센서 운영 체제를 위한 공유 스택 기법의 성능 분석

(Performance Analysis of Shared Stack Management for Sensor Operating Systems)

구 본 철 † 허 준 영 †† 홍 지 만 ††† 조 유 근 ††††
(Boncheol Gu) (Junyoung Heo) (Jiman Hong) (Yookun Cho)

요 약 무선 센서 네트워크의 발달에 따라 그 응용분야는 점점 더 복잡해져 가고 있음에도 불구하고, 대부분의 센서 노드 플랫폼은 여전히 심각한 자원 제약을 가지고 있다. 특히 적은 메모리 공간과 메모리 관리 유닛(MMU)의 부재는 스레드의 스택 관리에 있어 메모리 공간 낭비, 스택 오버플로우와 같은 문제를 야기해왔다. 이에 다 수의 스레드가 하나의 스택을 공유 함으로써 기존의 고정 크기 스택에 의해 낭비되는 메모리의 양을 최소화 시킬 수 있는 공유 스택 기법이 제안되었다. 본 논문에서는, 고정 크기 스택 기법과 공유 스택 기법의 수학적 분석 모델을 제시하였다. 그 모델을 바탕으로 각각의 스택 오버플로우 확률을 계산하고 공유 스택 기법이 고정 크기 스택보다 더 안정적임을 확인하였다.

키워드 : 센서 운영 체제, 스택 분석, 공유 스택, 스택 오버플로우

Abstract In spite of increasing complexity of wireless sensor network applications, most of the sensor node platforms still have severe resource constraints. Especially a small amount of memory and absence of a memory management unit (MMU) cause many problems in managing application thread stacks. Hence, a shared-stack was proposed, which allows several threads to share one single stack for minimizing the amount of memory wasted by fixed-size stacks. In this paper, we present the memory usage models for thread stacks by deriving the overflow probability of the fixed-size stack and the shared-stack and also show that the shared-stack is more reliable than the fixed-size stack.

Key words : sensor operating systems, stack analysis, shared-stack, stack overflow

1. 서 론

무선 센서 네트워크(Wireless Sensor Network)는 온도, 습도와 같은 환경 데이타를 주기적으로 수집하는 다수의 소형 센서 노드로 구성된 무선 네트워크이다. 그 응용 분야는 전장 감시부터 환경 감시, 건강 관리, 재고 추적에 이르기까지 많은 분야에서 널리 사용되고 있으며, 그 활용분야가 폭넓어짐에 따라 무선 센서 네트워크를 구성하고 있는 센서 노드들이 수행해야 할 작업의 종류와 복잡도는 점점 증가하고 있다. 비록 하나의 센서 노드가 마이크로 프로세서와 메모리, 무선 통신 장치를 갖춘 작은 컴퓨터처럼 작동하지만, 계산 능력, 메모리, 전력 공급의 측면에서 심각한 자원제약을 가지고 있다[1,2]. 그렇기 때문에 센서 운영체제를 위한 운영체제, 즉 센서 운영체제는 다수의 병렬적이고 복잡한 작업을 자원 제약적인 환경에서 효율적으로 관리할 수 있어야 한다.

TinyOS[3], Sensor OS[4], Contiki[5]와 같은 대다수의 센서 운영체제들은 효율적인 작업관리를 위해 이벤트

· 본 연구는 2단계 BK21 사업과 숭실대학교 교내 연구비 지원으로 이루어졌음
· 이 논문은 2007 한국컴퓨터종합학술대회에서 '센서 운영 체제를 위한 공유 스택 기법의 성능 분석'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : 서울대학교 컴퓨터공학부
bcgu@os.snu.ac.kr
†† 학생회원 : 서울대학교 컴퓨터공학부
jyheo@os.snu.ac.kr
††† 종신회원 : 숭실대학교 컴퓨터학부
jiman@ssu.ac.kr
(Corresponding author임)
†††† 종신회원 : 서울대학교 컴퓨터공학부 교수
ykcho@snu.ac.kr

논문접수 : 2007년 9월 27일
심사완료 : 2007년 11월 22일

정보과학회논문지: 시스템 및 이론 제35권 제1호(2008.2)

