내장형 시스템을 위한 실시간 자바 쓰레드의 구현 및 성능 평가

강희구¶ 성민영∪ 김태현 이동렬 김소영 김광영 김형수 장래혁 신현식
¶(주)바로비전 서울대학교 컴퓨터공학과
hgkang@varovision.com {minyoung,thkim,lomen,uglyduck,corn,javaos}@cselab.snu.ac.kr
{naehyuck,shinhs}comp.snu.ac.kr

Implementation and Performance Evaluation of Real-Time Java Thread for Embedded Systems

Heegoo Kang¶ Minyoung Sung∪ Taehyoun Kim Dongryeol Lee Soyoung Kim
Kwangyoung Kim Hyungsoo Kim Naehyuck Chang Heonshik Shin
Dept. of Computer Engineering, Seoul National University

요 약

자바는 플랫폼 독립성, 보안, 멀티 쓰레드 지원, 가베지 콜렉터 등 많은 장점을 갖고 있는 언어이다. 하지만 현재의 자바 가상 기계(JVM)는 시간 제약 조건을 갖는 주기적인 태스크 수행, 실시간 스케쥴링 등 실시간 응용을 위한 지원이 미흡한 실정이다. 본 논문에서는 실시간 운영체제인 VxWorks와 내장형 JVM인 퍼스널 자바 상에서 실시간 응용 프로그램의 지원을 위한 자바 쓰레드를 설계, 구현한다. 이를 위해 기존의 자바 쓰레드에 시작시간, 주기, 종료시한 등의 시간 속성을 삽입하고, 실시간 자바 쓰레드 API 클래스를 구현한다. 실시간 스케쥴러를 세가지 방식으로 구현한다. 첫째는, JVM을 수정하지 않고 시간 속성에 따라 쓰레드에 우선 순위를 할당하여 스케쥴링하는 방식이고, 둘째는 JVM을 수정하여 시간 속성에 따라 VxWorks 커널 스케쥴러를 통해 스케쥴링하는 방식이며, 셋째는 독립적인 스케쥴링 태스크를 구현하는 방식이다. 쓰레드 생성, 문맥 교환 등의 오버헤드와 스케쥴링 시 종료시한 초과 비율 등의 기준을 통해 각각의 성능을 비교한 결과, VxWorks 커널 스케쥴러를 이용한 방식이 가장 적합하다는 결론을 얻을 수 있었다.

1. 서론

자바는 내장형 시스템의 응용 프로그램 작성에 사용되어 온 C, C++ 등의 언어를 사용할 때 발생하는 복잡성과 개발과정의 오버헤드를 감소시키기 위해 개발되었다 [1]. 자바는 플랫폼 독립성, 네트워크 이동성, 실행 코드의 재사용성, 보안 기능 등의 장점으로 인해 인터넷 응용 분야에서 각광 받고 있다. 그 중 주목할 만한 특징은 자바의 실행 모델이다. 자바 응용 프로그램은 플랫폼 독립적인 자바 바이트 코드(bytecode)로 구성되어 특정 플랫폼 상에서만이 아니라 자바 가상 기계(JVM; Java Virtual Machine)가 구현되어 있는 어떠한 플랫폼에서도 수행될 수 있다. 이러한 장점으로 인해 자바는 내장형 시스템(embedded system)을 위한 응용 개발용으로까지 그 영역을 확대해 나가고 있다. 썬 마이크로시스템즈에서는 이미 내장형 시스템을 위한 퍼스널 자바(PJAVA;PersonalJava) [2]를 발표하고, 셋톱박스나 PDA와 같은 내장형 시스템에 적용하고 있다.

그러나, 앞에서 언급한 여러 가지 장점에도 불구하고 일반적으로 내장형 시스템에서 요구하는 제약 조건을 만족시키지 못하는 한계 때문에 자바를 내장형 시스템 응용에 적용하는 것은 쉽지 않다. 일반적으로 내장형 시스템은 실시간 제약 조건을 가진다. 실시간 시스템은 계산의 논리적 정확성 뿐 아니라 계산 결과를 종료시한(deadline) 내에 도출해야 하는 제약조건을 가진다.

