Linux 기반의 실시간 시스템 지원 Middleware를 위한 Timer Handling 기법

박호준°, 김문회, 이창훈
건국대학교 컴퓨터 정보통신 공학과
{hjpark, mhkim, chlee}@kkucc.konkuk.ac.kr

Timer Handling Technique for Real-Time Middleware on Linux

Ho-Joon Park°, Moon Hae Kim, Chang-Hoon Lee
School of Computer Science and Engineering
Konkuk University, Seoul 143-701, Korea

요 약

리눅스 컴퓨터 시스템에 실시간 응용프로그램들을 실행할 경우, 실시간 응용프로그램 자체의 내부나 하부에 실시간 시스템 지원을 위한 엔진을 내장하고 있어야 한다. 그러나 기존의 리눅스 시스템들에 실시간 응용프로그램을 적용하기 위한 실시간성을 보장 받기 위해서는 스케줄러의 실시간성 빈약 등으로 인해 운영체제의 수정이 불가피한 상황이다. 또한 운영체제의 수정은 호환성의 결여라는 문제를 야기시킨다. 이를 위해 사용하는 방식이 Middleware 방식이다. 실시간 시스템 지원 Middleware 는 운영체제와 실시간 지원 응용프로그램 사이에 위치하여 운영체제 자체가 가지고 있는 부족한 실시간성을 보완하여 주고, 응용프로그램이 적절하고 효과적으로 시스템 서비스들을 이용할 수 있게 해준다. 본 논문에서는 Middleware 가 부족한 실시간성을 보완하기 위하여 사용할 수 있는 Middleware 스케줄러의 방식에 대하여 논하고, 효율적인 설계방식을 제안한다.

1. 서 론

리눅스에서 실시간 응용 프로그램들을 실행시키기 위한 실시간성을 보장하기 위해서는 운영체제의 핵심 요소인 커널(kernel)에서 이를 지원하여야 한다. 그러나 기존의 커널은 이러한 실시간 지원에 대하여 기능이 준비되지 않거나 실시간 지원이 미약하게 설계되어 바로 적용하기에는 많은 어려움이 있다.

특히 실시간 시스템의 가장 중요한 요소인 실시간성에 대하여 기존의 리눅스 커널은 완벽한 선점성의 미확보로 인해 현재의 실시간 시스템을 지원하기 위해서는 기존의 커널의 재작성이나 수정이 불가피한 상황이다.

그러나 커널을 수정하거나 재작성을 하게 되면 기존의 리눅스에서 개발된 여러 지원 소프트웨어들이나 내부 시스템 서비스들, 지원 라이브러리들과의 호환성이 결여되어 실시간 응용프로그램들이 이러한 서비스들을 이용해야 하는 경우 시스템 전체를 재작성하게 되는 자원 낭비를 초래하게 된다.

이를 위하여 Middleware 방식을 채택하는 것이 바람직하다. Middleware 방식은 기존의 커널과 실시간 태스크들 사이에서 기존 커널이 가질 수 없었던 부족한 실시간성을 보완하여 주고, 실시간 태스크들이 효과적으로 시스템 서비스들을 사용할 수 있도록 중간 매개체 역할을 하는 부분이다.

실시간 태스크들을 처리하기 위해 Middleware 는 스케줄링 지원, 네트워크 지원, I/O 지원 등으로 그 임무가 나뉘는데, 특히 실시간성 확보를 위해 Middleware 내부의 스케줄러의 역할은 중요하다. 이러한 Middleware 내부의 커널 스케줄러를 도와 실시간 태스크들의 스케줄링 임무를 하는 요소를 Middleware 스케줄러라 한다.

Middleware 스케줄러에서 주로 실시간성을 확보하고 기존 커널의 부족한 선점성을 확보하는데 사용되는 것은 우선순위 조절과 Signal 처리 방식이다. 본 논문은 Middleware 방식에서 사용되고 있는 우선순위 조절과 Signal 처리 방식의 문제점과 효율적인 설계에 대하여 기술한다.

2 장에서는 Middleware 방식 실시간 시스템의 구조와 기존 Middleware Timer Handler 의 처리 방식과 그에 따른 문제점 등을 논하고, 3 장에서는 효율적인 Middleware Timer Handler 의 처리방식에 대하여 논한다.

