ARX 실시간 운영체계를 위한 사용자 레벨 쓰레드

서양민○ 박정근 홍성수
서울대학교 전기공학부 실시간운영체계 연구실

User-Level Threads for the ARX Real-Time Operating System

Yangmin Seo Jungkeun Park Seongsoo Hong
Real-Time Operating Systems Laboratory
School of Electrical Engineering, Seoul National University, Seoul, 151-742, Korea

요 약

내장 실시간 시스템이 높은 우선순위의 비동기적 이벤트를 적시에 처리하려면 필수적으로 적은 비용의 선점 다중쓰레드를 지원해야 한다. 사용자 레벨 쓰레드는 커널 레벨 쓰레드 보다 적은 비용의 유연한 추상적 기법들을 제공하지만, 기존의 실시간 시스템에서는 스케쥴링과 시그날(signal) 처리가 단순하다는 이유로 커널 레벨 쓰레드가 선호되어 왔다. 본 논문에서는 내장 실시간 시스템에 적합한 새로운 사용자 레벨 다중쓰레드 방식을 제안한다. 이 기법은 가상 쓰레드(virtual threads)와 개선된 스케쥴링 이벤트 업콜(scheduling event upcall) 메카니즘을 기반으로 한다. 가상 쓰레드는 사용자 레벨 쓰레드에게 커널 레벨의 실행 환경을 제공할 수 있도록 사용자 레벨 쓰레드를 커널 레벨로 형상화한 것이다. 이 쓰레드는 필요에 의해 잠시동안 사용자 레벨 쓰레드에 묶이는 수동적인 존재이다. 스케쥴링 이벤트 업콜 메카니즘은 쓰레드 블록킹과 타이머 만기와 같은 커널 이벤트를 유저 프로세스에게 전달할 수 있게 한다. 본 논문의 개선된 업콜 방식은 scheduler activation과 시그날과 같은 전통적인 업콜 구조에서 예측하기 힘든 요소들을 배제했다. 순간적인 시스템의 과부하 상황에서도 이벤트를 놓치지 않으면서 커널과 유저 프로세스의 비싼 동기화 작업들을 피할 수 있도록 하는 잠금(lock)이 필요 없는 이벤트 큐를 사용한다. 본 기법은 서울대학교 실시간 운영체계 실험실에서 구현한 ARX위에서 완벽하게 구현되었다. ARX 사용자 레벨 쓰레드가 사용자 레벨 쓰레드의 장점을 손상하지 않으면서 솔라리스와 윈도즈 98과 같은 상용 운영체제의 커널 쓰레드보다 성능이 우수함을 실험적 비교에 의해서 입증한다.

1. 개요

전통적으로 프로그램 작성에 있어, 사용자 레벨 쓰레드가 두 가지 이유로 커널 레벨 쓰레드보다 선호되어 왔다[1]: (1)커널 레벨 쓰레드는 응용 프로그램을 커널이 제공하는 한가지 쓰레드의 구현 방식으로 제한되어야 하기 때문에, 프로그래밍 환경에 잘못된 추상화 기법을 제공할 여지가 있다; 그리고 (2)커널 레벨 쓰레드는 사용자 레벨 쓰레드보다 오버헤드가 더 크다. 한편, 실시간 프로그래밍에서는, 커널 레벨 쓰레드의 단점에도 불구하고 자주 구현 모델로 선정되었다. 커널 레벨 구현은 쓰레드 스케쥴링과 실시간 처리에서 중요한 시그날 처리를 단순화 할 수 있다. 커널의 지원이 없으면, 사용자 레벨 쓰레드는 커널 레벨 쓰레드 블록킹과 시그날 전달를 처리할 수 없기 때문이다. 따라서, 한 사용자 레벨 쓰레드가 블록킹 시스템 콜을 경유하여 커널 모드에서 블록되면, 그 쓰레드의를 포함하는 프로세스는 실행 가능한 상태에 있는 다른 쓰레드들과 함께 블록된다. 이 문제를 극복할 목적으로, scheduler activation [1]과 first-class user-level thread [4]와 같은 몇가지 방법이 제안되었다. 그러나 이들은 주로 다중 프로세서에 중점을 두었고, 실시간 시스템 문제를 정확히 언급하고 있지 않기 때문에, 내장 실시간 시스템에는 부적절하다. 예를 들어, [1]에서 각 이벤트를 알리려면 유저 레벨 스택이 업콜 인자를 운반해야 하므로, 값비싼 scheduler activation이 필요하게 된다. 그러므로 이벤트 처리 중에 또다른 이벤트가 발생하면, 커널은 다른 scheduler activation을 만들고 현재 scheduler activation을 인터럽트시키는 업콜을 해야한다. 이 방식은 다중 프로세서 시스템에서 요구되지만, 단일 프로세서 내장 시스템에 대해서는 너무 큰 오버헤드를 유발한다. 따라서 대부분의 실시간 운영체제 시스템은 오직 커널 레벨 쓰레드만을 지원하고 있다[3, 5].