그러나, 현재의 JVM은 하드웨어 독립적인 바이트 코드를 기반으로 하여 플랫폼간의 호환성은 증진되었으나 플랫폼 종속적인 네이티브(native) 코드에 비하여 속도가 느리며, 낮은 쓰레드 동기화 성능, 가베지 콜렉터와 JIT 컴파일러에 의한 예측할 수 없는 지연 등 시간적 요구사항을 만족하기 어려운 문제들을 가지고 있다. 따라서, 현재로서는 자바를 경성 실시간 시스템(hard real-time system)보다는 연성 실시간 시스템(soft real-time system)에 적용하는 것이 타당하다. 본 논문에서는 대표적 RTOS인 VxWorks에 PJAVA를 이식하고, 실시간 자바 쓰레드의 구현 및 성능을 살펴본다. 자바 쓰레드에 시작시간(start time),주기(period), 종료시한(deadline) 등의 시간 속성을 추가하고, 세 가지 방식의 실시간 스케쥴러 구현을 제시한다. 첫째는, 자바 가상 머신을 수정하지 않고 시간 속성에 따라 자바 쓰레드에 우선순위를 할당하여 스케쥴하는 방식이고, 둘째는 JVM을 수정하여 시간속성에 따라 OS 커널 스케쥴러를 통해 스케쥴하는 방식이며, 셋째는 스케쥴러 태스크를 생성하고 이를 이용해 자바 쓰레드를 스케쥴하는 방식이다. 또한 종료시한을 넘긴 경우를 처리하기 위

한 핸들러(deadline handler)를 구현하여 멀티미디어 응용을 지원하도록 한다. 마지막으로, 실험을 통해 쓰레드 생성 시간과 종료시한 초과비율 등을 성능 척도로 하여 세가지 구현방법을 비교한다.

2. 실시간 자바 쓰레드의 설계 및 구현

2.1 구현 환경

실시간 자바 쓰레드의 구현 환경은 다음과 같다. 타겟 보드로는 16MB 메모리와 이더넷 컨트롤러를 갖춘 50 MHz MPC860 CPU 보드를 사용하였다. 운영체제로는 Windriver사의 VxWorks 5.3.1을 사용했고 개발 툴로는 Tornado 1.0 for WindowsNT를 사용하였다. 또한, 실험을 위해 소스가 공개된 PersonalJava 3.0을 타겟 보드에 이식하였다. 주요 이식 작업의 내용은 VxWorks의 태스크 제어함수를 이용하여 자바 쓰레드 제어 인터페이스를 구현하는 것과, VxWorks의 세마포어를 이용하여 자바 모니터(monitor)를 구현하는 것이다. 그 밖에, 메쏘드 시그너쳐에 대응하는 네이티브 호출 루틴 작성, CPU의 바이트 정렬(byte ordering) 고려 등의 작업이 수반되었다. VxWorks의 스케쥴 가능한 기본 단위는 태스크이며 자바 쓰레드는 결국 이 태스크로 사상되고, VxWorks에서 제공하는 스케쥴링 기법에 따라 태스크가 스케쥴된다.스케쥴링 기법은 선점형(preemptive) 우선순위 기법을 사용하며 같은 우선순위 태스크는 라운드로빈 방식(round-robin)으로 스케쥴된다.

2.2 실시간 자바 쓰레드 API의 설계

실시간 자바 쓰레드 클래스 RTThread는 기존의 자바 클래스를 상속받고, 그 외에 시작시간, 주기, 종료시한 등 시간 속성, 이를 접근하는 함수, 스케쥴링 정책, 우선순위 할당을 위한 자료 구조 및 종료시한 핸들러 관련 함수들을 추가하여 구현하였다.