드리븐 모델을 선택했다. 이벤트 드리븐 모델이 센서 네트워크 환경에서 명백하게 자원 효율적인 작업 관리 모델이라고 할 수 있지만, 이벤트 드리븐 모델을 기반으로 하는 프로그래밍 방법은 응용 프로그램의 개발과 디버깅 그리고 유지 보수를 어렵게 만든다[6]. 이러한 문제점은 무선 센서네트워크의 응용 프로그램이 점점 복잡해 짐에 따라 더욱 부각되어 왔고 이에 MANTIS OS[7]와 같이 프로그램의 실행 흐름을 쉽게 파악할 수 있고 콜스택(call stack)을 사용한 디버깅과 예외처리를 지원하는 멀티스레드 기반의 센서 운영체제가 등장하기 시작했다.

하지만, 멀티스레드의 지원을 위하여 각각의 스레드에 고정된 크기의 메모리 공간을 스택에 할당하는 고정 스택 기법은 센서 운영체제의 메모리 오버헤드를 증가시킨다[8]. 일단 스택에 할당된 메모리 공간은 스레드의 생명주기 동안 크기가 변하지 않기 때문에 메모리 할당 시점에 충분한 크기를 할당하여 스택 오버플로우를 방지해야만 한다. 이러한 방식의 문제점은 스레드 스택에 할당된 메모리 공간이 실제로 스레드가 실행되는 동안 대부분 사용되지 않는다는 점에 있다. 이렇게 실제로 사용되지 않는 공간은 센서 운영체제의 메모리 오버헤드를 유발하고 스레드 개수의 증가는 메모리 부족 오류나 스택 오버플로우의 직접적인 원인이 된다. 이러한 문제점을 해결하기 위해 [9]에서 공유 스택 기법이 제안되었고 이를 적용하면, 다수의 스레드가 하나의 공유 스택을 런타임 스택으로 공유하고 각각의 스레드가 컨텍스트 유지를 위해 실제로 필요한 크기만큼의 메모리 공간을 점유함으로써 고정 크기 스택의 메모리 오버헤드를 감소시킬 수 있다.

스택 오버플로우는 MMU와 같은 하드웨어의 메모리 보호 지원을 기대할 수 없는 센서 노드의 안정성을 심각하게 저해한다. 또한, 센서 노드의 불안정한 상태는 멀티 홉(multi-hop) 통신을 사용하는 무선 센서 네트워크의 네트워크 단절과 같은 치명적인 성능저하를 불러올 수 있다. 본 논문에서는 고정 크기 스택 기법과 제안된 공유 스택 기법의 성능을 분석하기 위해 수학적 모델을 제시하고 두 기법의 오버플로우 확률을 도출하였다. 그 결과를 바탕으로 우리는 공유 스택 기법이 고정 크기 스택 기법보다 더 안정적임을 확인하였다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 고정 크기 스택과 공유 스택 기법에 대하여 설명하고 3장에서는 각각의 스택 관리 기법의 성능 분석을 위한 수학적 모델을 제시한다. 4장에서는 제시된 모델을 이용하여 성능을 비교하고 5장에서 결론을 맺는다.

