서 제공하고 있는 I/O 플랫폼 태스크들에 대한 명령 형식과 신호 형식에 대한 예는 표 2와 같다.

본 논문에서 제안한 태스크 그룹 기반 비선점형 EDF 사용자 태스크 스케줄링 기법의 동작 흐름은 표 3과 같다. 그림 4에서 기술한 태스크 T_1, T_2, 그리고 T_3의 스케줄링을 제안한 태스크 그룹기반의 비선점형 EDF 사용자 태스크 스케줄링 기법에 적용한 동작 결과는 그림 11과 같다. 실시간 속성인 마감시한을 가지는 사용자 태스크 T_1의 서브태스크 T_{1_A}가 이벤트 핸들러 T_{1EH}에 의해 지연 처리 호출되면 사용자 태스크 스케줄러는 해당 센서 노드 작업을 식별하기 위한 태스크 T_1의 식별자와 마감시한을 리스트 자료 구조로 구현된 WorkList에 삽입한다 [표 3의 (I)과 그림 11의 동작 과정 (a)]. 삽입 방식은 센서 노드 작업의 구성 요소인 사용자 태스크의 마감시한이 촉박한 순으로 삽입하게 된다 [그림 11의 동작 과정 (d)와 (e)].

표 2. 완료 통지 이벤트를 발생시키는 TinyOS의 I/O 플랫폼 태스크

I/O 플랫폼 태스크	명령(command) 형식	신호(signal) 형식
RF 기반의 라디오 송신	call Radio.send()	event Radio.sendDone()
센서 감지	call Sensor.sense()	event Sensor.senseDone()
시리얼 출력	call Serial.flush()	event Serial.flushDone()
ROM 기록	call ROM.write()	event ROM.writeDone()

표 3. 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법

[I] WorkList: 이벤트 핸들러에 의해 지연 처리 호출되는 사용자 태스크를 저장한 큐이며, 사용자 태스크의 마감시한을 기준으로 오름차순으로 태스크들을 정렬한다. WorkList는 사용자 태스크의 식별자와 마감시한을 구성되는 리스트 자료 구조이다.
[II] TaskList: 개별 사용자 태스크가 실행 완료하는데 요구되는 TinyOS 플랫폼 태스크들이 저장되는 리스크 자료 구조이다. 개별 사용자 태스크와 해당 TinyOS 플랫폼 태스크들은 하나의 태스크 그룹으로 형성되고, 형성된 태스크 그룹은 사용자 태스크로부터 입력된 마감 시한 기반의 실시간 속성을 상속 받는다.
[III] HoldTask포인터: PreemptStack 구조에서 사용하는 HoldTask 포인터는 가장 최근에 선점당한 태스크의 식별자를 가리킨다.
[IV] PreemptStack: 우선순위가 높은 (혹은 마감시한이 짧은) 태스크에 의해 선점당한 태스크의 마감시한 정보와 식별자를 저장하는 스택 구조이다.

```
Task_Group_Based_Scheduler()
{
    foreach entry task in WorkList
        if (WorkList and TaskList are empty) then Switch
Processor_State to Sleep_Mode;
        else
            Previous_Task ← Current_Task
            Current_Task ← Select a task from TaskList;
 [V]            if (Previous_Task ≠Current_Task) then
HoldTaskPointer ← Previous_Task;
            Run (Current_Task);
        endif
    endforeach
}
```

WorkList에 삽입된 정보는 요청한 센서 노드 작업에 대한 완료 통지 이벤트를 전달받을 때까지 계속 유지된다. 사용자 태스크 T_1이 다른 태스크에 의해 선점되지 않는다면, 마감시한을 입력 받을 수 없는 기존의 지연 처리 호출 인터페이스를 통해 지연 처리 호출 및 실행되는 서브태스크들(예를 들어, T_{1_A}, T_{1_B}, T_{1_C}, 그리고 T_{1_D})은 호출된 순서대로 리스트 자료 구조로 구현된 TaskList에 저장 및 관리된다 [표 3의 (II)와 그림 11의 동작 과정 (b)]. 우선순위가 높은 태스크에 의해 선점당한 태스크는 TaskList 자료구조에 저장된다. 그리고 선점당한 태스크의 식별자는 HoldTask 포인터로 관리된다 [표 3의 (III)과 그림 11의 동작 과정 (d)]. 마지막으로 우선순위가 높은 센서 노드 작업에 의해 선점당한 태스크들은 스택으로 구현된 PreemptStk에 저장한다 [표 3의