시간 속성 변수들은 자바에서 현재 시간을 반환하는 System.currentTimeMillis()함수에 맞추어, long타입으로 정의하였으며 1/1000초(ms)의 정확도를 갖는다. 다음은 RTThread 클래스에 정의된 메쏘드들이다.

- RTThread.RTThread():클래스 생성자. 기존의 자바 쓰레드 생성자에 시간 속성 인자를 추가했다.
- RTThread.setAttr(start, period, deadline):시작시간, 주기, 종료시한을 설정.
- RTThread.getStart():시작시간을 반환.
- RTThread.getPeriod():주기를 반환.
- RTThread.getDeadline():종료시한을 반환.
- RTThread.setPolicy(int policy):스케쥴링 정책결정.
- RTThread.setRTpriority():시간속성과 스케쥴링 정책에 따라 우선 순위를 할당.
- RTThread.setDeadlineHandler(name):핸들러 등록.

그림 1. RTThread 클래스의 주요 메쏘드

2.3 실시간 자바 쓰레드 스케쥴러의 구현

실시간 자바 쓰레드는 주기적으로 수행되어야 한다. 따라서 현 주기에서 수행된 쓰레드의 run()이 다음 주기의 시작 시간 이후에도 다시 수행되어야 한다. 주기적 쓰레드의 스케쥴링 정책으로 RM과 EDF가 있다 [3].

RM(Rate Monotonic) 스케쥴링은 주기가 적은 쓰레드가 높은 우선 순위를 갖는 방식이다. 실행시 우선 순위가 바뀌지 않는 정적 우선순위 스케쥴링이다.

EDF(Earliest Deadline First) 스케쥴링은 종료시한이 가까울수록 높은 우선순위를 할당하는 방식이다. 시간에 따라 우선순위가 변하는 동적 우선순위 스케쥴링이다. 쓰레드의 매 주기마다 우선순위가 바뀔 수 있기 때문에 스케쥴링 오버헤드가 비교적 크다.

● 자바로 작성된 스케쥴러에 의한 스케쥴링

자바로 스케쥴러를 작성할 경우 JVM을 통한 VxWorks 태스크에 대한 제어가 자바 쓰레드 API로 제한되기 때문에 실시간 자바 쓰레드의 정확한 스케쥴링이 어렵다. Thread.setPriority()가 쓰레드 객체 생성 후부터 쓰레드가 Runnable 상태가 되기 전까지만 호출될 수 있기 때문에 실행시 우선순위를 변경하는 것이 불가능하다. 따라서, RM의 구현은 가능하지만, EDF와 같이 동적 우선순위 스케쥴링 정책은 구현 불가능하다. 자바 쓰레드는 우선순위 레벨이 10이기 때문에, 쓰레드의 개수가 10개 이상인 경우 서로 다른 우선순위의 쓰레드임에도 불구하고, 같은 우선순위를 할당받는 일이 발생할 수 있다. 이는 스케쥴 가능성(schedulability)을 낮추는 요인이 된다. 그러나, JVM의 수정이 없으므로 플랫폼 독립성을 가질수 있다는 점과 쓰레드 생성이나 수행시 오버헤드를 줄일 수 있다는 장점이 있다. 알고리즘은 다음과 같다.

---

1. 쓰레드 생성시 쓰레드 리스트에 새로운 쓰레드 추가하고 주기에 따라 쓰레드 리스트를 정렬한다.
2. 쓰레드 시작 전에 쓰레드 리스트에 있는 쓰레드들의 주기에 따라 우선 순위 할당한다. 전체 쓰레드의 개수가 10개 이상일 경우 1부터 10까기 각각의 우선 순위마다 같은 개수의 쓰레드를 갖도록 한다.
3. 쓰레드 시작 후 현재 시간보다 시작시간이 늦을 경우 그 시간만큼 쓰레드를 지연시킨다.
4. 매 주기마다 3번을 반복한다.