2. 실시간 시스템 지원 Middleware

2.1 Middleware 방식의 실시간 시스템 구조

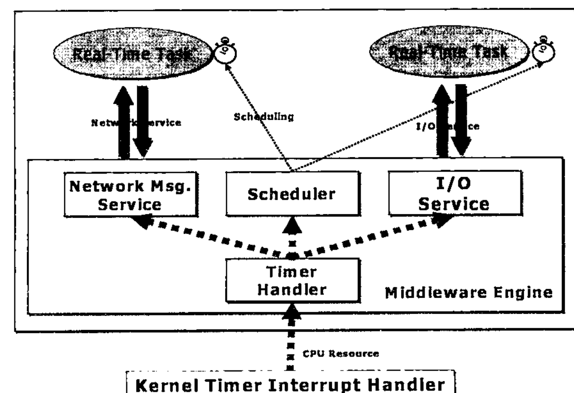

Figure 1. 실시간 시스템 지원 Middleware의 구조

<그림 1>은 실시간 시스템 지원 Middleware 의 일반적

인 구조이다.

실시간 응용프로그램에서 Middleware 는 실시간 태스크들의 하단부에 위치하여 태스크들과 커널의 중간에서 각종 서비스들을 지원해주는 중간 매개체 역할의 임무를 수행한다. 이렇게 구성된 Middleware 는 리눅스의 스레드 시스템의 장점들을 활용하기 위해 하나의 프로세스 내에서 라이브러리 형식으로 상위의 실시간 태스크들과 결합된다.

또한 경우에 따라서는, 상단부의 각 실시간 태스크들은 오브젝트 형식으로 관리되기도 한다.

Middleware 의 각 부분은 실시간 지원 시스템의 엔진 역할을 수행하는 부분으로서 다음과 같은 임무를 가진다.

(1) 실시간 태스크들의 스케줄링 부분
(2) 각 태스크들의 네트워크 메시지 관련 지원 부분
(3) 각 태스크들의 I/O 작업 관련 지원 부분

이러한 임무들은 실시간 응용프로그램 내에서 모두 동기적으로 작동하게 되며, 실시간 태스크들에 대한 서비스를 수행한다.

이중, Middleware 내부에서 실시간 태스크들의 스케줄링에 관여하는 것은 커널 스케줄러의 부족한 실시간성 때문이다. 첫번째 이유로, 실시간 태스크들은 모두 실시간성[5]을 가져야 하지만 리눅스 커널 내의 스케줄러는 이를 지원하고 있지 않다. 두번째 이유로는, 실시간 지원 태스크들의 CPU 자원 사용이 초과될 경우에는 이를 강제적으로 환원시켜야 하지만 리눅스 커널의 스케줄러는 선점성 확보[1]라는 측면에서 이 기능이 미약하다.

2.2 실시간 지원 시스템 Middleware 스케줄러

실시간 지원 Middleware 스케줄러는 리눅스 커널 스케줄러가 지원해 주지 못하는 실시간성에 대해 지원해 주는 부분이다. 여기에는 다음과 같은 요소들을 포함하고 있다.

(1) 현재 실행이 되어야 할 태스크들을 CPU 자원을 할당 받게 한다.
(2) 작업 시간이 초과된 실시간 태스크들은 태스크 자체의 의지 없이도 강제로 CPU 자원을 회수한다.
(3) Middleware 의 기타 서비스들에서 요청된 실시간 태스크들의 실행을 보장한다.

이런 항목들을 실행시키기 위해 Middleware 스케줄러는 다른 실시간 태스크들보다도 먼저 CPU 자원을 할당 받아야 한다. 또한 커널 스케줄러에는 하드웨어 Timer Interrupt Handler 에서 설정된 시간 - 10msec(1/100 초) - 값[1]이 있다. 이렇게 설정된 값을 Time Slice[5] 또는 Time Quantum[1] 이라 한다. 따라서 실시간 지원 Middleware 스케줄러는 매 Time Quantum 때 마다 CPU 자원을 할당 받아 상위의 실시간 태스크들의 스케줄링 임무를 수행한다. 이런 조건을 만족시키기 위해 Middleware 스케줄러는 다음과 같이

(1) Middleware 스케줄러는 SCHED_RR 스케줄링 정책으로 초기화되고, 또한 유지되어야 한다.
(2) 리눅스 시스템 내의 어떤 실시간 태스크들보다도 가장 우선순위가 높아야 한다.
(3) 커널 스케줄러로부터 명백하게 CPU 자원을 할당 받기 위해 Time Quantum 의 주기와 맞는 시간을 Timer 로 설정하여 매 Time Quantum 때마다 Signal 을 받도록 한다.

이중에서 항목 (3)의 Signal 을 사용하게 되면 한 프로세스내의 모든 태스크들은 중지되고 특정 Signal 을 사용한 태스크만이 CPU 자원을 할당 받게 되므로 Middleware 스케줄러가 실시간성을 보장하는 선점성을 확보하는데 꼭 필요한 요소이다.

