http://dx.doi.org/10.5392/JKCA.2016.16.05.217

리눅스 사용자 영역에 실시간성 제공을 위한 미들웨어

Middleware to Support Real-Time in the Linux User-Space

이상길, 이승율, 이철훈
충남대학교 컴퓨터공학과

Sang-Gil Lee(sk0137@cnu.ac.kr), Seung-Yul Lee(winrate92@cnu.ac.kr),
Cheol-Hoon Lee(clee@cnu.ac.kr)

요약

리눅스는 범용 운영체제로 스케줄링 특성 상 실시간성을 제공할 수 없는 단점이 있으며, 이를 해결하기 위해 RTiK-Linux을 통해 커널 영역에 실시간성을 지원했다. 하지만 RTiK-Linux 개발 초기 단계로 사용자 영역을 지원하지 않아 실시간성을 요구하는 응용프로그램 개발에 어려움이 있다. 본 논문에서는 RTiK-Linux를 개선하여 사용자 영역에 실시간성을 제공하는 RTiK미들웨어를 설계 및 구현한다. RTiK 미들웨어는 응용 프로그램에서 프로세스 정보와 요청 주기 등록한 뒤, 시그널을 통해 요청한 주기에 따라 사용자 영역에 API를 통해 실시간성을 제공한다. 구현한 RTiK미들웨어의 성능 검증 및 평가를 위해 RDTSC 명령어를 사용하여 생성된 실시간 쓰레드의 주기를 측정하였고, 유저 영역의 1ms 주기에서 오차 범위 내에서 정상 동작함을 확인하였다.

■ 중심어 : | 리눅스 | 범용운영체제 | 실시간 운영체제 | 미들웨어 |

Abstract

Linux it self does not support real-time. To solve this problem RTiK-Linux was designed to support real-time in the kernel space. However, since the user space does not support real-time, it is not easy to develop application. In this paper, we designed and implemented a RTiK-middleware to support real-time in the user space. RTiK-middleware provides real-time scheduling for user space through signal request period after to register process information with request period using apis on application. To evaluate the performance of the proposed RTiK-middleware, we measured the periods of generated real-time threads using RDTSC instructions, and verified that RTiK-middleware operates correctly within the error ranges of 1ms.

■ keyword : | Linux | General-purpose OS | Real-time OS | Middleware |

I. 서 론

x86 기반 운영체제는 2015년 현재 윈도우 운영체제가 높은 점유율을 보이고 있으며, 국내 시장에서는 그 차이가 더욱 크다[1]. 최신 버전의 윈도우의 경우 카피 당 수 십 만원의 구매 비용이 소요되어 공공기관의 경

* 이 연구는 충남대학교 학술연구비에 의해 지원되었음
접수일자 : 2015년 12월 29일
수정일자 : 2016년 01월 26일
심사완료일 : 2016년 01월 29일
교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

우 다수의 PC를 사용하기 때문에 그에 따른 운영체제 구입비용이 지출되고 있다. 또한 국내 PC 환경이 윈도우에 종속된 문제가 있어 2014년 8월 정부 기관인 미래창조과학부(이하 미래부)에서 공개 소프트웨어 활성화 방안을 발표하였다[2].

이를 대체하기 위한 리눅스는 범용 운영체제의 한 종류로써 CFS(Complete Fair Scheduling) 알고리즘을 사용하여 프로세스를 관리한다. 모든 프로세서가 공평하게 수행되도록 하는 본 알고리즘은 특정 프로세스의 우선순위를 완벽하게 보장할 수 없어, 실시간성을 만족할 수 없는 문제가 있으며, 이는 리눅스 커널이 2.6에서 3.x로 버전 업 되는 과정에서도 유사한 구조를 사용했기 때문에 서드파티 솔루션이 필요하게 된다[3-5]. 이를 해결하기 위하여 윈도우 운영체제에 실시간성을 제공하는 RTiK를 참고하여 RTiK-Linux를 개발하였고, 리눅스 운영체제에 실시간성을 제공할 수 있음을 확인하였다[6][7].

통신미들웨어와 태블릿PC를 위한 윈도우 환경에서의 연구 등 활발히 연구된 RTiK와는 달리[8][9], RTiK-Linux는 초기 개발단계에 머물러 있어 실시간성을 필요로 하는 프로그램 개발 시, 개발자가 RTiK-Linux의 내부 구조를 모두 알아야 하는 불편함이 있어 실시간성을 활용한 개발에 어려움이 있다. 또한 기존 실시간 태스크가 수행되지 않음에도 실시간성 제공을 위한 모듈이 동작하여 시스템 부하를 통한 안정성을 해칠 수 있는 단점이 있다[6].

본 논문에서는 위의 단점을 개선한 RTiK미들웨어를 설계 및 구현하였다. RTiK미들웨어는 사용자 영역에 실시간성을 부여할 수 있도록 API를 제공하고 있어 이를 통해 RTiK미들웨어의 모든 기능을 활용할 수 있으며, 개발자가 내부 구조를 몰라도 사용할 수 있도록 사용의 편의성을 개선하였다. 2장에서는 관련 연구로 리눅스에 실시간성을 제공하는 확장 커널과 RTiK-Linux의 구조를 살펴보고 단점을 확인한다. 3장에서는 제안한 RTiK미들웨어의 동작 구조와 실시간 태스크의 처리과정, 그리고 API에 대해 기술한다. 4장에서는 API를 사용한 실시간 주기 확인 프로그램을 작성하여 RTiK미들웨어의 성능을 측정하여 타 솔루션과의 비교를 하며 결론 및 향후 연구를 통해 마무리한다.