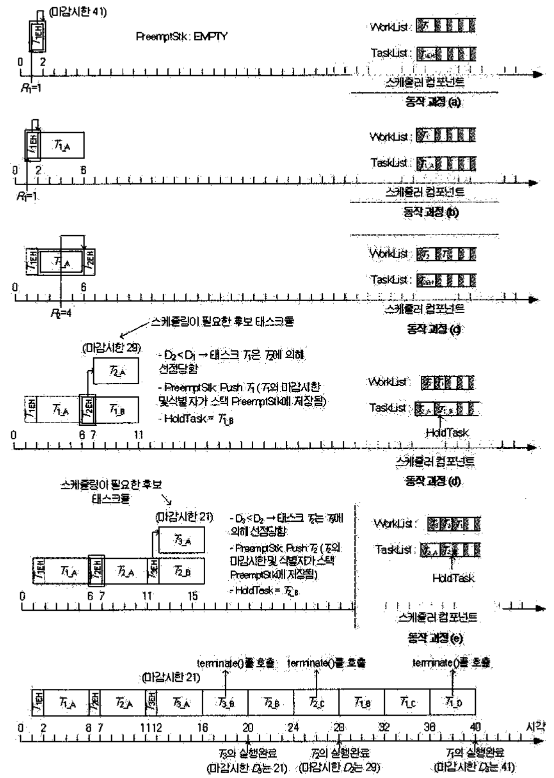

그림 11. 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법

(IV)와 그림 11의 동작 과정 (d)와 (e)].

지금까지 언급한 WorkList와 TaskList, 그리고 PreempStk 구조는 TinyOS의 스케줄러 컴포넌트에 구현되었다. PreemptStack은 선점된 해당 태스크의 실시간 속성인 마감시한을 저장하기 위한 1바이트와 태스크 식별자를 저장하기 위한 1바이트로 구성된다. 선점당한 태스크의 속성 정보들이 저장되는 PreemptStk이 스택으로 구현되어 있기 때문에 마감시한의 값이 짧은 태스크의 속성 정보가 PreemptStk의 맨 상위에 위치한다. 따라서 새로 호출한 태스크를 실행할지 아니면 이전에 호출한 태스크를 먼저 실행할지를 결정하기 위해서는 WorkList의 맨 앞에 저장한 태스크의 마감시한과 PreemptStk의 맨 상위에 저장된 태스크의 실시간 속성인 마감시한을 비교하여 결정한다 [그림 11의 동작 과정 (d)와 (e)]. 그리고 선점당한 태스크는 TaskList에서 사용하는 HoldTask 포인터로 관리된다. HoldTask 포인터가 가리키는 태스크보다 마감시한이 짧은 사용자 태스크로부터 지연 처리 호출되는 서브태스크들이 계속

도착하면, HoldTask 포인터 앞으로 전방삽입이 되고 이를 통해 해당 서브태스크들을 동일한 태스크 그룹으로 형성할 수 있다 [그림 11의 동작 과정 (d)와 (e)]. 그림 11의 동작 과정을 좀 더 자세히 살펴보면 다음과 같다. 먼저 시각 1과 2에서 태스크 T_1의 서브태스크 T_{1EH}에 의해 지연 처리 호출되는 서브태스크 T_{1_A}가 실행된다. 이때 TaskList에 현재 실행 중인 서브태스크 T_{1_A}의 태스크 식별자를 저장한다 [표 3의 (I), 그림 11의 동작 과정 (a)와 (b)]. 시각 7에서 태스크 T_2의 서브태스크 T_{2_A}가 도착하면 태스크 T_2로부터 상속받은 서브태스크 T_{2_A}의 마감시한이 서브태스크 T_{1_A}에 의해 지연 처리 호출되는 서브태스크 T_{1_B}의 마감시한보다 짧기 때문에 서브태스크 T_{2_A}의 선점이 발생한다 [그림 11의 동작 과정 (c)]. 선점당한 서브태스크 T_{1_B}는 HoldTask 포인터에 의해 관리되며 사용자 태스크 T_1의 식별자와 마감시한이 PreemptStk에 저장된다 [표 3의 (V)]. 선점이 발생된 후부터 생성되는 태스크들이 HoldTask 포인터가 가리키는 태스크의 마감시한보다 짧으면, HoldTask 포인터 앞으로 전방 삽입이 된다 [표 3의 (III), 그림 11의 동작 과정 (d)와 (e)]. 이러한 전방 삽입은 결과적으로 센서 노드 작업을 구성하는 사용자 태스크와 사용자 태스크에 의해 지연 처리 호출되는 TinyOS 플랫폼 태스크들을 논리적으로 결합시켜 태스크 그룹으로 형성하는 효과가 있다. 그리고 요청한 센서 노드 작업에 대한 완료 통지 이벤트를 수신하면 사용자 태스크는 terminate()를 호출하여 WorkList로부터 자신의 정보를 삭제한 후에 선점작업을 완료한다. 그림 11에서 분석한 바와 같이, 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법은 마감시한을 매개변수로 가질 수 없는 TinyOS 플랫폼 태스크들에 대하여 요청한 사용자 태스크의 실시간 속성인 마감시한을 상속시키는 방안을 적용시켜 센서 노드 작업의 실시간 속성을 만족시킨다.