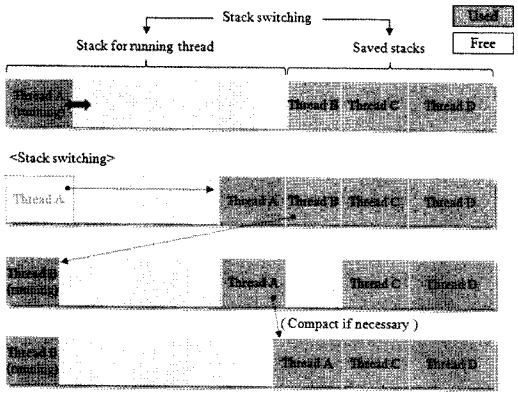
2. 고정 크기 스택 기법과 공유 스택 기법

대부분의 멀티스레드 기반 운영체제의 스레드는 고유의 런타임 스택을 필요로 한다. 각각의 스택에 할당된 메모리 공간은 해당 스택을 사용하는 스레드가 연산을 수행하는 동안 오버플로우가 발생하지 않도록 충분한 메모리 공간을 유지하고 있어야 하며, 한 번 스택에 할당된 메모리 공간은 스레드가 수행을 종료할 때까지 계속 메모리 공간을 점유하게 된다. 이러한 방식의 기법을 우리는 고정 크기 스택 기법이라고 부른다. 고정 크기 스택 기법의 단점은 스레드가 실행하는 동안 해당 스택에 할당된 공간의 대부분을 실제로 사용하지 않기 때문에 많은 양의 메모리 공간이 낭비될 수 있다는 것이다. 일반적인 컴퓨팅 환경의 경우 MMU와 충분한 크기의 보조 저장소를 사용하여 가상 메모리 시스템을 구현함으로써 이러한 메모리 공간 오버헤드를 완화할 수 있다. 가상 메모리 시스템에서 사용하지 않은 공간은 물리적인 메모리 공간이 아니라 단지 가상 주소 공간이기 때문이다[10].

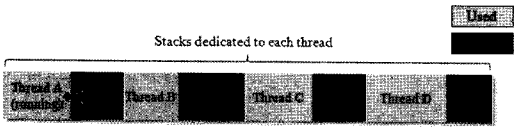
하지만 센서 노드의 경우 그 크기와 비용의 제약으로 인하여 향후 몇 년 동안 가상 메모리 시스템을 구현할 수 있는 하드웨어적인 지원을 기대하기 어렵다. 센서 노드는 일단 스레드 스택에 메모리 공간을 할당하면 그 공간은 실질적인 메모리 공간을 차지하는 단순한 메모리 관리 기법을 사용한다. 만약 해당 시스템이 모든 스레드에게 할당할 만큼의 충분한 메모리 공간을 가지고 있다면 이는 그다지 큰 문제는 아닐 것이다. 하지만 무선 센서 노드와 같은 메모리 제약적인 시스템에서 스레드의 수적인 증가는 메모리 부족 오류나 스택 오버플로우의 직접적인 원인이 된다. 그러므로 센서 운영체제에서 고정 크기 스택 기법을 적용하는 것은 메모리의 효율성과 시스템의 안정성 측면에서 적합하지 않다.

공유 스택 기법은 고정 크기 스택을 사용함으로써 발생하는 메모리 오버헤드를 줄이기 위해 개발되었다[9]. 그림 1은 고정 크기 스택 기법과 공유 스택 기법이 수행하는 스레드 스택 관리 방식을 보여주고 있다. 그림 1(a)에서 볼 수 있듯이 공유 스택 기법은 모든 스레드가 하나의 공유 스택을 런타임 스택으로서 공유하며, 오직 현재 실행 중인 스레드만이 공유 스택을 점유할 수 있다. 실행 중인 스레드가 선점 지점에서 대기 상태로 들어갈 경우 힙과 같은 스택 저장소에 컨텍스트를 복사하여 저장하고, 다음 실행될 스레드는 저장되어 있는 자신의 컨텍스트를 공유스택으로 다시 복사하여 실행을 재개 한다.

고정 스택을 사용하는 스레드는 스택 오버플로우를 고려하여 최대한의 스택크기를 실행 주기 내내 보유하고 있기 때문에 많은 양의 메모리를 낭비할 수 있다. 그림 1(b)에서 볼 수 있듯이 고정 크기 스택에서 낭비되는 메모리의 양이 스레드의 개수에 비례하여 증가하는 반면, 그림 1(a)에서 볼 수 있듯이 공유 스택 기법에서



(a) 공유 스택 기법



(b) 고정 크기 스택 기법

그림 1 스택 기법에 따른 메모리 구성

는 현재 실행 중인 스레드만이 여분의 메모리 공간을 추가로 점유하고 있을 뿐 나머지 스레드들은 선점 지점에서 실제로 필요로 하는 양의 메모리 공간을 점유한다. 그렇기 때문에 고정 크기 스택 기법의 메모리 오버헤드는 공유 스택 기법을 사용함으로써 상당 부분 완화 시킬 수 있다. 예를 들면 그림 1에서 보여지듯이 고정 스택을 사용할 경우 하나의 시스템에 동시에 4개의 스레드만이 존재할 수 있지만, 공유 스택을 사용할 경우에는 4개 이상의 스레드가 존재할 수 있는 여분의 메모리 공간이 존재함을 알 수 있다. 공유 스택의 가장 중요한 특징은 모든 스레드가 실행 시간 동안 실제로 사용하고 있는 양만큼의 메모리 공간을 점유한다는 사실이다.