II. 관련 연구

1. 실시간 운영체제

실시간 운영체제(Real-Time Operating System, RTOS)는 실시간 응용 프로그램을 위해 개발된 운영체제로 높은 응답성을 요구하고 고 정밀 시스템인 임베디드 시스템이나 마이크로 컨트롤러에 사용된다. 실시간 운영체제는 시스템의 기능 중 CPU 시간 관리 부분에 초점을 맞추어 설계되어 응용 프로그램의 처리 요청을 정해진 시간 내에 수행할 수 있도록 한다[11][12].

2. RTLinux

리눅스는 시분할 라운드 로빈 정책과 우선순위 기반의 공정한 프로세스 실행 및 응답성을 중요시하는 범용 운영체제이다. 태스크 실행 권한은 우선순위 기반 비선점이며, 시간 정밀도는 250Hz(즉, 4ms) 주기성의 연성 실시간 시스템으로 분류된다. 이러한 연성 실시간 리눅스 커널에 실시간 기능을 패치하여 이중 커널 구조로서 약 100us의 주기성을 보장하는 경성 실시간 리눅스인 RTLinux(Real-Time Linux)가 도입되었다.

[그림 1]은 RTLinux의 구조를 나타낸 것이다. RTLinux 스케줄러는 리눅스 커널을 Idle 태스크로 취급하여 실시간 태스크 혹은 실시간 커널이 비 활성화되었을 때 실행한다. 리눅스는 어떠한 경우라도 인터럽트를 중지 시키지 못하며, 선점에 대한 권한이 없다. 이러한 두 운영체제가 동시에 수행될 수 있는 것은 인터럽트 제어 하드웨어에 대한 소프트웨어적 에뮬레이션에 의한다.

실시간 태스크는 커널 공간에서 수행된다. 즉, 실시간 쓰레드는 커널 메모리 공간 내에 실행되므로 스왑 아웃되지 않고 또한 TLB(Translation Lookaside Buffer) 미스 횟수를 줄여 준다. 실시간 쓰레드는 프로세서 슈퍼바이저 모드로 동작하기 때문에, 하드웨어 전반적 제어가 가능하다. 어떤 임의의 자원을 리눅스의 프로세스가 점유함으로써 실시간 태스크가 그 자원을 기다리는 현

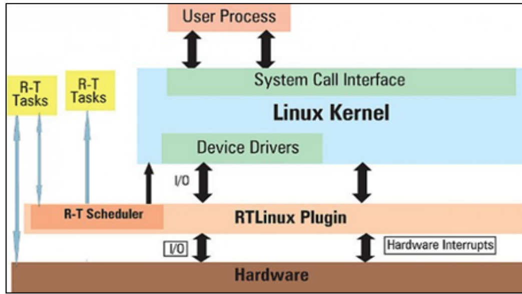

그림 1. RTLinux의 동작 구조

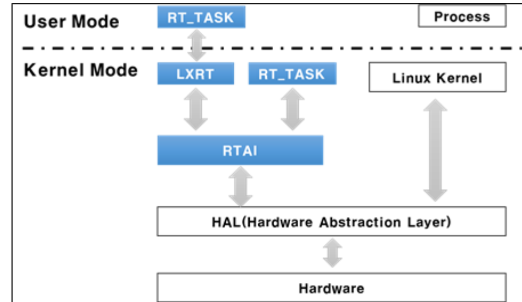

그림 2. RTAI의 동작 구조

상을 허용하지 않는다. 따라서 메모리 요청이나, 어떤 자료 구조에 대한 동기화나 스핀 락을 공유하지 않아야 한다. RTLinux에서 모든 인터럽트 제어권은 RTLinux가 가진다. 기존 리눅스의 인터럽트는 RTLinux에서 소프트웨어적인 에뮬레이션으로 처리한다. 실시간 인터럽트 처리를 위하여 Request_RTirq()와 free_RTirq() 함수의 등록으로 RTLinux를 활성화한다.

실시간 태스크의 응용 범위가 극히 제한적이기 때문에 필요에 따라 리눅스 프로세스에 의한 확장 기능이 필요하다. 이를 위하여 RTLinux에서는 커널 주소공간에 공유 메모리를 이용하여 실시간 태스크와 기존 리눅스 프로세스 간의 데이터 전달을 위하여 실시간 FIFO를 두고 있다[6][13].

3. RTAI

RTAI(Real-Time Application Interface)는 RTLinux 개발 초기에 함께 탄생된 표준 리눅스 커널로서, 리눅스 커널에 기반을 둔 개방형 실시간 운영체제이다.

[그림 2]는 RTAI 커널의 구조를 나타낸 것이다. RTAI는 기본적으로 인터럽트 디스패처로 구성되는데, RTHAL(Real-Time Hardware Abstraction Layer)의 개념을 사용하여 하드웨어 인터럽트 가로채기를 통한 리눅스에 실시간성을 지원하고, 기존 리눅스로 인터럽트를 재 경로 설정하도록 한다. RTLinux와 마찬가지로 기존 리눅스 커널은 유휴 태스크로 취급되어 RTAI 스케줄러에 의해 관리되는 실시간 태스크보다 항상 낮은 우선순위를 갖게 된다.

