TinyOS를 위한 동적 우선순위 스케줄러

유종선*, 허신*
*한양대학교 컴퓨터공학과
e-mail : ujong3@hanyang.ac.kr

Dynamic Priority Level Scheduler for TinyOS

Jong-Sun Yoo*, Shin Heu*
*Dept of Computer Science & Engineering, Han-Yang University

요 약

센서 네트워크에 사용되는 운영체제 중 TinyOS는 Event-driven 방식이며 Component 기반의 센서 네트워크 운영체제이다. 이러한 TinyOS는 일단 태스크가 시작되면 마칠 때까지 다른 태스크가 기다려야하는 비선점형(Non-preemption) 방식이다. 최근 연구에서 TinyOS의 빠른 반응성을 위해 선점(Preemption)할 수 있는 기능이 추가되었다. 그러나 프로그래밍할 때 우선순위를 미리 주어줘야 하는 단점이 있다. 본 논문에서는 좀 더 유연하게 우선순위를 변경할 수 있는 방식을 제안하고자 한다.

1. 서론

센서 네트워크는 군사, 과학 분야에서 널리 활용할 수 있는 새로운 기술이다. 이를 이용하여 사람이 아닌 센서노드라는 소형 장치가 실제 필드를 모니터링할 수 있다. 최근 들어 센서 네트워크에 대한 연구가 많이 이루어지고 있다.

센서노드는 센서 네트워크를 구성하는 일부분으로 컴퓨터시스템과 유사한 구조로 이루어 졌지만, 극도로 제한된 자원을 가지고 있다. 예를 들어 센서노드 한 개에는 8bit MCU와 8~128 Kbyte의 플래시 메모리, 512 Byte~4 Kbyte의 RAM으로 구성될 수 있다. 따라서 이러한 극심한 자원의 제약으로 인해 센서 네트워크 운영체제의 크기가 작아질 수밖에 없지만, 사용자의 요구 조건은 만족시킬 수 있는 기능도 있어야 한다.

센서 노드에 들어가는 운영체제 중 대표적인 것이 UC 버클리에서 개발된 초소형 센서 네트워크 운영체제인 TinyOS[1]가 있다. TinyOS는 크기가 4000 Byte이하, 메모리 256 Byte이하의 초소형 운영체제이다. 컴포넌트 기반의 구조로 이루어 졌으며 이벤트 발생에 의해 동작한다. 각각의 컴포넌트는 재사용이 가능하며, 이러한 컴포넌트들을 서로 연결함으로써 TinyOS를 구성한다. TinyOS는 C 언어를 기반으로 만들어진 nesC언어[2]로 프로그래밍 된다.

TinyOS의 프로세스는 태스크와 이벤트로 나뉘며, 미룰 수 있는 계산 작업은 태스크로 사용한다. 태스크는 서로 다른 태스크에 의해 선점이 되지 않는다. 이러한 태스크들을 다루는 TinyOS의 스케줄러는 FIFO 구조의 큐를 사용한다.[3]

최근 연구에서 TinyOS의 빠른 반응성을 위해 태스크간에 선점을 할 수 있도록 Priority Level Scheduler[4]가 제안됐다. 그러나 우선순위를 컴파일 이전에 고정적으로 주어줘야 하기 때문에 유연성이 떨어지는 문제가 있다.

본 논문은 이러한 Priority Level Scheduler가 좀 더 유연하게 활용될 수 있도록 Dynamic Priority Level Scheduler를 제안한다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구를 보이고, 3절에서 Priority Level Scheduler의 한계점을 보인다. 4절에서는 본 논문에서 제안하는 Dynamic Priority Level Scheduler를 보이고, 마지막으로 5절에서 본 논문의 결론을 보인다.