2.3 Middleware 스케줄러 내부의 Middleware Timer Handler

Middleware 스케줄러는 다시 두 부분으로 나눌 수 있다. 하단부는 커널로부터 CPU 자원을 확보하는데 필요한 부분이고, 상단부는 실시간 태스크들의 스케줄링을 관리하는 부분이다. 특히 하단부는 실시간 태스크들의 실시간성과 선점성을 확보해 주는 중요한 부분으로서 이를 Middleware Timer Handler 라 하고, <그림 1>에서 본 바와 같이 커널 내부의 Hardware Timer Interrupt Handler 의 스케줄링 작업으로 인한 CPU 자원 획득으로 실행된다.

3. Middleware Timer Handler 의 설계

3.1 고려사항

Middleware Timer Handler 를 설계하기 위한 처리 방식으로는 Signal Handler 를 사용하는 방법과 Middleware Timer Handler 스레드를 따로 두는 방법으로 구분할 수 있다. 이 2 가지 방법 모두 다 Timer 로서의 Signal 기법을 사용하여야 한다. Signal 기법을 사용하는데 있어서는 다음과 같은 단점들을 가지고 있으므로 주의해야 한다.

(1) 실시간 태스크들을 포함하고 있는 리눅스 상의 프로세스가 Signal 을 받으면 기본 동작(Default Action)을 취하게 된다. 대부분의 기본 동작은 '종료'이다. - 기본동작을 바꿀 수 있는 처리가 필요하다. 이러한 행동을 'Signal Handler 를 부착한다'[2]고 한다.
(2) 한 개의 프로세스 내에 다수의 스레드들이 존재 할 경우, 어떤 스레드가 배달된 Signal 을 처리할 지 불분명해진다.
(3) Signal 이 특정 태스크에 배달되는 주기가 짧아 질수록 Signal Handler 의 행동이 불분명해 질 수 있다. - Signal 이 배달되는 주기가 극단적으로 시스템의 Timer Quantum 값에 가깝게 되면 Signal Handler 가 처리하지 못하는 빈도가 매우 증가하게 된다. 이는 리눅스 커널 스케줄러의 문제로 Signal 처리 루틴이 커널의 Timer Interrupt 에 의하여 강제로 CPU 자원을 뺏긴 다음 CPU 자원을 할당 받을 때 다른 Signal 이 배달되었을 경우 해당 Signal 이 Block 되거나, 무시되기 때문이다.

이러한 Signal 처리의 문제점들을 해결하기 위해서는 가급적 Timer Handler 의 실행코드를 단순화 시켜야 하며, Middleware Timer Handler 의 Signal 처리 코드 부분을 독자적인 스레드로 따로 분리하여 다루어야 한다.

3.2 Signal Handler 를 사용한 Middleware Timer Handler 설계

POSIX.1 의 권고안에 의한 Signal 처리[2]는 기존의 실시간 시스템을 포함한 모든 형태의 Signal 처리라는 점에서 전통적인 Signal Handling 기법이라고 한다. <그림 2>는 Signal Handler 기법에 따른 방식으로 Middleware Timer Handler 를 설계한 것이다.

이 기법의 특징은 Middleware Timer Handler 의 주요 코드 부분이 모두 Signal Handler 에 내장되어 있다는 것이

다. 이렇게 처리되면 원칙적으로 다른 Task 들의 실행이 금지[2]되므로 Middleware Timer Handler 의 작동을 보장 받을 수 있다는 점이 장점이다.

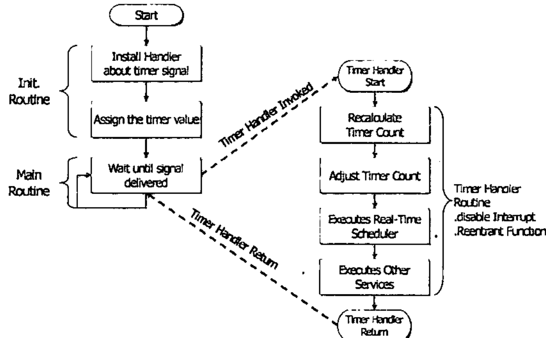

Figure 2. Signal Handling처리 방식의 Middleware Timer

그러나 3.1 절에서 다룬 바와 같이 Signal Handling 처리 코드 부분에 Middleware Timer Handler 의 주요 코드가 내장되어 있으면 배달된 Signal 을 유실할 수 있는 확률이 급격히 증가한다. <그림 3>은 이를 도표로 나타낸 것이다.