이를 해결하기 위해 본 논문에서는 가상 쓰레드(virtual thread)와 스케쥴링 이벤트 업콜(scheduling event upcall)을 제안한다. 가상 쓰레드는 사용자 레벨 쓰레드에게 커널 레벨의 실행 환경을 제공하기 위해 사용자 레벨 쓰레드를 커널 레벨로 형상화한 것이다. 이 가상 쓰레드는 필요에 의해 잠시동안 사용자 레벨 쓰레드에 묶이는 수동적인 존재이다. 커널은 결코 이 쓰레드를 스케쥴하지 않는다. 스케쥴링 이벤트 업콜 메카니즘은 쓰레드 블록킹/언블록킹(blocking/unblocking), 타이머 만기와 같은 커널 이벤트를 유저 프로세스에게 전달할 수 있게 한다. 따라서 커널이 커널 쓰레드를 스케쥴하듯이, 실시간 응용 프로그램이 자신의 쓰레드를 선점적으로 스케쥴할 수 있다.

2. 사용자 레벨 다중 쓰레딩

2.1 가상 쓰레드

가상 쓰레드는 커널 쓰레드와 유사한 역할을 한다. 이 쓰레드는 사용자 쓰레드에게 커널 스택을 제공하고 커널에게는 현재 실행중인 또는 커널 모드에서 블록된 사용자 쓰레드를 구별할 수 있는 식별자를 제공한다. 그러나 커널 쓰레드와 달리 동적으로 생성되는 수동적인 객체이다. 즉, 가상 쓰레드는 사용자 쓰레드의 커널 레벨 형상화(incarnation)에 불과하다. 이러한 형상화는 사용자 쓰레드가 커널로 들어갈 때 하나의 가상 쓰레드에 묶이면서 만들어진다. 가상 쓰레드는 수동적인 존재이기 때문에 사용자 쓰레드를 멀티플렉싱하거나 커널에 의해서 스케쥴링되지 않는다. 그림 1은 가상 쓰레드의 커널 데이터 구조체를 보여준다. 커널은 사용자 쓰레드를 오직 그와 묶인 가상 쓰레드에 의해서 구별하게 되며, 사용자 레벨의 쓰레드 스케쥴링에 영향을 줄 수 없다. 결과적으로 사용자 레벨 쓰레드 스케쥴러는 스케쥴링 이벤트 업콜 메커니즘의 도움으로 쓰레드 스케쥴링의 전권을 행사하게 된다.

2.2 스케쥴링 이벤트 업콜

응용 프로그램이 사용자 레벨에서 쓰레드를 제어하기 위해서는 쓰레드 블록킹과 타임아웃과 같은 커널 이벤트에 대해 적절히 반응해야 한다. 본 논문에서는 커널 이벤트를 효율적으로 전달하기 위한 방법으로 스케쥴링 이벤트 업콜(scheduling event upcall)