● VxWorks 커널 스케쥴러를 이용한 스케쥴링

VxWorks는 태스크의 우선 순위를 동적으로 설정하는 함수인 taskPrioritySet()을 제공하므로 EDF 스케쥴링 정책을 지원할 수 있다. 그러나, EDF의 경우 자바 쓰레드가 수행 중이라도, 새로 생성된 쓰레드의 종료시한이 현재의 쓰레드보다 앞선다면 새로운 쓰레드가 바로 수행되어야 하지만, 모든 쓰레드의 시작 전에 우선 순위가 할당되는 것을 가정하고 있기 때문에 새로운 쓰레드가 수행되지 못한다는 단점이 있다. 쓰레드 시작을 위해서는 쓰레드 객체의 start()가 호출되어야 한다. start()에서 threadCreate()라는 JVM 내부 함수를 이용해, 쓰레드 내부 구조를 생성하고, VxWorks태스크를 생성한다.VxWorks에서는 taskSpawn()이라는 태스크 생성을 위한 함수가 있고, 인자로서 ThreadRTO() 함수의 주소를 준다. 이 함수에서 자바 쓰레드의 run()을 수행하게

된다. JVM에 추가 구현해야 할 부분은 다음과 같다. 우선 **start()** 호출시 자바 응용 프로그램에서 시간 속성과 우선순위, 스케쥴링 정책, 종료시한 핸들러 관련 자료를 JNI(Java Native Interface)를 통해 JVM의 쓰레드 자료구조에 저장한다. 다음, VxWorks 태스크 생성시 쓰레드 우선순위에 따라 태스크의 우선 순위를 정한다. 마지막으로 **ThreadRTO()**에 스케쥴링 정책을 구현한다.

RM의 구현은 앞의 알고리즘과 유사하다. 차이점은 자바 쓰레드의 우선 순위 레벨이 256인 VxWorks 태스크의 우선 순위로 바뀌었고, VxWorks의 타이머 라이브러리를 이용하여 종료시한 핸들러를 구현할 수 있다는 것이다. EDF 스케쥴링 정책은 RM 구현 위에 우선순위 조정 부분을 추가한다. **run()** 메쏘드 수행이 끝날 때마다, 쓰레드의 종료시한을 다음 주기의 종료시한으로 조정하고 전체 쓰레드 리스트의 우선 순위를 조정한다.

- **스케쥴러 태스크를 이용한 스케쥴링**

스케쥴러 태스크는 실시간 쓰레드보다 높은 우선 순위를 갖고, 실시간 쓰레드의 우선 순위에 모두 같은 값을 할당한다. 스케쥴러 태스크가 시간 속성에 따라 모든 실시간 자바 쓰레드의 스케쥴링을 수행하는 방법이다. 무한대의 우선순위 단계를 지원할 수 있다. 그러나, 문맥 교환(context switch)의 횟수가 많아진다는 단점이 있다. 하지만 실행 중인 쓰레드보다 높은 우선순위를 갖는 쓰레드에 의한 선점이 가능하도록 할 수 있다. 구현을 위해서 쓰레드 자료 구조를 새로 정의하고, 이 쓰레드 자료 구조의 리스트를 유지하도록 한다.

- **종료시한 핸들러**

종료시한 핸들러는 쓰레드의 수행 중에 종료시한을 넘긴 경우 실행되며, 핸들러의 수행이 끝난 후에 다시 쓰레드의 수행을 재개한다. 종료시한 핸들러는 **RTThread** 클래스의 **setDeadlineHandler()**를 이용해 사용자가 정의할 수 있다. 비디오 플레이어일 경우 종료시한을 넘긴 프레임을 스킵한다던가, 네트웍 응용일 경우 처리하는 패킷 수를 줄이는 일을 종료시한 핸들러를 이용해 할 수 있다. VxWorks의 **wdStart()**함수에 시간과 핸들러의 함수 포인터를 인자로 주면 주어진 시간 후에 핸들러가 수행된다. **run()** 메쏘드 뒤에 **wdCancel()**로 타이머를 취소함으로써, **run()** 메쏘드가 종료시한 내에 수행된 후에는 핸들러가 수행되는 것을 막을 수 있다.