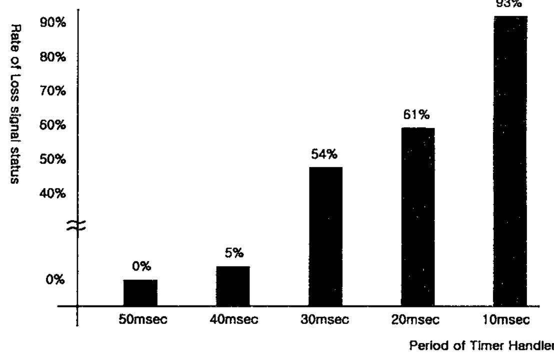

Figure 3. Signal 유실 대 주기 도표

3.3 독자적인 스레드를 사용한 Middleware Timer Handler 설계

본 논문에서는 <그림 4>와 같은 처리 형태를 제안한다. <그림 4>에서 보면 권고안에 따른 전통적인 기법과 차이가 나는 것은 Middleware Timer Handler 가 처리하는 임무들에 대하여 Signal Handler 라는 특별한 순간에 처리되는 것이 아니라 일반 코드라는 점에서 독특하다. 이런 형태의 처리는 Timer Handler 태스크가 설치되어 있는 실시간 처리 Middleware 기반 응용프로그램에서 좀 더 유연한 처리를 보장한다.

이렇게 제안된 Middleware 방식의 효율적인 Timer Handler 의 작동을 정리하여 보면 다음과 같다.

(1) Middleware Timer Handler 태스크의 스케줄링 우선순위를 최우선순위로 초기화한다.

(2) Middleware Timer Handler 는 독자적으로 다른 태스크들과 분리되어야 한다.

(3) Middleware Timer Handler 에서 Signal 처리 부분은 코드를 구성하지 않는다.

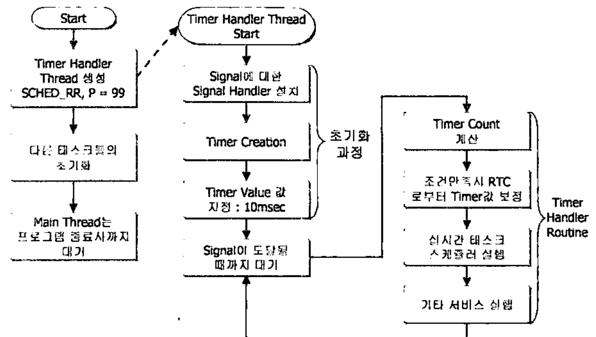

Figure 4. Real-Time Middleware에 적합한 Timer Handler

4. 결론 및 향후 과제

본 논문은 효과적인 실시간 시스템 지원을 위해 Middleware 스케줄러의 역할 중 Middleware Timer Handler 의 효율적인 설계에 관해 논하였다. Middleware Timer Handler 의 처리 방식 중 Signal Handling 방식은 다른 태스크들보다 Timer Handler 가 우선적으로 처리되는 것을 보장 받고, 인터럽트가 금지 되었지만, 이로 인해 Time Quantum 의 주기처럼 매우 짧은 주기에서는 Signal 을 놓치는 상태가 되어 오히려 제대로 처리를 하지 못하는 것을 볼 수 있었다.

이전에는 시스템 처리 능력이 상대적으로 약해 실시간 응용 프로그램에서 가급적 태스크들의 수를 줄여 스케줄링으로 인한 Overhead 를 줄이는 방식으로 Signal Handling 기법을 사용했지만, 현재의 컴퓨터 시스템들은 이들을 처리할 충분한 처리 능력을 가지고 있으므로, 이를 활용하여 태스크 수를 늘림으로써 기존의 Signal Handling 방식의 단점인 인터럽트 금지로 인한 Signal 유실 문제를 극복할 수 있었다.

향후 과제로는 POSIX.1b 실시간 확장 권고안에 기술되어 있는 Signal 배달자체가 Buffering[7] 되는 특징을 활용하여 더욱 효과적인 Middleware Timer Handler 를 설계할 것이다.

5. 참고문헌

[1] O'Reilly, "Understanding the Linux Kernel", Daniel P. Bovet & Marco Cesati

[2] Prentice Hall PTR, "Practical UNIX Programming", Kay A. Robbins & Steven Robbins

[3] http://www.kesl.org/pdsfiles/emb_tutorial/944183035-linux.html, related to http://www.rtlinux.org/~rtlinux

[4] Embedded Systems Programming, Jerry Epplin, 1997/10

[5] Kim, K., "Object-Oriented Real-Time Distributes Programming and Support Middleware", Proc. ICPADS 2000, Iwate, Japan, July 2000, Keynote paper, pp.10-20.

[6] Kim, J.G., Kim, M.H., Min, B.J., and Im, D.B., "A Soft Real-Time TMO Platform - WTMOS - and Implementation Techniques", Proc. ISORC'98, Kyoto, Japan, April 1998

[7] http://hegel.ittc.ukans.edu/projects/posix/signals.html