RTAI 스케줄러에 의한 실시간 태스크는 커널 수준에서 수행되어 경성 실시간을 지원한다. 그러나 기존 리눅스의 라이브러리를 활용하는 다양한 기능성을 지원하기 위해 LXRT(Linux Real-Time) 모듈을 통한 사용자 수준의 실시간 태스크의 수행도 가능하게 한다. LXRT 모듈은 커널 수준에서 실행되며 RTAI 스케줄러에 관리되어 리눅스 프로세스보다 높은 우선순위로 실시간성을 보장받는다[6][14].

4. RTiK-Linux

RTiK-Linux는 2011년에 개발된 리눅스에 실시간성 제공을 위한 솔루션이다. RTiK-Linux는 리눅스에 실시간성을 제공하기 위해 윈도우에 실시간성 제공을 위해 개발된 RTiK을 리눅스 환경에 이식하였다[6][7].

[그림 3]은 RTiK-Linux의 구조를 타나내는 그림이다. RTiK-Linux는 시스템과는 독립적인 타이머를 활용하기 위해 Local APIC Timer Interrupt를 사용한다[15]. 윈도우에서는 운영체제에서 이를 사용하지 않기 때문에 RTiK를 위한 독립적인 타이머로 사용이 가능하였으나, 리눅스에서는 스케줄링을 위하여 사용하고 있기 때문에, RTiK-Linux를 위한 별도의 인터럽트 핸들러를 IDT(Interrupt Descriptor Table) 에 등록시켜 RTiK-Linux의 핸들러를 수행하도록 하였다. 즉, RTiK-Linux 모듈이 등록될 때 기존의 리눅스에서 수행되던 Local APIC Timer 핸들러를 대신해 RTiK-Linux의 Local APIC Timer 핸들러를 수행하게 된다. 따라서 리눅스는 RTiK-Linux 모듈이 적재되면 RTiK-Linux 디스패처에 의해 실행권을 부여받게 되는데, RTiK-Linux의 유휴(Idle) 태스크 수행 시간에 리눅스

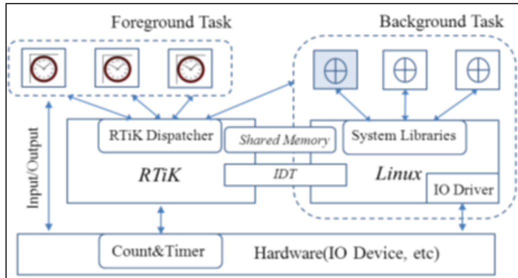

그림 3. RTiK-Linux의 구조

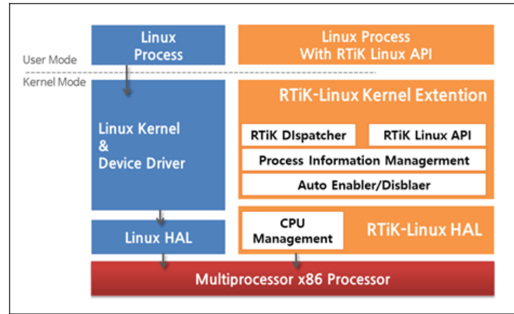

그림 4. 제안하는 RTiK미들웨어의 구조

의 태스크가 수행 되도록 구현되어 있다[6].

RTiK-Linux는 Local APIC Timer 를 통하여 커널 영역에 실시간성을 위한 주기를 제공하고 있으며, 10ms, 1ms, 0.1ms의 실험을 통해 RTLinux와 유사한 성능을 보이고 있는 것을 확인하였으나, 이는 주기 제공을 위한 기본 연구만이 진행되어 있으며, 멀티프로세서 환경을 위한 연구방안 또한 비교적 최근에 제시되었다[6][10].

이에 따라 기존 제안된 RTiK-Linux의 기능을 확장하여 실시간 시그널을 통해 사용자 영역에 실시간성을 제공하는 방법과 사용자 영역에서 실시간 태스크를 사용하는 프로세스를 관리하기 위한 방안을 본 연구에서 수행하였고, 이를 쉽게 사용할 수 있도록 API를 설계 및 구현하였다.

Ⅲ. 리눅스 사용자 영역에 실시간성 제공을 위한 미들웨어 설계 및 구현

1. 실시간성 제공을 위한 미들웨어 설계 방안

본 논문에서 제안하는 RTiK미들웨어는 범용운영체제인 리눅스에 실시간성을 부여한다. 일반 프로그램이 동작하는 유저 영역에서 처리할 수 없는 인터럽트 관리와 프로세스 관리 등의 동작을 위해 시스템 커널에 확장하는 디바이스 드라이버 형태로 사용된다.

[그림 4]는 기존 RTiK-Linux를 개선한 RTiK미들웨어의 구조를 나타낸다. RTiK-Linux의 HAL에서는 실시간 타이머 관리를 위한 모듈이 있다. RTiK-Linux Kernel Extension에서는 프로세스 관리를 위한 모듈과 실시간 타이머의 처리 함수에 해당하는 RTiK Dispatcher를 포함하고 있다.

이를 바탕으로 본 미들웨어는 커널 영역에서 유저 영역에 실시간성을 제공할 수 있도록 위치해 있다. 본 단락에서는 RTiK미들웨어의 설계 방안에 대해서 기술한다.

1.1 유저 영역에 실시간성 제공 방법

커널 영역에서는 실시간성을 부여하기 위해 HAL Extension에서 발생하는 인터럽트를 사용하여 실시간 시스템에 사용되는 Time Tick ISR을 구현하였다. 이를 유저 영역에 전달하기 위하여 리눅스에서 제공하는 실시간 시그널(Real-Time Signal, RTS)을 사용하였다.