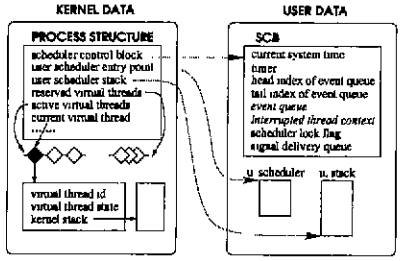

그림 1: 다중쓰레드를 위한 커널/유저 자료구조

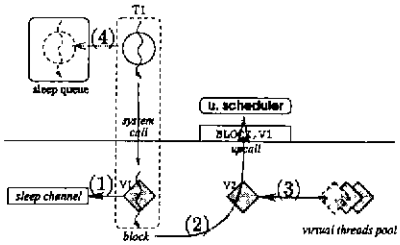

그림 2: BLOCK 스케줄링 이벤트 업콜의 예

방식을 제안한다. 스케줄링 이벤트가 발생할 때, 커널은 업콜을 통하여 사용자 레벨 스케줄러에게 알려준다. 스택을 통해 인자를 넘겨주는 전형적인 함수의 호출과는 달리 업콜의 인자는 커널과 유저가 공유하는 스케줄러 제어 블럭(scheduler control block: SCB)의 이벤트 큐를 통하여 전달된다. SCB는 사용자의 주소공간에 위치하고 있으며, 이벤트를 담기위한 충분한 공간을 갖고 있기 때문에 과부하 상태에서 발생하는 이벤트를 잃어버리지 않는다. 그림 1은 SCB에 포함된 객체들을 보여준다.

이벤트가 발생하면, 커널은 먼저 현재 쓰레드의 문맥(context)을 저장한다. 이는 다음과 같이 쓰레드의 수행모드에 따라 다르게 처리된다.

- 커널 모드: 커널은 가상 쓰레드를 생성하여 (가상 쓰레드의 풀에서 얻거나, 새로 생성한다) 프로세스의 현재 가상 쓰레드로 만든다.
- 유저 모드: 커널은 유저 문맥을 유저 스택위에 복사하고, 레지스터 파일 안의 스택 포인터를 조정하고 이를 SCB의 "interrupted thread context"필드에 저장한다.

문맥을 저장한 후, 커널은 이벤트를 구성하고 업콜 인자로 이를 채워 이벤트 큐에 넣고 사용자 스케줄러를 호출한다.

그림 2는 쓰레드가 I/O 이벤트를 기다리면서 시스템 콜에서 블록할 때 BLOCK 이벤트가 전달되는 예를 보이고 있다.

2.3 이벤트의 종류와 업콜 인자

- BLOCK: 쓰레드가 시스템 콜에서 블록할 때 커널은 BLOCK이벤트를 즉시 업콜한다. 이 이벤트의 인자는 블록된 가상 쓰레드의 식별자와 블록의 이유이다. 만약 블록의 이유가 커널 내에서의 동기화때문이라면 lock을 갖고 있는 가상 쓰레드의 식별자를 넘겨준다.
- WAKEUP: 블록된 쓰레드가 커널에서 깨어날 때 관련 가상 쓰레드의 식별자와 함께 WAKEUP이 전달된다.
- TIMER: 사용자 레벨 스케줄러가 미리 설정한 타이머가 만기되었을 때 전달되며, 타이머의 지연값이 함께 전달된다.
- SIGNAL: 시그날을 프로세스에 넘겨주어야 할 때, 커널은 프로세스 구조체의 siginfo 구조체 안에 담긴 시그날 정보와 함께 SIGNAL 이벤트를 업콜한다.
- EXIT: 프로세스가 exit(), exec()와 fork()를 호출한 경우, 커널은 이 이벤트를 사용자 레벨 스케줄러에게 보낸다.
- RESUME: 선점 후에 프로세스가 재시작될 때, 커널은 선점 지속기간과 함께 RESUME 업콜을 사용자 레벨 스케줄러에게 보낸다. 쓰레드 실행 시간을 추적하기 위해 사용자 레벨 스케줄러가 이 이벤트를 사용한다.

각 이벤트들에 대한 자세한 내용은 [7]을 참고하길 바란다.