4. 성능 평가 모델

태스크 그룹 기반 비선점형 EDF 사용자 태스크 스케줄링 기법의 성능 평가를 위하여 큐잉 이론 기반의 $M/E_k/1$ 모델을 사용하였다. M은 TinyOS에서 비동기적으로 발생하는 이벤트에 의해 실행되는 태스크의 생성 간격이 비기억 (Memoryless) 분포를 따르는 것을 의미한다. E_k는 하나의 센서 노드 작업을 처리하기 위해 형성된 태스크 그룹이 얼랑 (Erlang) 분포에 따라 모형 매개변수 (Shape Parameter) k개의 태스크로 이루어진 것을 의미한다. 1은 태스크를 처리하는 프로세서의 수가 한 개임을 나타낸다. TinyOS에서 생성되는 이벤트의 도착간격이 서로 독립적이라고 가정하면, TinyOS의 실행 준비 큐에 도착하는 센서 노드의 작업은 포아송 (Poisson) 분포를 따른다. 따라서 센서 노드 작업의 시작간격 X는 수식 (1)과 같은 지수 랜덤 (Exponential Random) 변수로 정의할 수 있다. 이 때, 도착율 (Arrival rate) λ는 단위 시간당 센서 노드 작업이 발생하는 횟수의 기대값 (Expected Value)을 의미한다.

$$X \sim \mathrm{Exp}onential\left(mean : \frac{1}{\lambda}\right) \tag{1}$$

그리고 한 개의 센서 노드 작업은 지연 처리 호출되는 다수 태스크들로 구성된다. 따라서 센서 노드 작업의 실행시간 Y는 각 태스크의 실행 시간 Y_i의 총합과 같다. Y_i들이 지수 분포를 따르고 서로 독립적인 랜덤 변수라면, 하나의 센서 노드 작업이 프로세서에 의해 처리되는 시간 Y는 수식 (2)와 같은 얼랑-k (Erlang-k) 랜덤 변수로 표현가능하다. 이 때, 처리율 (Service Rate) μ는 단위 시간당 프로세서가 처리하는 센서 노드 작업 개수의 기대값을 의미한다.

$$Y \sim ErLang_k\left(mean : \frac{1}{\mu}\right),\ k : \begin{cases} Y = Y_1 + Y_2 + \ldots + Y_k = \sum_{i=1}^{k} Y_i \\ Y_i \sim \mathrm{Exp}onential\left(mean : \frac{1}{\mu}/k\right) \end{cases} \tag{2}$$

얼랑-k 분포에서 모형 매개변수 k는 하나의 센서 노드 작업을 구성하는 태스크의 개수이다. 그리고 일반적인 스케줄링 상황을 시뮬레이션하기 위하여 태스크의 개수 역시 랜덤 변수로 고려하였다. 수식 (3)에 정의한 태스크의 랜덤 변수 K는 이산적 (Discrete)이면서 지수 랜덤 변수와 같이 한계 범위를 가지지 않는 특징이 있으며 K를 기하 (Geometric) 랜덤 변수로 설정하였다.

$$K \sim Geometric(mean : \bar{k}) \tag{3}$$

제안한 기법의 성능 평가 모델에서 실시간 속성인 마감시한은 센서 노드 작업의 우선순위를 동적으로 정하는 기준이 되며, 스케줄링 기법의 성능 평가에 필요한 중요한 인자가 된다. 그러나 기존의 $M/E_k/1$ 큐잉 모델은 기본적으로 선입선출 방식을 사용하기

때문에 센서 노드 작업의 마감시한에 대한 랜덤 변수를 고려하지 않는다. 따라서 본 논문에서는 마감시한을 고려하는 $M/E_k/1$ 스케줄링 큐잉 모델을 유도하였다. 먼저 마감시각의 랜덤 변수를 정의하였다. 다수의 태스크로 구성되는 센서 노드 작업에 대한 마감시각의 최대값은 동일한 작업의 다음 시작시각이다. 그리고 마감시각과 시작시각간의 차이로 표현된 마감시한은 센서 노드 작업의 실행시간보다는 큰 값을 가진다고 가정하였다. 먼저 수식 (4)에서 마감시한을 나타내는 랜덤 변수 Z와 실행시간 Y를 사용하여 수식 (5)에서 유도된 결합 확률밀도함수 (Joint PDF: Probability Density Function)를 정의한다.