실시간 시그널은 비동기 이벤트 전달을 목적으로 만들어졌으며, 기존 유닉스에서 사용하던 시그널의 기능을 확장하여 대기열을 제공하여 같은 종류의 시그널을 중첩 발생할 수 있도록 하였다.

실시간 시그널은 SIGRTMIN으로 지정된 32번부터 SIGRTMAX로 지정된 63번까지의 번호를 사용하며 본 논문에서 제안한 미들웨어에서는 34번을 지정하여 사용하고 있다. [그림 5]는 선언된 실시간 시그널의 모습이다.

```
21  #define RTiK_SIGNAL 34
```

그림 5. 실시간 시그널인 RTiK_SIGNAL의 선언

본 미들웨어에서는 실시간 시그널을 사용하여 사용자 영역에 신호를 보내는 것으로 실시간성을 부여한다. 리눅스의 시스템 콜을 사용하여 시그널을 대기할 때, Block 상태에 빠지는 것을 이용하여 원하는 시간동안 프로세스를 중단 시키도록 하며, Local APIC Timer를 사용하여 정확한 시간에 Block 상태에 빠졌던 프로세스에 신호를 보내 주기성을 제공한다.

[그림 6]은 주기성을 제공하기 위한 동기화 과정을 나타낸 그림으로 각 단계는 다음의 과정으로 설명한다.

1) RTiK_User_WaitSignal()

RTiK 미들웨어 API로 제공하는 RTiK_User_WaitSignal을 사용자가 작성한 프로그램에서 주기성을 원하는 곳에 호출한다. 이 함수를 사용해서 실시간 시그널을 받기 위한 Block 상태로 진입할 수 있다.

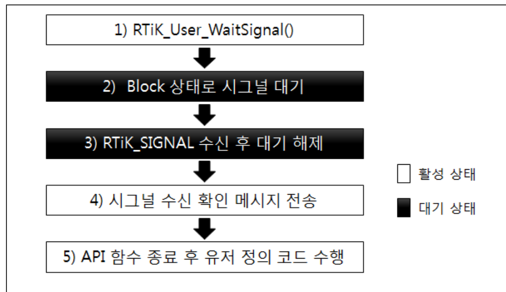

그림 6. Wait&Signal의 동작 과정

2) Block 상태로 시그널 대기

미들웨어 API를 통해서 시그널 대기 상태로 변경될 때, 앞서 정의한 실시간 시그널을 대기하여 해당 시그널이 RTiK 미들웨어에서 전송될 때 까지 유저 프로세스는 블록 상태로 대기한다.

3) RTiK_SIGNAL 수신 후 대기 해제

RTiK 미들웨어에서 주기에 따라 실시간 시그널인 RTiK_SIGNAL을 전송하면, 대기 상태에 있던 프로세스가 이를 수신하여 대기 상태에서 해제되어 프로세스가 재 수행된다.

4) 시그널 수신 확인 메시지 전송

RTiK 미들웨어에서는 시스템의 효율을 위한 방법을 사용하는데, 이는 시그널을 받는 프로세스의 생존 여부에 따라 실시간 타이머를 수행/중단 할 수 있도록 한다. 시그널 수신 확인 메시지는 이를 위해 실시간 시그널을 받는 프로세스의 생존 여부를 알리기 위한 시그널 수신 확인 메시지를 미들웨어로 전송한다.

5) API 함수 종료 후 유저 정의 코드 수행

RTiK_User_WaitSignal 함수에서 제공하는 기능을 모두 수행한 후, 유저가 주기성을 얻기 위해 작성한 코드를 수행한다.

위와 같은 과정을 통해 프로세스 수행의 동기화를 제공한다. 이를 간단히 도식화하여 [그림 7]과 같이 나타낼 수 있다.

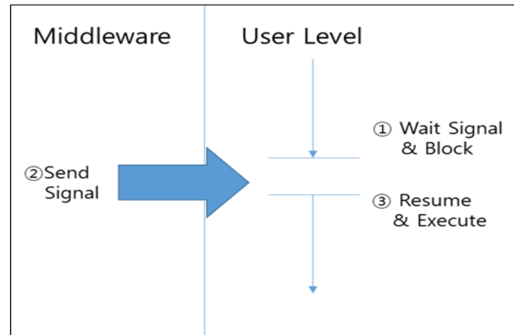

그림 7. 유저 영역 수행에 동기화를 위한 동작 과정

1.2 유저 프로세스 정보 관리를 위한 자료구조

커널 영역인 디바이스 드라이버에서는 유저 영역 프로세스의 정보를 얻을 수 있기 때문에 실행되는 모든 프로세스에 시그널을 전송할 수 있으나, 이 방식은 매우 비효율적이며 시스템의 성능 저하를 유발할 수 있기 때문에 실시간성을 제공받는 특정 프로세스에만 시그널을 전송할 수 있도록 프로세스 정보를 관리해야 한다.

[표 1]는 본 미들웨어에서 프로세스 정보를 관리하기 위한 RTiK_ProcInfo_t 구조체를 정의한 것이다.