2.4 커널/유저 동기화

사용자 레벨 스케줄러와 커널은 공유하는 이벤트 큐를 두고 서로 경쟁하게 된다. 이 동기화 문제의 단순한 해결책은 커널이 지원하는 잠금 기능(locking primitive)을 사용하는 것이다. 그러나 이 방식은 사용자 레벨 스케줄러가 실행 될 때마다 시스템 콜을 해야하기 때문에 매우 비싸다. 대안으로 커널과 유저 프로세스 사이의 잠금이 필요 없는 동기화 방식을 사용한다. 이를 위해 다음 방식으로 사용자 레벨 스케줄러를 특별 설계한다.

사용자 레벨 스케줄러를 두 개의 영역으로 나눈다. 상위 영역에서는 이벤트 큐를 비울 때까지 이벤트를 모조리 소비하고, 하위 영역에서 다음으로 실행될 쓰레드를 선택한다. 하위 영역의 실행은 안전하게 중지될 수 있다. 이를 가능하게 만들기 위해, 사용자 레벨 스케줄러는 자기 자신만의 유저 스택을 갖고 있지만, 이의 다른 문맥 저장 영역이나 쓰레드 제어 블록을 가지지 않는다.

유저 프로세스의 SCB의 lock 필드는 사용자 레벨 스케줄러가 재진입이 불가한 코드(non-reentrant code)를 수행할 때 또는 문맥 교환을 방지하기를 원할 때 세트된다. 이벤트가 발생하면, 유저 프로세스의 상태와 무관하게 커널은 이벤트를 이벤트 큐에 넣을 수 있다. 커널은 업콜을 마치기 전에 사용자 스케줄러가 실행될 필요가 있는지를 점검한다. lock 필드의 값이 0이고 업콜 시점에 이벤트 큐가 비어있었을 경우에만 사용자 레벨 스케줄러를 실행한다. 큐가 비지 않았다는 사실은 사용자 레벨 스케줄러가 큐에서 이벤트를 꺼내는 동안 커널의 업콜 핸들러로 인해 인터럽트되었음을 의미한다. 따라서 이러한 경우에는 사용자 레벨 스케줄러를 실행하지 않는다. 만약 업콜 시점에 이벤트 큐가 비어있었으면, 사용자 레벨 쓰레드의 실행중이거나, 또는 사용자 레벨 스케줄러가 하위 영역을 실행하고 있었음을 암시한다. 어느 경우든지, 커널은 새로운 사용자 레벨 스케줄러를 실행한다. 실행중인 사용자 레벨 스케줄러가 있었다면, 스케줄러는 같은 스택을 사용하기 때문에, 새로운 스케줄러가 예전 스케줄러를 지운다. 사용자 레벨 스케줄러를 안전하게 중단되도록 설계했기 때문에, 이러한 과정은 완벽히 안전하게 동작한다. 사용자 레벨 스케줄러의 예제는 [7]을 참고하기 바란다.

2.5 이벤트 큐의 크기

실시간 시스템에서, 중대한 이벤트에 적시에 반응하는 일은 매우 중요하다. 그렇지 않으면, 시간적 결함이 발생해서 시스템에 심각한 피해를 입힐 수 있다. 전달된 이벤트에 적시에 반응하는 일은 응용 프로그램 작성자의 책임이지만, 중대한 이벤트를 놓치지 않는 일은 실시간 커널의 몫이다. 본 논문의 이벤트 큐는 제한된 크기의 원형 큐로 구현되기 때문에, 시스템이 이벤트가 폭주하는 순간적인 오버로드를 경험할 때에는 이벤트를 잃어버릴 수 있다. 다행히도, 이벤트 큐에 동시에 놓일 수 있는 이벤트의 최대 개수를 분석할 수 있다. 따라서, 다음 분석을 통해 큐의 크기를 제한할 수 있다.

- BLOCK과 WAKEUP: 이 이벤트들은 사용자 레벨 쓰레드가 호출한 시스템 콜의 수행 도중에 발생된다. 따라서 이러한 종류의 이벤트의 총 개수는 한 프로세스 내에서 블록할 가능성이 있는 쓰레드의 개수를 넘지 않는다. 단, 예외적으로 사용자 레벨 스케줄러가 BLOCK 이벤트를 꺼내기 전에, 같은 쓰레드의 WAKEUP 이벤트가 그 이벤트(BLOCK 이벤트)의 바로 뒤에 있을 경우도 있다. 한 프로세스 내의 블록 당할 수 있는 쓰레드의 개수를 b라고 하자. 이런 종류의 이벤트는 많아야 $b+1$개가 동시에 큐에 존재할 수 있다.