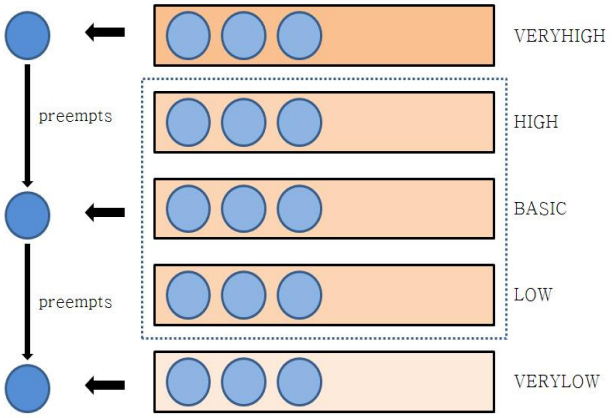
2. 관련연구

Cork 대학교의 Cormac Duffy가 처음으로 TinyOS에 Priority Level Scheduler(PL Scheduler)[4]라는 선점 기능을 추가하였다. 이 PL Scheduler는 일반 운영체제에서 사용되는 선점기능하고 비교하여 다소 독특한 구조를 가진다.

일단 TinyOS의 기본 스케줄러를 기반으로 하여 총 5개의 FIFO 큐를 사용한다. VERYHIGH, HIGH, BASIC, LOW, VERYLOW로 총 5개의 우선순위가 있고, 각 우선순위마다 큐가 하나씩 존재한다.

(그림 1)과 같이 큐가 이루어져 있다. (그림 1)에서 파란색 원은 태스크를 의미하고 각 태스크는 각 우선순위 큐에 저장되어 있다. 최상위에 있는 VERYHIGH 우선순위 작업은 하위에 있는 모든 작업을 선점한다. 만약 하위 큐의 작업이 수행 중에 VERYHIGH의 작업이 수행되려고 할 때 바로 선점하여 VERYHIGH 순위의 작업이 수행된

다. 중간에 있는 HIGH, BASIC, LOW 우선순위의 작업은 기존의 TinyOS처럼 서로 선점할 수 없고 원자적으로 수행된다. 여기서 HIGH 우선순위의 큐가 먼저 수행되고 BASIC 우선순위, LOW 우선순위 순으로 수행된다. 최하위에 있는 VERYLOW 우선순위 작업은 다른 순위에 의해 선점 당한다.



(그림 1) PL Scheduler의 구조

선점이 수행되면 현재 수행되는 작업의 정보는 메모리 어딘가에 저장되고 Context Switch가 수행된다. *push* 함수를 사용하여 모든 레지스터 값을 메모리에 저장하고, *pop* 함수를 사용하여 메모리에 저장된 값을 레지스터로 복구한다.

3. Priority Level Scheduler의 한계

PL Scheduler는 TinyOS의 반응성을 높이기 위해 추가된 선점형 스케줄러 알고리즘이다. TinyOS에 선점을 추가한 시도는 좋지만, 우선순위를 미리 정적으로 주어야 하는 한계가 있다.

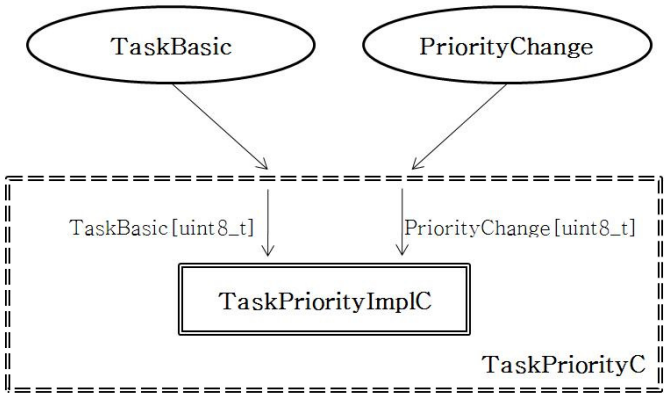
```
configuration exampleAppC{
}
implementation{
   // 우선순위가 LOW인 태스크 생성
   components new LowTaskC() as LowPriorityTask;
   // 우선순위가 VERYHIGH인 태스크 생성
   components new VeryHighTaskC() as VeryHighPriorityTask;
   // 우선순위가 VERYLOW인 태스크 생성
   components new VeryLowTaskC() as VeryLowPriorityTask;
   // exampleAppP의 컴포넌트와 메인
   components exampleAppP, MainC;

                . . .

   // 각각 태스크에 대해 와이어링(wiring)함
   exampleAppP.LowPriority -> LowPriorityTask;
   exampleAppP.VeryHighPriority -> VeryHighPriorityTask;
   exampleAppP.VeryLowPriority -> VeryLowPriorityTask;

                . . .
}
```