비록 공유 스택 기법이 문맥 교환 시 과도한 복사 오버헤드를 유발할 수 있음에도 센서 운영체제에서 사용될 수 있는 이유는 바로 센서 네트워크 응용프로그램의 대부분이 특정 이벤트를 기다리고 그에 해당하는 작업을 처리하는 I/O에 바운드된 작업들이기 때문이다[11, 12]. 이는 CPU 자원이 메모리에 비해 상대적으로 덜 제약적임을 의미하고, 공유 스택 기법을 통하여 메모리 오버헤드를 CPU 오버헤드로 교환할 수 있다. 또한 협동적인 스레드(cooperative thread)를 사용하면 문맥 교환 횟수를 감소시킬 수 있고 전체적인 CPU 오버헤드를 감소시킬 수 있다.

3. 스택 사용 모델

3.1 표기

스택 사용 모델을 기술하기 위하여 다음 표기를 정의한다.

- M : 스택을 위해 할당된 메모리 크기
- n : 시스템 내의 전체 스레드 개수
- k : 각 스레드 스택에 할당된 메모리 크기 ($=M/n$)
- S_i : 스택이 사용한 메모리의 양이 i 인 상태
- $P_{i,j}$: S_i 에서 S_j 가 될 확률
- F_OFP_1 : 고정 크기 스택 하나의 오버플로우 확률
- F_OFP_n : 고정 크기 스택 n 개의 오버플로우 확률
- S_OFP : 공유 스택 모델의 오버플로우 확률

3.2 고정 크기 스택의 오버플로우 확률

스택의 동작은 데이타를 삽입, 삭제, 읽기 세 가지로 분류된다. 이 때 P_{ij} 는 S_i 이전의 상태와는 상관없이, 오직 S_i 에서의 삽입 확률, 삭제 확률, 읽기 확률 세 가지만으로 결정된다. 또한 S_i 와 S_{i+1} $(i + 1 < k)$ 사이에는 $S_{i'}$ $(0 < i' < 1)$가 존재하지 않으므로 각 상태는 이산적이다. 따라서 S_i 의 변화 양상은 이산 매개변수 마코프 체인(discrete parameter Markov chain)으로 표현될 수 있다.

S_i 에 대해서 메모리 연산 각 회 마다 삽입, 삭제, 읽기가 될 경우 상태는 각각 S_{i+1}, S_{i-1}, S_i로 전이 된다. 즉, 상태의 전이 과정은 1회 메모리 연산에 대해서 어떠한 경우에도 한 단계 이상 일어나지 않기 때문에 이 경우는 이산 매개변수 마코프 체인의 특수한 경우인 출생-사망 과정(birth-death process)이다. 현재 상태가 S_i 일 때 삽입, 삭제, 읽기 연산의 확률을 b_i, d_i, a_i 라고 한다면 각각은 다음과 같이 표현된다.

$$b_i = p_{i,i+1} \qquad (i \geq 0)$$
$$d_i = p_{i,i-1} \qquad (i \geq 1)$$
$$a_i = p_{i,i} \qquad (i \geq 0)$$

모든 S_i 에 대해서 일어 날 수 있는 연산은 위 세 가지에 한정되므로 $b_i+d_i+a_i=1$ 인 것은 자명하다. 임의의 b_i, d_i 는 일반적인 응용 프로그램에 대해서 경향성을 가지지 않는다고 해도 무방하므로 모든 b_i, d_i 는 같고 이 값을 각각 b 와 d 로 둘 수 있으며 결과적으로 고정 크기 스택의 상태 다이어그램으로 그림 2처럼 나타낼 수 있다.