- **TIMER**: 이 이벤트는 SCB의 timer 필드가 적절하게 세트되어 있을 때에만 전달된다. 커널은 **TIMER** 이벤트를 전달할 때 이 필드를 리셋하기 때문에, 사용자 레벨 스케줄러가 다시 한번 필드를 세트하지 않는 한 또 다른 **TIMER** 이벤트는 전달될 수 없다. 그러나, 사용자 레벨 스케줄러는 이벤트 큐에서 6TIMER 이벤트를 제거하기 전에 timer 필드를 세트하기 때문에, 동시에 두 개의 **TIMER** 이벤트가 큐에 있을 수 있다.
- **SIGNAL**: 오직 현재 실행중인 쓰레드만이 이 이벤트를 만들 수 있기 때문에 많아야 한 개의 동기적 **SIGNAL** 이벤트가 큐에 존재한다. 한편, 외부 인터럽트 소스에서 발생한 비동기적 **SIGNAL** 이벤트는 여러 개가 있을 수 있다. i를 전용 인터럽트 핸들러 쓰레드의 개수라고 하면, 큐에 존재하는 비동기적 **SIGNAL** 이벤트의 개수는 $i+1$을 넘지 않는다. 그리고, **TIMER** 이벤트와 같은 이유로 1이 추가된다.

분석 결과와 여분의 큐 헤더를 합해서 큐 크기의 한계 값을 구하면 다음과 같다.

$$b+i+5$$

SIGNAL 이벤트는 충분히 많은 수의 미처리 인터럽트를 담을 수 있으며, 큐 크기는 분석한 바와 같이 한계값을 가지므로, 커널이 이벤트를 잃어버리지 않는다는 보장을 할 수 있다. 이렇게 결정된 이벤트의 크기는 사용자가 쓰레드 시스템이 초기화되기 전에 **thread_mtparam_size()**를 이용하여 시스템에 알려주면 되며, 쓰레드 라이브러리는 쓰레드 시스템이 초기화 될 때 **register_uscheduler()**를 통해서 커널에게 알려준다.

3. ARX 실시간 운영체계

본 논문에서 설명된 사용자 레벨 쓰레드 커널 메카니즘을 ARX 실시간 운영체계에서 구현했다. ARX는 완전히 선점형인 실시간 커널을 가지고 있는 독립적인 운영체계이고, POSIX 호환인 쓰레드 라이브러리와 다중 쓰레드 작업에 대해 안전한 표준 I/O 라이브러리를 가지고 있다. ARX는 현재 Pentium PC와 StrongARM NC위에서 동작한다. ARX는 VFAT 파일 시스템, TCP/IP 프로토콜, X11R5 윈도우 시스템을 지원한다. ARX는 유연한 2단계 스케줄링을 지원하고, 완벽한 다중쓰레딩, 그리고 효율적인 사용자레벨 I/O[6]를 지원하도록 설계 되었다. 특히 ARX에서의 스케줄링은 커널이 프로세스를 스케줄링하고, 사용자 레벨 스케줄러가 프로세스의 쓰레드를 스케줄하는 구조로 되어있다. 커널 스케줄러는 CPU 대역폭의 사용량에 따라서 weighted fair queuing 알고리즘이나 또는 그 변종으로 프로세스를 스케줄하게 된다[2].

4. 성능 평가

ARX 실시간 운영체계에 구현된 제안된 기법들의 성능을 확인하기 위해서 광범위한 실험과 측정을 하였으나, 본 논문에서는 지면 관계상 ARX의 기본 성능과 업콜의 성능만을 보이도록 한다. 세 가지 다른 운영체계, ARX, Solaris, Windows 98에서 다중쓰레드 자바 프로그램을 수행하였을 때의 쓰레드 성능을 측정한 결과는 [7]을 참고하기 바란다.