(그림 2) PL Scheduler를 사용한 프로그램

PL Scheduler를 사용하여 응용프로그램을 개발하면 (그림 2)와 같이 할 수 있다. (그림 2)에서는 VERYHIGH, LOW, VERYLOW에 해당하는 작업을 생성한다. 그러나 각 우선순위에 해당하는 작업을 생성할 때마다 서로 다른 컴포넌트를 사용하는 것을 볼 수 있다. 이는 응용프로그램을 개발할 때마다 우선순위를 미리 정해야 할 뿐만 아니라 수행 중에도 변경할 수 없는 단점이 있게 된다. 따라서 유용하게 사용하기 위해 좀 더 유동적인 방법이 필요하다.

4. Dynamic Priority Level Scheduler

앞 절에서 PL Scheduler의 문제점을 살펴봤다. 이번 절에서는 본 논문에서 제시하는 Dynamic Priority Level Scheduler(DPL Scheduler)의 디자인을 알아본다. DPL Scheduler의 핵심은 우선순위마다 따로 존재했던 컴포넌트들을 하나의 컴포넌트가 중재하여 관리하는 것이다. 개발자 입장에서 중재하는 하나의 컴포넌트만 사용하면 되기에 훨씬 간단하게 프로그래밍 할 수 있게 된다. 또한, 중재하는 컴포넌트를 통해 수행 중에 우선순위로 바꿀 수 있는 이득이 있다.



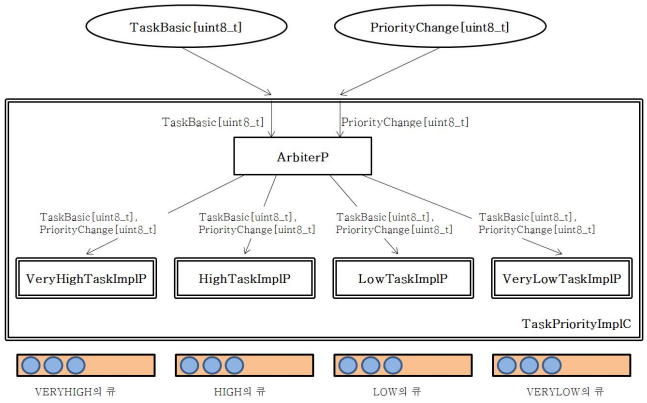
(그림 3) DPL Scheduler의 컴포넌트 구조

기본적인 컴포넌트의 구조는 (그림 3)과 같다. (그림 3)은 DPL Scheduler의 구조 중 상위 컴포넌트들을 나타낸다. 응용프로그램 개발자는 상위 컴포넌트인 TaskPriorityC를 연결해서 DPL Scheduler를 사용할 수 있다. TaskPriorityC 컴포넌트가 제공하는 인터페이스는 TaskBasic과 PriorityChange이며, TaskBasic 인터페이스는 기본적으로 TinyOS에서 제공해주는 것이고, Priority Change는 추가된 인터페이스로서 수행 중에 우선순위를 바꿀 수 있도록 한다.

TaskPriorityC는 gerneric 컴포넌트로 여러 개의 인스턴스를 생성할 수 있게 한다. TaskPriorityC가 사용하는 TaskPriorityImplC는 실질적인 구현 부분으로 제공하는 인터페이스는 TaskPriorityC와 동일하다. 그러나 태스크를 구분하기 위해 TaskBasic[uint8_t], PriorityChange[unit

8_t] 인터페이스같이 [uint8_t]라는 8bit 정수 값을 식별자로 사용한다.

DPL Scheduler의 하위 구조 중 TaskPriorityImplC의 구조는 (그림 4)와 같다.

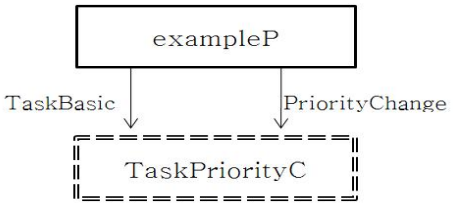


(그림 4) DPL Scheduler의 하위 컴포넌트

중간에 ArbiterP를 제외하고 하위 4개의 컴포넌트들은 기존의 PL Scheduler 구조의 일부분이다. 4개의 컴포넌트들은 각각 큐가 존재하게 된다.