통상적인 스택 연산의 경우 b_0 의 값은 0이 아니므로 본 마코프 체인은 약분 불능이고 (irreducible), 비주기적(aperiodic)이다. 따라서 초기값에 상관없이 모든 S_i

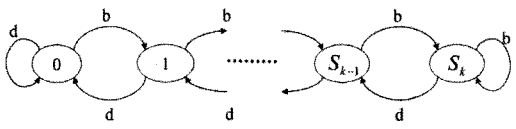


그림 2 고정 크기 스택의 상태 다이어그램

에 대하여 극한 확률(limiting probability) v_i가 존재하고 b의 d에 대한 비율, 즉 탄생-죽음 비율 r을 b/d라고 한다면 v_i, v_0는 다음과 같이 쓸 수 있다[13].

$$v_i = \frac{b}{d}v_{i-1} = \begin{cases} rv_{i-1} & (b \neq d) \\ v_{i-1} & (b = d) \end{cases} \quad (1)$$

$$v_0 = \begin{cases} \dfrac{1-\left(\dfrac{b}{d}\right)}{1-\left(\dfrac{b}{d}\right)^{k+1}} = \dfrac{1-r}{1-r^{k+1}} & (b \neq d) \\ \dfrac{1}{k+1} & (b = d) \end{cases} \quad (2)$$

이를 통하여 F_OFP_1을 구하여 보면, 그림 2의 S_k 상태에서 b 만큼의 확률로 삽입 연산을 수행하는 경우이므로 다음과 같이 쓸 수 있다.

$$F\_OFP_1 = bv_k = b\left(\frac{b}{d}\right)^k v_0 = \begin{cases} br^k \dfrac{1-r}{1-r^{k+1}} & (b \neq d) \\ b \cdot \dfrac{1}{k+1} & (b = d) \end{cases}$$

위 식을 전체 메모리 M에 대해서 n개의 스레드가 존재할 때로 확장시켜 보면, 각 스레드 스택에 할당된 메모리의 크기 $k = M / n$이 되고 이때 F_OFP_n은 n개의 스레드 스택 중 한 개도 오버플로우가 일어나지 않은 경우에 대한 여사건의 확률과 같다. 즉,

$$F\_OFP_n = 1-(1-F\_OFP_1)^n = \begin{cases} 1-(1-br^k \dfrac{1-r}{1-r^{k+1}})^n & (b \neq d) \\ 1-(1-b \cdot \dfrac{1}{k+1})^n & (b = d) \end{cases}$$

이다.

3.3 공유 스택의 오버플로우 확률

공유 스택에서의 메모리는 힙 공간, 공유 스택 공간, 그리고 대기 된 스레드들의 문맥을 담고 있는 저장 공간으로 구분된다. 이들 중 힙과 공유 스택 공간은 현재 실행중인 스레드 뿐만 아니라 대기 된 스레드들도 문맥 교환에 따라서 임의로 사용할 수 있다는 관점에서 봤을 때 하나의 통일된 가용 공간으로 간주 할 수 있다. 이 때 T를 스레드의 크기라고 했을 때 그림 1의 (a)에서 나타낸 공유 스택의 메모리 사용 모델은 다음과 같이 변화한다.

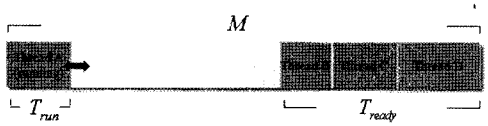


그림 3 공유 스택의 메모리 구성

또한 가용 공간의 크기(M_{free})는 현재 실행 중인 스레드의 스택 크기(T_{run})와 대기 중인 스레드들의 문맥을 저장한 공간의 크기(T_{ready})를 사용하여 다음과 같이 표현된다.

$$M_{free} = M - T_{run} - T_{ready}$$

가용 공간에 대해서 고정 크기 스택의 경우와 마찬가지로 메모리 연산은 삽입, 삭제, 읽기로 구별되고, 각 메모리 연산 1회에 대해서 전이할 수 있는 범위는 1단계 이하이며 각 상태는 이산적이므로 공유 스택의 S_i 역시 이산 매개변수 마코프 체인으로 표현할 수 있다. 3.2 에서의 확률 계산 과정과 상황을 동일하게 만들기 위하여 이 때 사용된 가정들을 적용 하면 b_i=b, d_i=d 이다. 다만, 공유 스택 모델의 경우 T_{run}이 0일 때 삭제 연산이 일어난다고 하더라도 $S_i \rightarrow S_{i-1}$ $(1 \leq i \leq M)$ 상태로 전이되는 것이 아니라 $S_i \rightarrow S_i$로 동일하게 상태를 유지하게 된다. T_{run}이 0일 상황이라는 것은 현재 실행 중인 스레드 스택의 크기가 0 인 것으로(가), S_0 상태와는 구별되는 것임을 유의해야 한다. 따라서 위 상황이 발생할 확률을 p_0라고 표기한다면 p_0는 3.2 절의 식 (1), 식 (2)에서의 v_0와 같으므로,

$$p_0 = \begin{cases} \dfrac{1-r}{1-r^{k+1}} & (b \neq d) \\ \dfrac{1}{k+1} & (b = d) \end{cases}$$