$$\begin{aligned} f_{Y,Z}(y,z)dydz &= Prob\{y \le Y \le y+dy, z \le Z \le z+dz\} \\ &= Prob\{y \le Y \le y+dy\} \cdot Prob\{z \le Z \le z+dz\} \\ &= f_Y(y)dy \cdot f_{X|\{X>y\}}(z)dz = f_Y(y) \cdot \frac{f_x(z)}{Prob\{X>y\}}dydz \end{aligned} \tag{4}$$

$$\therefore f_{Y,Z}(y,z) = \underbrace{\frac{(k\mu)^k}{(k-1)!} \cdot y^{k-1} \cdot e^{-k\mu y}}_{f_Y(y)} \cdot \underbrace{\frac{\lambda \cdot e^{-\lambda z}}{e^{-\lambda y}}}_{\frac{f_X(z)}{Prob\{X>y\}}} \cdot u(z-y),\ 0 \le y \le z \tag{5}$$

마감시간만을 고려한 랜덤 분포를 얻기 위해 수식 (5)로부터 마감시한 Z의 주변 확률밀도함수(Marginal PDF)를 구한다. 주변 확률밀도함수의 정의로부터 유도한 결과를 수식 (6)과 수식 (7)에 기술하였다.

$$f_Z(z) = \int_{-\infty}^{\infty} f_{Y,Z}(y,z)dy = \lambda \cdot e^{-\lambda z} \cdot \frac{(k\mu)^k}{(k-1)!} \cdot \int_0^z y^{k-1} \cdot e^{-y(k\mu-\lambda)}dy \tag{6}$$

$$\therefore f_z(z) = \frac{\gamma(k,(k\mu-\lambda)z)}{(k-1)!} \cdot \left(\frac{k\mu}{k\mu-\lambda}\right)^k \cdot \lambda \cdot e^{-\lambda z}, \tag{7}$$
$$where\ \gamma(\alpha,\beta) = \int_0^\beta t^{\alpha-1} \cdot e^t dt$$

마감시한을 표현하는 랜덤변수 Z의 확률밀도함수를 사용하여 Z의 기대값 $E[Z]$를 수식 (8)과 같이 유도할 수 있다.

$$\begin{aligned} E[Z] &= \int_0^\infty z \cdot f_z(z)dz = \int_0^\infty z \cdot \frac{\gamma(k,(k\mu-\lambda)z)}{(k-1)!} \cdot \left(\frac{k\mu}{k\mu-\lambda}\right)^k \cdot \lambda \cdot e^{-\lambda z}dz \\ &= \frac{\lambda}{a^k} \cdot \int_0^\infty z \cdot e^{-\lambda z} \cdot \underbrace{\left(1 - e^{-(k\mu-\lambda)z}\sum_{i=0}^{k-1}\frac{((k\mu-\lambda)z)^i}{i!}\right)}_{\frac{\gamma(k,(k\mu-\lambda)\cdot z)}{(k-1)!}}dz, \\ &\qquad where\ a = \frac{k\mu-\lambda}{k\mu} \\ &= \frac{\lambda}{a^k} \cdot \left(\int_0^\infty z \cdot e^{-\lambda z}dz - \int_0^\infty \sum_{i=0}^{k-1}\left(\frac{(k\mu-\lambda)^i}{i!} \cdot z^{i+1} \cdot e^{-k\mu z}\right)dz\right) \\ &= \frac{\lambda}{a^k} \cdot \left(\frac{1}{\lambda^2} \cdot \int_0^\infty t^1 \cdot e^{-t}dt - \sum_{i=0}^{k-1}\left(\frac{(k\mu-\lambda)^i}{i!} \cdot \left(\int_0^\infty z^{i+1} \cdot e^{-k\mu z}dz\right)\right)\right) \\ &= \frac{\lambda}{a^k} \cdot \left(\frac{1}{\lambda^2} - \frac{1}{(k\mu)^2}\sum_{i=0}^{k-1}\left(a^i \cdot (i+1)\right)\right) \\ &= \frac{\lambda}{a^k} \cdot \left(\frac{a^k}{\lambda} \cdot \left(\frac{1}{\lambda} + \frac{1}{\mu}\right)\right) = \frac{1}{\lambda} + \frac{1}{\mu} \end{aligned} \tag{8}$$