표 1. 프로세스 정보 관리를 위한 RTiK_ProcInfo_t 구조체

이름	자료형	설명
pid	pid_t	유저 영역의 프로세스 ID
periodic	unsigned int	해당 프로세스가 원하는 주기
next	struct _processInfo*	리스트 관리를 위한 포인터
life	int	전력관리를 위한 자료구조

리눅스에서는 PID(Process ID)를 통해 프로세스 정보에 접근이 가능하기 때문에, 리눅스의 프로세스 정보를 관리하는 구조체를 직접 포인트 하는 것이 아닌 유저 영역 프로세스의 PID를 저장한다. 다음으로 주기 값을 저장하여 해당 프로세스가 원하는 주기를 ms 단위로 입력받아 저장한다. 이 필드를 사용하여 원하는 주기에 시그널을 받을 수 있도록 처리한다. 또한 해당 구조체를 효율적으로 관리하기 위해 주기를 우선순위로 하여 연결 리스트로 관리된다.

[그림 8]은 프로세스 정보 관리를 위해 연결 리스트 구성을 보여준다. 전역 변수로 선언된 RTiK_ProcInfoList에서 현재 사용되고 있는 프로세스의 정보를 연결 리스트로 관리한다. 주기를 우선순위로 관리되고 있는 것을 보여주기 위해 5ms의 주기를 갖는 프로세스(1150)이 등록되어 있는 상황에서 3ms의 주기를 갖는 프로세스(2130)이 추가된다. 이때 우선순위를 기준으로 오름차순으로 확인하여 리스트의 순서가 변경된 것을 나타내는 그림이다.

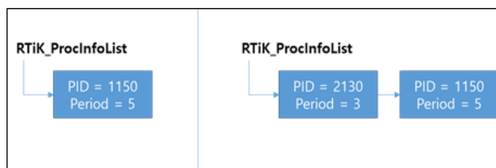

그림 8. 프로세스 정보 관리를 위한 리스트

또한 프로세스 정보를 담는 자료구조의 경우 각각의 공간을 동적 할당받지 않고 전역 변수로 버퍼에 미리 담겨 있는 것을 사용하도록 한다. 이는 동적할당시 발생하는 메모리 검색에 의한 수행 시간이 예측 불가능한 것을 방지하고자 함으로 다음 [그림 9]와 같이 전역 변수로 선언한다.

```
/*
 * Global Variable List
 */
RTiK_ProcInfo_t* RTiK_ProcInfoList              = NULL;
RTiK_ProcInfo_t* RTiK_ProcInfoAvailableList     = NULL;
RTiK_ProcInfo_t
RTiK_ProcInfo_availableArray[100] ;
```

그림 9. 프로세스 관리를 위해 사용되는 전역변수 목록

자료구조의 공간을 할당하기 위해 버퍼로 사용되는 RTiK_ProcInfo_availableArray 배열을 선언하고 있으며, 리스트 관리를 위한 두 개의 포인터인 RTiK_ProcInfoList 와 RTiK_PricInfoAvailableList를 선언하였다. 이는 사용되고 있는 목록과 사용할 수 있는 목록을 구분하기 위하여 사용되고 있다.

1.3 유저 영역 프로세스 등록 과정

유저 영역 프로세스 정보 관리를 위한 자료구조를 사용하여 프로세스 정보를 등록하기 위한 과정으로 [그림 10]에서 나타내는 순서가 필요하다.

[그림 10]은 프로세스 등록 과정을 나타낸 것으로 밝은 색으로 표기한 블록은 유저 영역에서 수행되는 작업을 말하며, 어두운 색 배경이 적용된 블록은 커널 영역에서 수행되는 작업을 나타낸다. 다음을 통해 각 단계를 설명한다.

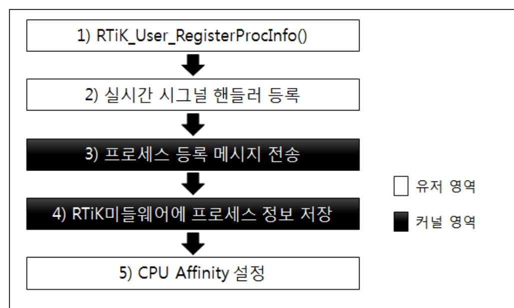

그림 10. 프로세스 등록 과정

1) RTiK_User_RegisterProcInfo() Call

RTiK 미들웨어 API로 제공되는 함수인 RTiK_User_RegisterProcInfo를 사용자가 작성한 프로그램의 초기화 과정에서 호출한다. 이 함수를 통해 프로세스

정보 등록을 요청하게 되며, 이 과정에서 해당 프로세스가 원하는 주기 정보를 인자로 넘겨준다.

2) 실시간 시그널 핸들러 등록

RTiK 미들웨어에서 사용하는 실시간 시그널인 RTiK_SIGNAL의 핸들러를 등록한다. 리눅스의 프로세스에서 설정된 기본 핸들러에서는 부가 기능을 수행할 수 없기 때문에 시그널을 수신 후 부가 기능을 처리할 수 있도록 하는 핸들러 등록 과정이 필요하다.

3) 프로세스 등록 메시지 전송

RTiK 미들웨어에 기능을 요청하기 위해 ioctl() 함수를 호출한다. API의 경우 유저 영역에서 수행하기 때문에 미들웨어의 기능에 직접 호출할 수 없어 미들웨어에 프로세스 등록 메시지를 보내는 방법을 사용한다.

4) RTiK 미들웨어에 프로세스 정보 저장

프로세스 등록 메시지를 받은 RTiK 미들웨어에서는 넘겨 받은 프로세스 정보를 미들웨어에서 관리하는 목록에 추가한다. 해당 과정을 거친 후 RTiK 미들웨어에서 유저 프로세스에게 시그널을 보낼 수 있게 되는 것이다.