ArbiterP 컴포넌트는 TaskPriorityImplC 컴포넌트에서 온 post command를 받은 후 우선순위에 따라 아래에 있는 적절한 우선순위 컴포넌트에게 post command를 보낸다. post command를 받은 컴포넌트는 자기 우선순위 큐에 저장하거나 VERYHIGH 인 경우 선점을 한다. 스케줄러에 의해 태스크가 수행하는 차례가 되면, 큐에서 event를 발생하여 사용자 응용프로그램까지 event가 도달하게 되면 사용자 태스크가 수행된다. 또한 상위 컴포넌트에서 어떤 태스크의 우선순위 변경 command가 오면, 해당 태스크가 저장된 우선순위 큐에서 그 태스크를 제거하고, 새로운 우선순위 컴포넌트로 post command를 보낸다. 그러나 현재 CPU에서 수행 중인 태스크의 우선순위는 변경되지 않는다.

PL Scheduler에서는 특정 우선순위를 사용하려면 (그림 4)의 하단의 4개의 컴포넌트 중 하나를 선택해서 사용해야 한다.



(그림 5) DPL Scheduler 사용예시(wiring 부분)

그러나 본 논문이 제안하는 DPL Scheduler는 TaskPriorityC 컴포넌트만 쓰면 가능하다. 또한 PriorityChange 인터페이스를 통해 큐에 대기 중인 우선순위를 바꿀 수 있다.

본 논문에서 제안한 DPL Scheduler를 사용한 예시로 컴포넌트 연결은 (그림 5)와 같다. 또한, 개략적인 소스 구조는 (그림 6)과 같다.

```
module exampleP{
        . . .
    // 태스크 인터페이스 사용
    uses interface TaskBasic as PriorityTask;
    // 우선순위 변경 인터페이스 사용
    uses interface PriorityChange as Changer;
        . . .
}
implementation{

                . . .

        // 태스크를 HIGH 우선순위로 post함
        call PriorityTask.postTask(HIGH);

                . . .

    // 실제 태스크의 코드
    event void PriorityTask.runTask(){
                . . .
                // 태스크 수행부분
                . . .
    }

                . . .
}
```

(그림 6) DPL Scheduler 사용예시(모듈부분)

예제 프로그램인 exampleP 컴포넌트를 TaskPriorityC와 연결해 주면 된다. 실제 태스크 구현 부분은 (그림 6)의 소스와 같은 방식으로 하면 된다. 태스크를 post 할 때는 TaskBasic의 postTask()를 호출하면 되고, 인자 값으로 우선순위 값을 넣어 준다. 태스크 구현은 TaskBasic의 runTask()에 구현하면 된다.

5. 결론

본 논문은 TinyOS에 사용된 선점형 스케줄러인 Priority Level Scheduler의 한계를 분석하고, 이를 좀 더 유연하게 사용할 수 있는 방법을 제시하였다. Priority Level Scheduler는 우선순위마다 특정 컴포넌트를 사용해서 정적으로 사용할 수밖에 없었다. 그러나 본 논문에서 제시하는 방법은 우선순위 컴포넌트들을 한 번에 묶어서 이를 중재하는 컴포넌트인 ArbiterP를 사용하여 좀 더 유연하게 동작할 수 있게 한다.

향후에는 Dynamic Priority Level Scheduler를 사용함으로써 TinyOS의 실시간성 보장 연구에 보탬이 될 것이다.

참고문헌

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler,

and K. Pister. “System architecture directions for networked sensors.” In Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 93 - 104, Cambridge, MA, Nov. 2000.

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

[3] TEP 106 : “Schedulers and Tasks”
http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html

[4] Cormac Duffy, Utz Roedig, John Herbert and Cormac J. Sreenan. “Adding preemption to tinyos.” In To appear in the Fourth Workshop on Embedded Networked Sensors (EmNets 2007), University College Cork, Ireland. ACM Digital Library, June 2007.