수식 (8)의 결과를 분석하여 보면, 마감시한 Z의 기대값은 도착간격 X의 기대값과 실행시간 Y의 기대값의 합과 같다는 것을 보여준다. 수식 (8)에서 프로세서의 처리율 μ가 고정되어 있는 경우, 도착율 λ가 증가하면 단축된 센서 노드의 작업 시작 간격을 반영하기 위하여 센서 노드 작업의 평균 마감시간이 축소되도록 마감시한 분포를 생성한다. 그리고 프로세서의 처리율 μ가 감소하면 느려진 센서 노드의 작업 실행시간을 고려하여 센서 노드 작업들의 마감시간이 증가하도록 마감시한 분포를 생성한다. 즉, 수식 (8)에서 유도한 센서 노드 작업의 마감시한 랜덤 변수 Z는 처리기 이용률 (Processor Utilization)인 $\rho = \lambda/\mu$에 따른 센서 노드 작업의 마감시한 분포를 적절하게 생성되고 있음을 확인할 수 있다. 마감시한 Z의 또 다른 속성을 파악하기 위하여 주어진 실행시간 Y=y에 대한 마감시한 Z의 조건부 누적분포함수 (Conditional CDF : Cumulative Distribution Function)를 유도하였다. 수식 (9)에서는 y보다 항상 큰 값을 가지는 Z를 고려하여 유도한 Z의 조건부 누적분포함수의 결과를 보여준다.

$$\begin{aligned} F_{Z|Y}(y+s|y) &= \frac{\int_{-\infty}^{y+s} f_{Y,Z}(y,z)dz}{f_Y(y)} = \frac{\int_y^{y+s} f_Y(y) \cdot \lambda \cdot e^{-\lambda(z-y)}dz}{f_Y(y)} \\ &= \lambda \cdot e^{\lambda y} \cdot \int_y^{y+s} e^{-\lambda z}dz = -e^{\lambda y} \cdot \left(e^{-\lambda z}\right)\Big|_y^{y+s} \\ &= -e^{\lambda y} \cdot \left(e^{-\lambda y - \lambda s} - e^{-\lambda y}\right) = 1 - e^{-\lambda s} \end{aligned}$$

$$F_{Z|Y}(z|y) = Prob\{Z \le z | Y = y\} = \frac{\int_{-\infty}^{z} f_{Y,Z}(y,z')dz'}{f_Y(y)} \tag{9}$$

수식 (9)에서는 z를 $y+s$로 치환하여 0보다 큰 s에 대해 지수분포와 동일한 누적분포함수를 얻는다. 여유시간에 대한 랜덤 변수 S를 마감시간 Z와 실행시간 Y의 차 ($S = Z - Y$)로 정의하여 수식 (10)에서 유도한 여유시간 S에 대한 확률 분포를 얻을 수 있다. 여유시간이란 센서 노드의 작업을 실행 완료하는 시점으로부터 남은 마감시한을 의미한다.

$$Prob\{Z > y+s\} = Prob\{Z - y > s\} = Prob\{S > s\} = e^{-\lambda s}$$

$$\begin{aligned} Prob\{S > s+t | S > t\} &= \frac{Prob\{S > s+t, S > t\}}{Prob\{S > t\}} \\ &= \frac{Prob\{S > s+t\}}{Prob\{S > t\}} \\ &= \frac{e^{-\lambda(s+t)}}{e^{-\lambda t}} = e^{-\lambda s} \\ &= Prob\{S > s\} \end{aligned} \tag{10}$$

하나의 센서 노드 작업이 수행중이고 스케줄링 큐에 대기하고 있는 센서 노드 작업이 없는 상황에는 마감시한 Z의 분포를 유도하는 과정에서 언급한 바와 같이 마감시각은 항상 종료시각보다 늦은 시각이므로 수행 중인 센서 노드 작업은 마감시한을 반드시

만족시킨다. 만약 마감시한의 분포가 고정 크기의 여유시간으로 생성되는 경우, 실행시간이 짧은 센서 노드 작업일수록 가까운 마감시한을 가지게 되어 EDF 기반의 스케줄링 기법은 마치 SJF(Shortest Job First) 방식과 유사하게 동작된다. 지금까지 설명한 마감시한 생성 분포 수식들은 제안한 기법의 실험 및 성능 평가에 적용하였다.