가 된다. 또한 가용 공간에 대한 삭제 연산은 (가) 상황일 경우와 (가) 상황이 아닐 경우로 분할 되고, (가) 상황일 경우의 확률을 p_0라고 했을 때 (가) 상황이 아닐 경우의 확률은 $(1-p_0)$로 표시할 수 있으므로 $b + d = 1$이고 이 식은 다음과 같이 다시 쓸 수 있다.

$$b + p_0 d + (1-p_0)d = 1$$

이러한 조건들을 S_i에 대한 상태도로 표시하면 그림 4와 같다.

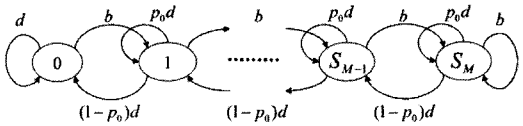


그림 4 공유 스택의 상태 다이어그램

그림 4에서 오버플로우가 일어나는 경우는 현재 상태가 S_M에 이른 상황에서 가용 공간 메모리에 대해서 삽입 연산이 일어났을 때이다. 각각의 상태 S_i에 이르는 확률을 v_i라고 정의하면 S_OFP는 다음과 같이 표시된다.

$$S\_OFP = bv_m$$

그림 4에 표시된 상태들의 확률을 구해보면 모든 상

태는 한 단계 이내의 전이만이 이뤄지므로 v_i는 v_{i-1}, v_i, v_{i+1}를 통하여 S_{i-1}에서 삽입 될 경우, S_i일 때 (가) 상황이 발생할 경우, S_{i+1}에서 삭제 연산이 일어날 경우의 합으로 표현할 수 있다. 이때 v_0는 S_0에 대한 이전 상태가 존재하지 않으므로 식 (3)으로 표시되고, 이외의 v_i $(1 \le i \le M-1)$를 나타낸 것은 식 (4)와 같다.

$$v_0 = dv_0 + (1-p_0)dv_1 \tag{3}$$

$$v_i = bv_{i-1} + p_0 dv_i + (1-p_0)dv_{i+1} \tag{4}$$

이들을 이용하여,

$$v_i = \frac{b}{(1-p_0)d}v_{i-1} = \left(\frac{b}{(1-p_0)d}\right)^i v_0 = \left(\frac{r}{1-p_0}\right)^i v_0$$

를 얻는다.

또한 모든 S_i에 대한 확률 v_i들의 합은 1이므로,

$$1 = \sum_{i=0}^{M} v_i = \sum_{i=0}^{M}\left(\frac{r}{1-p_0}\right)^i v_0$$

이로부터,

$$v_0 = \frac{1-\frac{r}{1-p_0}}{1-\left(\frac{r}{1-p_0}\right)^{M+1}}$$

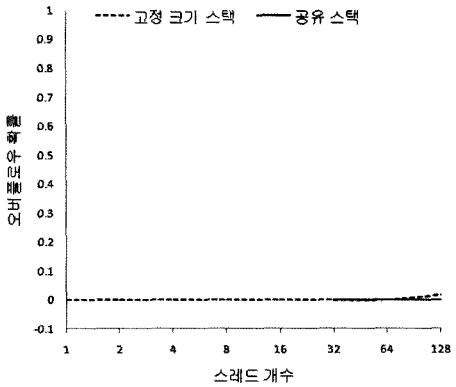
이다.

따라서 S_OFP는 다음과 같이 다시 쓸 수 있다.