이런 과정을 통해 RTiK 미들웨어 API를 사용하는 유저 프로세스 정보가 미들웨어에 등록되며, 이를 사용하여 프로세스에 시그널 주기에 맞추어 전송할 준비가 된 것이다.

2. 유저 영역에 실시간성 제공을 위한 API 구현

본 단락에서는 본 논문에서 제안한 설계를 통해 유저 영역에 실시간성을 제공하는 API의 구현에 대해 설명한다. 미들웨어는 API를 통해 유저 영역에서 실시간성을 제공받을 수 있도록 하였다.

2.1 미들웨어 API를 위한 ioctl()

유저 영역 프로그램에서 미들웨어의 기능을 제공받기 위한 API가 필요하다. API는 디바이스 드라이버의 IO 제어를 위하여 ioctl() 시스템 콜을 사용하게 된다[4]. ioctl()은 저 수준의 작업을 요청하는 것으로 open, write 등의 기능이 아닌 부가기능의 수행을 위한 코드를 다음과 같이 정의한다.

표 2. 미들웨어의 ioctl 코드

이름	설명
RTiK_IOCTL_MAGIC	타 디바이스 드라이버의 ioctl과 구분하기 위해 사용되는 문자
RTiK_IOCTL_SIGNAL_ACK	프로세스와 전력관리를 위해 사용되는 코드
RTiK_IOCTL_PROCESS_INSERT	커널 영역과 동기화를 위해 커널 영역에 프로세스의 정보를 등록하는 코드
RTiK_IOCTL_PROCESS_REMOVE	커널 영역에 프로세스 정보를 제거하기 위해 사용되는 코드
RTiK_IOCTL_GET_TIMER_STATUS	미들웨어의 실시간 타이머 동작 정보를 확인하기 위한 코드
RTiK_IOCTL_TEST_SIGNAL	미들웨어에게 디버깅을 위한 신호를 요청하는 코드

[표 2]는 미들웨어의 ioctl의 각 코드를 설명한 표이다. 본 ioctl 코드를 바탕으로 미들웨어에 접근하는 함수에 응답할 수 있는 것으로 _rtik_ioctl() 함수를 사용한다.

[그림 11]은 RTiK미들웨어 내의 ioctl을 처리하기 위해 사용되는 함수로 유저 영역에서 데이터를 불러오거나 내보낼 수 있다. 이를 구분하기 위해 _IOC_SIZE(cmd)를 활용하며, 유저 레벨에서 전달한 데이터의 크기로 데이터의 유/무를 확인한다. 다음을 통해 ioctl 코드의 동작을 기술한다.

```
/*
 * File Operation of RTiK-Linux Device Driver
 */
static DEFINE_MUTEX(_rtik_mutex);
static int _rtik_ioctl ( struct inode *inode, struct file *filep, unsigned int cmd, unsigned long arg )
{
    RTiK_Pass_Data user_data;
    int data_size = _IOC_SIZE (cmd);
    RTiK_ProcInfo_t * procInfo = NULL;
    int result = 0;
    printk(" [RTiK] _ioctl function call \n" );
    mutex_lock(&_rtik_mutex);
    switch ( cmd ) {
        case RTiK_IOCTL_PROCESS_INSERT: {

        case RTiK_IOCTL_PROCESS_REMOVE: {

        case RTiK_IOCTL_SIGNAL_ACK: {
        case RTiK_IOCTL_TEST_SIGNAL: {
        case RTiK_IOCTL_GET_TIMER_STATUS: {
        default:
            printk("Undefined IOCTL Code -> %d\n", cmd);
    }
    mutex_unlock(&_rtik_mutex);
    printk(" [RTiK] _ioctl done \n" );
    return result;
}
```

그림 11. RTiK미들웨어 내 ioctl 처리 함수

1) RTiK_IOCTL_PROCESS_INSERT

RTiK_IOCTL_PROCESS_INSERT는 API를 사용한

프로세스가 RTiK미들웨어에 정보를 저장하기 위하여 사용된다. 사용자 영역에서 주기와 PID를 넘겨받으며 몇 가지 예외처리를 한 다음 프로세스 정보를 저장하는 구조체를 통해서 리스트에 연결한다.

2) RTiK_IOCTL_PROCESS_REMOVE

RTiK_IOCTL_PROCESS_REMOVE는 API를 사용한 프로세스의 정보를 RTiK미들웨어 내의 리스트에서 삭제하기 위하여 사용된다. 유저 영역에서 전달한 pid를 통해 리스트에서 삭제하는 코드를 수행한다.

3) RTiK_IOCTL_SIGNAL_ACK

RTiK_IOCTL_SIGNAL_ACK는 API를 통해 실시간 시그널을 전달받은 프로세스가 정상적으로 시그널을 수신했다는 것을 미들웨어에 알릴 때 사용한다. 유저 영역에서 전달한 pid를 확인하는 것으로 리스트 내의 프로세스 정보 저장을 위한 구조체의 life 정보를 초기화 해준다.

4) RTiK_IOCTL_GET_TIMER_STATUS

RTiK_IOCTL_GET_TIMER_STATUS는 미들웨어에 실시간 타이머의 상태 정보를 유저 영역에 전달한다. 타이머의 수행 여부나 미들웨어를 사용하는 프로세스의 수 등 여러 가지 정보를 전달 받을 수 있도록 구현하였다.