5. 실험 및 성능 평가

앞서 2장과 3장에서 살펴본 스케줄링 기법을 현재 많이 사용되고 있는 센서 노드의 하드웨어 플랫폼인 Chipcon사의 CC2420DB 무선 센서 모트(Mote) [9]에 동작하는 TinyOS 환경에서 실험한 결과, 그림 2부터 그림 11에서 보여준 동작 결과와 동일함을 확인하였다. 그러나 TinyOS에서 운영 가능한 다양한 실시간 스케줄링 기법들의 성능을 평가하기 위하여 4장에서 기술한 $M/E_k/1$ 큐잉 모델을 기반으로 한 시뮬레이션 환경을 TinyOS 환경에서 설계 및 구현하였다. 구현된 시뮬레이션을 기반으로 하여 TinyOS 플랫폼 환경에서 운영 가능한 실시간 사용자 태스크 스케줄링 기법들의 성능을 비교 분석하였다. 성능 분석 대상인 스케줄링 기법은 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법, 비선점형 EDF 사용자 태스크 스케줄링 기법, 선점형 EDF 사용자 태스크 스케줄링 기법, 그리고 본 논문에서 제안한 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법이다. 성능 평가 요소로는 센서 노드 작업의 마감시한 만족율과 평균 대기율을 사용하였다. 그리고 TinyOS에서 기본적으로 제공하는 태스크 대기 큐의 크기가 7이므로 성능 분석을 위하여 하나의 루프 (Loop)를 가지면서 마감시한을 갖는 7개의 더미(Dummy) 사용자 태스크를 생성하고 실행하였다. 생성되는 개별 더미 태스크의 μ와 λ는 1에서 10까지 동적으로 조절이 가능하도록 하였다. 제안한 기법의 성능 평가에서 처리기 이용률에 따른 사용자 태스크의 마감시한 만족율과 평균 대기율을 측정하였다. 처리기 이용률의 증가에 대한 변화를 주기 위해서는 실행하고자 하는 태스크의 수를 임의로 증가하거나 태스크의 생성 간격 λ와 처리율 μ를 조절할 수 있다. 본 논문에서는 후자의 방법을 사용하여 이미 잘 알려져 있는 큐잉 모델에서 사용되는 성능 인자들을 활용하였다. 또한 센서 네트워크에서 이벤트가 매우 빈번하게 발생되는 상황과 이러한 상황으로 인하여 해당 이벤트를 처리하는 태스크의 생성 간격이 짧아져서 센서 노드 플랫폼에 과부하가 발생되는 지점인 처리기 이용률 1에 대한 실험도 수행하였다. 설정된 μ와 λ를 기반으로 하여 사용자 태스크와 사용자 태스크에 의해 지연 처리 호출되는 TinyOS 플랫폼 태스크의 실행 정보는 다음과 같다. 먼저, 각 태스크의 생성 간격은 수식 (1)의 랜덤 변수 X에 따른 분포로 생성된다. 각 태스크의 실행 시간은 수식 (2)의 랜덤 변수 Y에 따른 분포로 생성된다. 센서 노드의 마감시한은 수식 (7)에 따라 생성된다. 그리고 TinyOS 플랫폼 태스크의 생성 개수는 수식 (3)의 랜덤 변수 Z에 따른 분포로 생성된다. 마감시한 만족율은 스케줄링된 총 센서 노드 작업에 대해 마감시한을 만족시킨 센서 노드 작업의 비로 계산한다.

그림 12는 처리기 이용률을 나타내는 $\rho = \lambda/\mu$에 따른 마감시각 만족율을 보여준다. 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법은 성능 분석에 대한 최저 성능 경계 (Minimum Lower Bound)가 된다.

그림 12에서 보는 바와 같이 기존의 비선점형 TinyOS 환경에서 비선점형 EDF 및 선점형 EDF 사용자 태스크 스케줄링 기법은 TinyOS의 실시간 스케줄링을 크게 개선하지 못하고 있음을 알 수 있다. 그러나 본 논문에서 제안한 태스크 그룹 기반의 비선점형 EDF 태스크 스케줄링 기법은 나머지 스케줄링 기법들보다 향상된 성능을 보여준다.

그림 13은 처리기 이용률 (Processor Utilization)에 따른 평균 대기율을 보여준다. 센서 노드 작업의 대기시간은 해당 작업이 스케줄링 큐에서 프로세서