실험은 Intel 100MHz Pentium 프로세서와 256KB의 이차 캐쉬를 갖는 기계에서 이루어졌다. 측정은 Intel Pentium 프로세서의 64-bit cycle 카운터를 사용하여 이루어졌다.

표 1은 ARX와 QNX의 기본 성능 지수를 보여준다. ARX에서 보여지는 $8.09\mu s$의 보통의 스케줄링 지연은 QNX보다 훨씬 작은 값이다. 시스템 서비스의 오버헤드의 차이를 확인하기 위해서 **clock_gettime()** 시스템 콜을 사용하여 시간을 측정하였다. 이 결과 ARX가 QNX보다 6배 빠르게 동작함을 알 수 있다. 이런 큰 차이가 나는 것은 QNX가 마이크로커널 구조를 사용하기 때문이다. ARX는 클럭 인터럽트 핸들링에서도 QNX 보다 좋은 성능을 보여준다.

ARX 커널을 업콜에 상당히 의존하기 때문에 업콜 지연을 최소화하는 것이 매우 중요하다. 그림 3 round-robin 정책을 사용하고 고정 우선순위 스케줄링을 하는 사용자 레벨 스케줄러 구현했을 때 ARX 커널에서의 업콜 성능을 나타낸다. 업콜 지연은 커널 업콜 핸들러가 시작하고 사용자 레벨 스케줄러가 수행되

	scheduling latency	clock intr handling	nullsys syscall	clock_gettime syscall
ARX	8.09	2.88	2.15	4.06
QNX	10.1	n/a	n/a	24.01

(unit: μs)

표 1: ARX와 QNX의 성능 비교

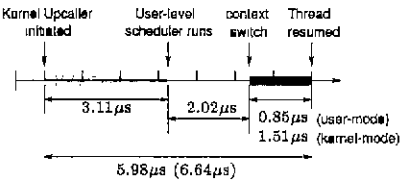

그림 3: 업콜의 성능

기까지의 시간으로 정의되고, 그 값은 $3.11\mu s$로 측정되었다. 그림 3에서 보듯이 쓰레드를 다시 수행시키기 위해서 필요한 시간은 쓰레드가 커널 모드에 또는 사용자 모드에 있는지의 여부에 크게 영향을 받지 않는다. 이것은 **resume()** 시스템 콜이 일반적인 시그날 검사 기능을 생략하도록 만든 특수한 trap handler에 의해서 다루어지도록 설계하였기 때문이다.

5. 결론

이 논문에서는 내장 실시간 시스템을 위한 효율적인 사용자레벨 다중쓰레드 기법을 제안하였다. 이 기법은 동적 가상 쓰레드 바인딩과 스케줄링 이벤트 업콜 메카니즘으로 구성되어 있다. ARX 실시간 운영체계는 이런 모든 기법을 구현했고, ARX 위에서 광범위한 측정을 하였다. 실험결과에 따르면 커널 레벨 쓰레드의 장점을 유지하면서도 Solaris나 Windows 98과 같은 상업용 운영체계의 커널 쓰레드보다 제안된 기법이 훨씬 적은 부하를 가짐을 알 수 있다.

참고 문헌

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM SOSP*, pages 95–109, October 1991.

[2] A. Dermers, S. Keshav, and S. Shenkar. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research & Experience*, pages 3–12, October 1990.

[3] M. Humphrey, G. Wallace, and J. A. Stankovic. Kernel-level threads for dynamic, hard real-time environment. In *Proceedings of the Real-Time Systems Symposium*, pages 38–48, December 1995.

[4] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM SOSP*, pages 110–121, October 1991.

[5] K. Schwan, H. Zhou, and A. Gheith. Multiprocessor real-time threads. *Operating Systems Review*, 26(1):54–65, January 1992.

[6] Y. Seo, J. Park, and S. Hong. Efficient user-level I/O in the ARX real-time operating sys tems. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers , and Tools for Embedded Systems*, June 1998.

[7] Y. Seo, J. Park, and S. Hong. Supporting preemptive user-level threads for embedded real-time systems. Technical Report SNU-EE-TR-1998-1, School of Electrical Engineering, Seoul National University, August 1998.