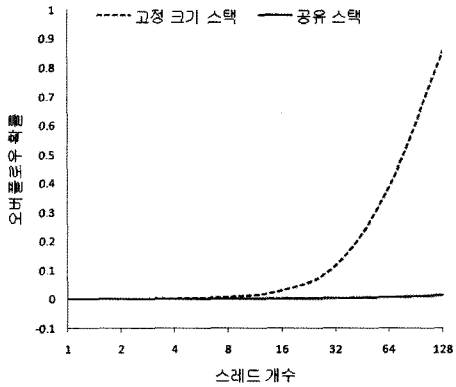
$$S\_OFP = bv_M = \begin{cases} b\left(\frac{r}{1-p_0}\right)^M \frac{1-\frac{r}{1-p_0}}{1-\left(\frac{r}{1-p_0}\right)^{M+1}} & (b \neq d) \\ \frac{1}{2+2\cdot\frac{M}{n}-2\cdot\frac{M}{n}\left(\frac{M}{n+M}\right)^M} & (b = d) \end{cases}$$

4. 성능 비교

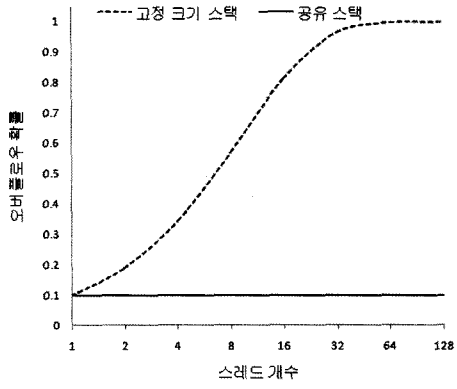
통상적으로, 센서 네트워크는 크기가 작고 다수의 노드들이 하나의 시스템을 이루는 까닭에 일반적인 내장형 시스템 보다 전력에 대한 제약이 크고 구성원 중 일부에만 문제가 발생한다고 하더라도 네트워크 분할 등의 치명적인 상황을 맞을 수 있다. 이 때문에 고성능의 처리 속도와 고속 데이타 송수신을 지양하고 속도 보다는 동작에 대한 안정성을 위주로 설계되어 있는 것이 일반적이다. 그리하여 이에 대한 성능 지표로 메모리 오버플로우 확률을 고려하는 것은 데이타 전송 속도 등과 같은 범용 운영체제에서의 지표 대비 타당하다고 할 수 있다. 본 장에서는 센서 네트워크를 이루는 일반적인 노드에 대한 조건을 설정하고 3장에서 다룬 각 사용 모델에 대해서 메모리 오버플로우 확률을 도출함으로써 안정성에 대한 성능 비교를 행하였다.



(a) b=0.45, r=0.818



(b) b=0.5, r=1.0



(c) b=0.55, r=1.222

그림 5 오버플로우 확률 비교 그래프

F_OFP와 S_OFP는 공히 메모리 크기 M, 스레드의 개수 n, 메모리에 대한 자료 삽입 확률 b에 대한 3

변수 함수이다. M을 현존하는 센서 네트워크 노드와 유사한 값으로 설정(M=4096)하고 b를 적절히 변화시켜 스레드 개수 n에 따른 확률을 얻어 냈다. 보편적으로 메모리의 삽입과 삭제 연산은 거의 같은 비율로 발생한다는 점에서 $r(=b/d)$ 값은 1과 비슷할 때에 현실 상황을 반영하는 유효한 값이라고 볼 수 있다. 이에 b=0.45, 0.5, 0.55 각각에 대해서 n에 대한 확률을 그림 5에서 그래프로 표시하였다.

각 경우 마다 고정 크기 스택을 갖는 모델, 공유 스택 모델 모두 스레드의 개수가 늘어남에 따라서 오버플로우 확률이 증가하는 경향성을 보인다. 그러나 공유 스택의 경우는 스레드 개수의 증가에도 오버플로우 확률의 증분이 완만하나, 고정 크기 스택의 경우 일정 시점 이후부터는 급격히 증가하여 확률값이 1에 수렴한다. 또한 오버플로우 확률의 차이는 n이 증가함에 따라 커지는 경향을 보이지만, 고정 크기 스택의 확률이 이미 1에 근접한 시점 이후에는 공유 스택 모델의 확률 증가분이 더욱 크다는 사실을 그림 5(c)의 n=128인 경우에서 확인 할 수 있다.