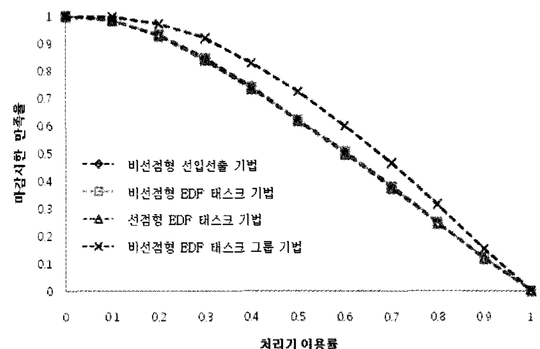

그림 12 스케줄링 기법에 따른 마감시한 만족율

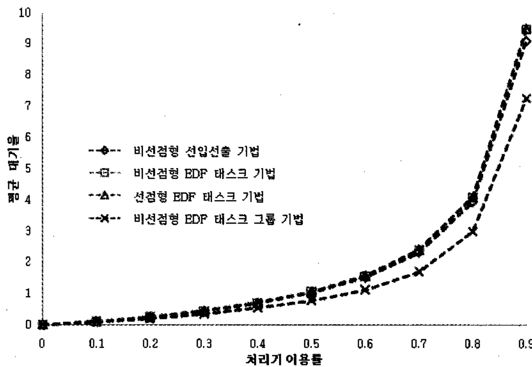

그림 13. 스케줄링 기법에 따른 평균 대기율

의 할당을 기다린 시간이다. 대기율은 센서 노드 작업의 실행시간에 대한 대기시간의 비를 의미한다. 그리고 평균 대기율은 스케줄링된 모든 센서 노드 작업에 대한 대기율의 평균으로 정의한다. 그림 13의 결과를 살펴보면, 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법이 다른 스케줄링 기법들보다 우수한 성능을 보여 주고 있다. 비선점형 EDF 및 선점형 EDF 태스크 스케줄링 기법이 센서 노드 작업의 대기시간을 개선시키지 못하는 이유는 비선점 환경의 TinyOS에서 실시간 속성의 상실 현상이 발생하여 센서 노드 작업을 구성하는 TinyOS 플랫폼 태스크들이 라운드-로빈(Round-Robin) 형태로 스케줄링되기 때문이다.

스케줄링 기법들의 복잡도(Complexity) 평가에 대한 분석을 살펴보면 다음과 같다. 먼저 스케줄링 기법들의 시간 복잡도 (Time Complexity)는 표 4와 같다. 표 4에서 $N(T)$는 사용자 태스크의 개수를 나타낸다. TinyOS에서 실행되는 실시간 스케줄링 기법들은 공통적으로 도착한 새로운 센서 노드 작업의 실시간 속성인 마감시한과 스케줄링 큐에서 대기하고 있는 센서 노드 작업들의 마감시한을 선형(Linear)으로 비교하는 과정을 거치므로 사용자 태스크의 삽입 비용은 모두 $O(N(T))$의 시간이 된다. 선점형 EDF 태스크 기법에서는 문맥 전환 오버헤드 α가 추가되며 비선점형 EDF 태스크 그룹 기법에서는 태스크 선점에 사용되는 스택 PreemptStk을 관리하는데 필요한 오버헤드 β가 추가된다. 문맥 전환에 필요한 시간 비용 α는 프로세서의 모든 레지스터를 저장 및 복원하는 명령의 총합과 동일하기 때문에 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법에서 소요되는 스택 포인터의 증감 비용 β보다 큰 값을 가진다.

표 4. 스케줄링 기법의 계산 복잡도

	사용자 태스크인 경우		플랫폼 태스크인 경우	
	삽입	인출	삽입	인출
비선점형 선입선출	-	-	O(1)	O(1)
비선점형 EDF	$O(N(T))$	O(1)	O(1)	O(1)
선점형 EDF	$O(N(T))+\alpha$	$O(1)+\alpha$	O(1)	O(1)
비선점형 EDF 태스크 그룹	$O(N(T))+\beta$	$O(1)+\beta$	O(1)	O(1)

표 5는 각 스케줄링 기법의 공간 복잡도(Space Complexity)를 평가하기 위해 측정된 메모리 (SRAM)의 사용량을 보여준다.

표 5에서 $N(T_{PLF})$는 TinyOS 플랫폼 태스크의 개수를 나타내며, #Reg는 프로세서에서 사용되는 모든 레지스터의 개수를 나타낸다. 태스크 그룹 기반의 비선점형 EDF 태스크 스케줄링 기법은 양방향 리스트를 유지하면서 선점당한 태스크에 대해서는 1바이트 크기의 태스크 식별자와 1바이트 크기의 마감시한을 스택 PreemptStk에 저장하기 때문에 비선점형 선입선출 방식의 사용자 태스크 스케줄링 기법보다 $2*N(T)$ 만큼의 추가적인 메모리가 요구한다. 그러나 TinyOS에서 사용자 태스크의 실시간 속성을 만족하기 위해 요구되는 태스크의 식별자와 도착시각과 실행시간, 그리고 마감시한 정보를 저장하기 위한 $4*N(T)$ 만큼의 추가적인 메모리 사용에 비하면, 선점당한 태스크를 관리하기 위하여 사용되는 2바이트 크기의 스택 오버헤드 비중은 크지 않다. 또한 선점형 EDF 사용자 태스크 스케줄링 기법은 태스크의 문맥전환과 복구에 필요한 모든 레지스터의 개수만큼 확보해야 하기 때문에 가장 많은 메모리 사용량을

표 5 스케줄링 기법의 공간 복잡도

스케줄링 기법	메모리(SRAM) 사용량 (바이트)
비선점형 선입선출	$=N(T)+N(T_{PLF})+2$ = Default
비선점형 EDF	$=\text{Default}+4*N(T)+2$
선점형 EDF	$=\text{Default}+4*N(T)+\#\text{Reg}*(N(T)-1)+\alpha$
비선점형 EDF 태스크 그룹	$=\text{Default}+4*N(T)+2*N(T)+2$

표 6 성능 평가의 분석

성능 평가 요소	실시간성		초소형성	
	마감시한 만족율	평균 대기율	메모리 (SRAM) 사용량	코드 크기
특징	전 구간 평균 6.3% 향상	26.6% 단축	-	1468 바이트
비고	최대 11.0% 향상	-	문맥 보존 메모리 불필요	TinyOS의 기본 사용량은 1302 바이트

요구한다.