2.2 미들웨어를 위한 API 설계 및 구현

본 논문에서 제안한 ioctl 코드를 이용하여 미들웨어를 통해 실시간성을 제공받을 수 있도록 API를 구현하였다. 미들웨어 API는 사용자가 내부구조를 명확하기 알고 있지 못한 상황에서도 실시간성을 제공받을 수 있도록 하기 위해서 사용된다. [표 3]은 RTiK미들웨어가 제공하는 API의 목록과 그 기능을 나열한 것으로 각 함수에 대한 설명은 다음 단락에서 소개한다.

1) RTiK_User_RegisterProcessInfo

본 함수는 RTiK_SIGNAL을 받기 위해 응용 프로그램의 정보를 미들웨어에 등록하고 시그널 핸들러를 등

표 3. RTiK 미들웨어에서 제공하는 API 목록

이름	설명
RTiK_User_RegisterProcessInfo	유저 프로세서의 정보 등록을 위해 사용하는 함수
RTiK_User_UnregisterProcessInfo	유저 프로세스의 정보 삭제를 위해 사용하는 함수
RTiK_User_WaitSignal	실시간 시그널 대기를 위해 사용하는 함수
RTiK_User_GetTimerStatus	타이머의 상태 값을 반환하기 위해 사용하는 함수
RTiK_User_CheckTimerPeriodic	타이머의 주기 측정을 위해 사용하는 함수

록하는 함수이다. 본 함수를 사용하지 않으면 미들웨어에서 해당 프로세스의 정보를 알 수 없기 때문에 프로세스 초기화 과정에서 호출해야 한다.

표 4. RTiK_User_RegisterProcessInfo 함수

int RTiK_User_RegisterProcessInfo (*unsgiend int* period)	
구분	설명
인자	unsigned int period: 요청할 주기 값(ms 단위)
반환 값	int: 함수 수행의 성공/실패 여부 반환

2) RTiK_User_UnregisterProcessInfo

본 함수는 유저 프로그램이 실행 중 더 이상 실시간 시그널을 받지 않기를 원할 때, 프로세스 정보를 삭제하기 위해 사용되는 함수이다. 이 함수를 사용하면 더 이상 실시간 시그널을 받지 않기 때문에 더 이상 동기화를 통한 주기성 획득이 불가능하다.

표 5. RTiK_User_UnregisterProcessInfo 함수

int RTiK_User_UnregisterProcessInfo ( )	
구분	설명
인자	없음
반환 값	int: 함수 수행의 성공/실패 여부 반환

3) RTiK_User_WaitSignal

본 함수는 미들웨어에서 전송하는 실시간 시그널을 대기하여 동기화를 얻을 때 사용하는 함수이다. 본 함수를 통해 유저 영역에서 주기에 따른 실시간성을 제공받을 수 있게 할 수 있다.

표 6. RTiK_User_WaitSignal 함수

int RTiK_User_WaitSignal ( )	
구분	설명
인자	없음
반환 값	int: 함수 수행의 성공/실패 여부 반환

4) RTiK_User_GetTimerStatus

본 함수는 RTiK미들웨어의 모니터링을 위해 사용하는 함수로써 미들웨어에서 타이머의 동작 상태등 여러가지 정보를 확인할 수 있도록 사용할 수 있다. 현재는 타이머의 동작 여부와 API를 사용하는 프로세스의 수를 반환해준다.

표 7. RTiK_GetTimerStatus 함수

int RTiK_User_GetTimerStatus ( )	
구분	설명
인자	없음
반환 값	int: 타이머 수행 여부 반환

5) RTiK_User_CheckTimerPeriodic

본 함수는 모니터링 프로그램을 위하여 작성한 함수로써 RTiK미들웨어 API를 사용한 예제 코드로 제공되고 있다. 본 함수는 API를 사용하여 유저 영역에서 타이머의 주기를 측정할 수 있도록 제작된 함수이다. 측정용 파일에 RDTSC(Read Time Stamp Counter) 명령어를 이용하여 Local APIC Timer Count 값을 기록한 다음 타이머 발생 주기의 차를 구하여 주기를 측정한다.

표 8. RTiK_User_CheckTimerPeriodic 함수

int RTiK_User_CheckTimerPeriodic ( )	
구분	설명
인자	없음
반환 값	int: 타이머 수행 여부 반환

Ⅳ. 실험 환경 및 실험 결과

실험 환경

RTiK미들웨어가 리눅스에 정상적으로 이식되어 실시간성을 제공함을 검증하기 위한 실험환경은 다음과 같이 구성하였다.

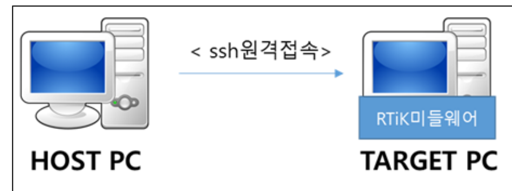

그림 12. 실험을 위한 PC 환경(1)

표 9. 실험을 위한 PC 환경(2)

	Host PC	Target PC
OS	Windows 7 x84	Ubuntu 10.04 LTS x86
Kernel Version	-	2.6.32.63 LTS
Tool	PuTTY for ssh remote	GCC 4.4.3
CPU	Intel i7-2600 @ 3.4 GHz	Intel Core2Duo E8400 @ 3.00 GHz

[그림 12]와 [표 9]은 실험 환경을 나타낸 것으로 타겟 컴퓨터에 리눅스 시스템을 설치한 다음 RTiK미들웨어를 이식한다. 리눅스는 사용자 다중 쉘 접속을 허용하기 때문에 ssh 프로토콜을 이용하여 원격 접속을 통해 디버깅 및 모니터링 하였다.