5. 결 론

무선 센서 네트워크 응용분야의 발달은 각각의 센서 노드에게 점점 더 복잡한 작업을 수행하도록 요구하고 있지만 센서 노드의 하드웨어는 여전히 심각한 자원 제약을 가진 채 남아 있다. 즉, 무선 센서 네트워크의 소프트웨어와 하드웨어 사이의 격차는 계속해서 넓어지고 있음이 분명하다. 공유 스택 기법은 하드웨어를 향상시키는 대신 소프트웨어의 개선을 통하여 이러한 격차를 줄일 수 있도록 설계되었으며, 구체적으로 CPU 자원을 희생하여 메모리의 효율성과 시스템의 안정성을 향상시킨 기법이다. 이러한 접근 방법은 센서 네트워크의 대다수의 응용 프로그램들이 특정 이벤트를 기다리는 데에 실행 시간의 대부분을 할애하고 있기 때문에 타당하다고 볼 수 있다. 본 논문에서는 공유 스택 기법의 안정성을 수학적 모델을 통하여 분석하였으며 기존의 고정 크기 스택과 비교하여 보다 더 안정적임을 확인하였다.

참 고 문 헌

[1] M. Tubaishat and S. Madria, "Sensor networks: An overview," IEEE Potentials, Vol.22, No.2, pp. 20-23, 2003.

[2] D.E. Culler, D. Estrin, and M.B. Srivastava, "Guest editors' introduction: Overview of sensor networks," IEEE Computer, Vol.37, No.8, pp. 41-49, 2004.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," Proc. of the 9th international conference on Architectural support for programming languages and operating systems, Cambridge, MA, USA, pp. 93-104, 2000.

[4] C.C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," Proc. of the 3rd international conference on Mobile systems, applications, and services, New York, NY, USA, pp. 163-176, 2005.

[5] A. Dunkels, B. Gr¨onvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," Proc. of the 1st IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, 2004.

[6] J.R. von Behren, J. Condit, and E.A. Brewer, "Why events are a bad idea (for high-concurrency servers)," Proc. of the 9th Workshop on Hot Topics in Operating Systems, Lihue(Kauai), Hawaii, USA, pp. 19-24, 2003.

[7] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," MONET, Vol.10, No.4, pp. 563-579, 2005.

[8] A. Gustafsson, "Threads without the pain," ACM Queue: Tomorrow's Computing Today, Vol.3, No.9, pp. 42-47, 2005.

[9] B. Gu, Y. Kim, J. Heo, and Y. Cho, "Shared-stack cooperative threads," Proc. of the 22nd Annual ACM Symposium on Applied Computing, 2007.

[10] A. Silberschatz, P.B. Galvin, and G. Gagne, Operating System Concepts, 6th Edition, Wiley-Interscience, 2003.

[11] J.L. Hill, System architecture for wireless sensor networks, Ph.D. thesis, 2003.

[12] G. Sachdeva, R. Domer, and P. Chou, "System modeling: A case study on a wireless sensor network," 2005.

[13] K.S. Trivedi, Probability and Statistics with Reliability, Queuing and Computer Science Applications, 2nd Edition, 2002.

구 본 칠

2004년 연세대학교 컴퓨터과학과 졸업(학사). 2004년~현재 서울대학교 컴퓨터공학부 박사과정(수료). 관심분야는 시스템 및 운영체제, 무선 센서 네트워크, 임베디드 시스템 및 미들웨어, 소프트웨어 라디오

허 준 영

1998년 서울대학교 컴퓨터공학과 졸업(학사). 2002년~현재 서울대학교 컴퓨터공학부 박사과정(수료). 관심분야는 시스템 및 운영체제, 무선 센서 네트워크, 결함 허용 시스템, 임베디드 시스템 보안

홍 지 만

2003년 서울대학교 컴퓨터공학부 졸업(박사). 2004년~2007년 광운대학교 컴퓨터공학부 조교수. 2007년~현재 숭실대학교 컴퓨터학부 조교수. 관심분야는 임베디드 운영체제, 결함 허용 컴퓨팅, 무선 센서 네트워크

조 유 근

1971년 서울대학교 졸업(학사). 1978년 미국 미네소타 대학교 컴퓨터 과학과 졸업(박사). 1985년 미국 미네소타 대학교 방문 교수. 2001년 한국 정보과학회 회장. 1979년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 운영체제, 알고리즘, 시스템 보안, 결함 허용 컴퓨팅