표 6은 제안한 태스크 그룹 기반 비선점형 EDF 사용자 태스크 스케줄링 기법의 성능 분석 결과를 요약하였다. 다른 기법들에 비해 마감시한 만족율을 최대 11% (전 구간 평균 6.3%)로 향상시켰다. 그리고 제안한 기법은 기존 스케줄링 기법의 문제점인 TinyOS 플랫폼 태스크들이 라운드 로빈 형태로 스케줄링 되어 요청한 센서 노드 작업들이 모두 늦게 완료되는 것을 제거하여 요청한 센서 노드 작업의 평균 대기율을 약 26.6%로 감소시켰다. 그리고 제안한 기법은 메모리의 사용량을 최소화하도록 설계되어 TinyOS의 초소형성 장점을 유지하였다.

6. 결 론

본 논문에서는 비선점형 태스크 스케줄링 정책을 사용하는 TinyOS 환경에서 실시간성을 제공하는 태스크 그룹 기반의 비선점형 EDF 사용자 태스크 스케줄링 기법을 제안하였다. 제안한 기법은 사용자 태스크의 마감시한 완료에 필요한 일련의 다수 태스크 그룹으로 형성한 후 그룹 내의 태스크들을 일괄적으로 스케줄링하고자 하는 기법이다. 제안한 기법은 기존의 비선점형 TinyOS 시스템과 호환성을 유지하면서 요청한 센서 노드 작업의 마감시한을 보장함과 동시에 평균 응답시간을 최소화하고자 한다. 제안한 기법은 초경량 센서 노드의 하드웨어 플랫폼인 Chipcon사의 CC2420DB 무선 센서 모트(Mote)에서 동작되는 것을 확인하였으며 또한 큐잉 이론 기반의 시뮬레이션 모델을 사용하여 성능 분석을 수행하였다. 제안한 태스크 그룹 기반의 비선점형 EDF 태스크 스케줄링 기법의 동작을 시험한 결과, 요청한 센서 노드 작업의 마감시한을 보장함과 동시에 평균 응답 시간이 향상되었음을 확인하였다.

참 고 문 헌

[1] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS," USENIX/ACM Symposium on Networked Systems Design and Implementation, pp. 1-14, 2004.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han, "MANTIS - system support for MultiModAl NeTworks of In-situ Sensors," ACM International Workshop on Wireless Sensor Networks and Applications, pp. 50-59, 2003.

[3] A. Dunkels, B. Gronvall, and T. Voigt, "CONTIKI - a lghtweight and flexible operating system for tiny networked sensors," IEEE International Conference on Local Computer Networks, pp. 455-462, 2004.

[4] P. Ganesan and A.G. Dean, "Enhancing the AvrX kernel with efficient secure communicaion using software thread integration," Real-Time and Embedded Technology and Applications Symposium, pp. 265-275, 2004.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC language: A holistic approach to networked embedded systems." ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1-11, 2003.

[6] V. Subramonian, H-M. Huang, S. Data, and C. Lu, "Priority scheduling in TinyOS - A case study," Technical Report WUCSE-2003-74, Washington University, 2002.

[7] P. Levis and C. Sharp, "Schedulers and tasks," TinyOS 2.x Extension Proposal 106.

[8] C. Duffy, U. Roedig, J. Herbert, and C. Screenan, "Adding preemption to TinyOS,"

Workshop on Embedded Network Sensors, pp. 88-92, 2007.
[9] CC2420DB. TI and Chipcon Corporation, Available at http://focus.ti.com/docs/prod/folders/ print/cc2420.html

손 치 원

2008년 2월 부산대학교 정보컴퓨터공학부 학사
2010년 2월 부산대학교 컴퓨터공학과 석사
2010년~현재 LG전자 MC 연구소 연구원
관심분야: 임베디드 시스템, 무선 네트워크, 실시간 스케줄링, 대기행렬 이론

탁 성 우

1995년 2월 부산대학교 컴퓨터공학과 학사
1997년 2월 부산대학교 컴퓨터공학과 석사
2003년 2월 미국미주리주립대학교 Computer Science 박사
2004년~현재 부산대학교 정보컴퓨터공학부 부교수
2004년~현재 부산대학교 컴퓨터 및 정보통신 연구소 겸임 연구원
관심분야: 유무선 네트워크, Soc 설계, 실시간 시스템, 위치인식, 최적화 기법, 그래프 이론, 큐잉이론