3. 성능 평가

시간 측정을 위해 VxWorks의 **vxTimeBaseGet()**를 사용하였으며, 기본 시간 단위는 320ns 이다. 표 1에 실시간 자바 쓰레드의 생성시간, 시작 지연시간, 문맥 교환시간 등의 기본적인 성능을 세가지 스케쥴링 구현 방식에 따라 요약하였다. 쓰레드 생성시간은 **new Thread()**에 의한 쓰레드 객체 생성과 생성자의 수행 시간을 합한 것이다. 시작 지연시간은 쓰레드의 **start()**가 호출된 후 실제로 **run()**이 수행되기 전까지의 시간이다. 스케쥴러 태스크 방식은 스케쥴러의 작업이 더해져서 약 50% 이상의 시간이 증가하였다.

표 1. 스케쥴링 방식에 따른 기본 성능

	기존자바	자바스케쥴	커널스케쥴	스케쥴태스크
쓰레드생성시간	14.4 ms	17.8 ms	18.2 ms	15.1 ms
시작 지연시간	10.6 ms	10.6 ms	12.8 ms	16.2 ms
문맥 교환시간	82 us	82 us	82 us	220 us

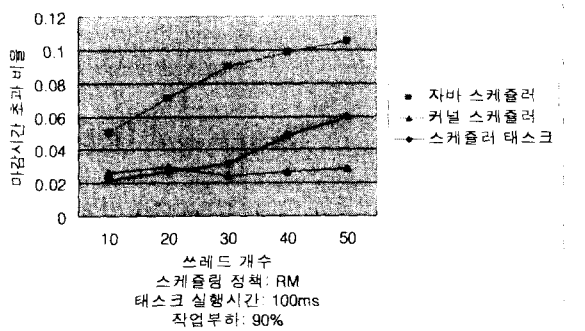

그림 2. 쓰레드 개수에 따른 종료시한 초과 비율

그림 2는 작업 부하가 100%로 고정되어 있을 때 쓰레드의 개수를 늘려감에 따른 종료시한 초과 비율을 보이고 있다. 자바 스케쥴러 방식은 다른 방식에 비해 종료시한 초과 비율이 크며, 쓰레드의 개수가 증가함에 따라 종료시한 초과 비율이 늘어나는 것을 볼 수 있다. 이는 자바 쓰레드의 우선 순위 레벨이 10가지이므로 쓰레드의 개수가 늘어남에 따른 스케쥴링의 부정확성으로 인한 것이다. 반면에 커널 스케쥴러 방식의 경우는 쓰레드 개수가 늘어남에도 불구하고 종료시한 초과 비율에 변화가 없는데 이는 쓰레드 개수에 비해 우선순위 레벨이 많기 때문이다. 스케쥴러 태스크 방식의 경우는 쓰레드 개수가 늘어남에 따라 스케쥴링 오버헤드가 증가하는 때문에 종료시한 초과 비율이 높아지게 된다.

4. 결론

본 논문에서는 실시간 응용 프로그램의 지원을 위해 자바 쓰레드를 확장한 실시간 자바 쓰레드 API를 정의하고, 스케쥴러를 세가지 방식으로 구현하였다. 각각의 방식에 대한 성능평가 결과 기본 연산시간이나 종료시한 초과 비율을 고려할 때 커널 스케쥴러를 통한 스케쥴링이 가장 적합하다는 결과를 얻을 수 있었다.그러나 실시간 자바 쓰레드의 지원만으로는 실시간 시스템을 위한 응용 프로그램 작성에 한계가 있다. 이 밖에 물리적 메모리 접근 허용, 우선순위 역전 문제의 해결 및 실시간 가베지 콜렉터 등이 지원되어야 할 것이다.

참고문헌

[1]Sun Microsystems Inc.,"The Java Language Environment: A White Paper," 1995.

[2]PersonalJava,http://java.sun.com/products/peronaljava/index.html.

[3]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, 1973.