실험은 RTiK미들웨어 API를 사용한 모니터링 프로그램을 작성한 다음 해당 프로그램을 사용하여 주기를 측정하였고, 정확한 주기를 측정하기 위해 RDTSC 명령어를 사용하였다.

실험에 사용된 타겟 컴퓨터에는 Ubuntu 10.04 LTS(Long-Term Support) 버전의 리눅스가 사용되었다. 커널은 2.6.32.63 버전을 사용하였으며 사용된 컴파일러는 GCC 4.4.3 버전이다.

실험 방법은 1ms의 주기로 설정된 타이머 인터럽트가 발생하여 사용자 영역에 시그널을 전송할 때 마다 클럭 틱 값을 저장한다. 또한 리눅스의 다른 프로세스

에 영향을 받지 않고 CPU를 선점하여 동작함을 보이기 위해 CPU의 사용율을 높이는 워크로드를 stress 라는 유틸리티 프로그램을 사용하여 시스템 부하를 적용하였다. 본 실험에서는 6개의 stress 프로세스가 동작하도록 구성하였다.

```
sslab@ubuntu:~/rtik-linux-source/interfaceCode$ stress -c 2 -i 4
stress: info: [3657] dispatching hogs: 2 cpu, 4 io, 0 vm, 0 hdd
```

그림 13. 시스템 부하 적용을 위한 stress의 구성

[그림 13]은 시스템 부하 적용을 위해 stress 프로그램을 구성하는 것으로 2개의 cpu fork 프로세스와 4 개의 io fork 프로세스를 구성하였다.

```
top - 23:38:28 up 18:42,  3 users,  load average: 5.32, 1.61, 0.56
Tasks:  89 total,   4 running,  85 sleeping,   0 stopped,   0 zombie
Cpu0  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   4079336k total,   374464k used,  3704872k free,   164656k buffers
Swap:  1124508k total,        0k used,  1124508k free,   138428k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3658 sslab     20   0  1828  192  116 R  100  0.0   1:16.52 stress
 3660 sslab     20   0  1828  192  116 R   83  0.0   1:05.91 stress
 3659 sslab     20   0  1828  188  112 R    4  0.0   0:02.09 stress
 3661 sslab     20   0  1828  188  112 D    4  0.0   0:01.61 stress
 3662 sslab     20   0  1828  188  112 D    4  0.0   0:01.64 stress
 3663 sslab     20   0  1828  188  112 D    4  0.0   0:01.92 stress
    1 root      20   0  2788 1628 1220 S    0  0.0   0:00.69 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S    0  0.0   0:00.01 ksoftirqd/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    7 root      20   0     0    0    0 S    0  0.0   0:00.01 ksoftirqd/1
    8 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
```

그림 14. top 명령어를 통한 시스템 자원 확인

[그림 14]는 리눅스에서 제공하는 top 명령어를 통하여 시스템 자원의 사용률을 확인한 그림이다. 이와 같은 환경에서 부하 테스트를 수행하였다.

2. 실험 방법 및 실험 결과

2.1 Idle 상태에서의 1ms의 주기 측정

RTiK미들웨어의 API의 성능을 평가하기 위해 시스템의 유휴 상태에서 사용자 영역에 1ms 주기를 가지는 프로세스를 등록하여 측정하였다.

[그림 15]는 RTiK미들웨어를 통해 사용자 영역의 1ms 주기를 가지는 프로세스를 등록하여 시스템 유휴 상태에서 측정한 결과를 나타내는 그래프이다. 성능 측정을 위해 1만 개의 데이터를 기록하였으며, 그래프의 x축 값은 RTiK미들웨어에서 사용자 영역에 시그널 전송을 통한 동작 횟수를 의미하고 y 축은 주기 값을 의미한다.

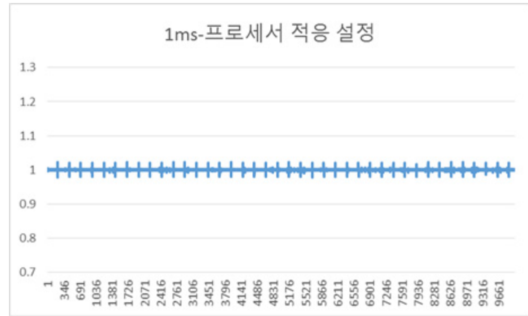

그림 15. Idle 상태에서 RTiK미들웨어를 사용한 1ms 주기

2.2 워크로드 적용 상태에서의 1ms 주기 측정

[그림 16]은 유틸리티를 사용하여 시스템 부하를 적용한 실험 결과로 유휴 상태의 실험과 동일한 1만 개의 데이터를 측정하여 기록하였다.

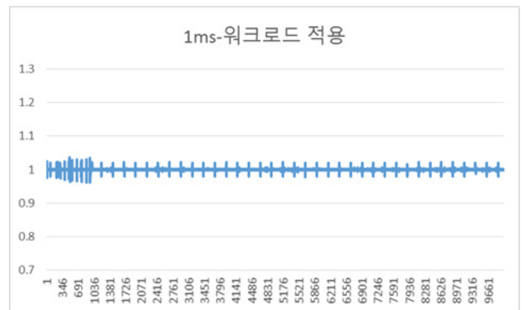

그림 16. 시스템 부하를 적용한 1ms의 주기

2.3 실험 결과

[표 10]은 1ms의 주기 측정 결과를 나타낸 것으로 결과 그래프에서 확인할 수 있는 최대값과 최소값, 평균값과 오차율을 나타낸 표이다. 결과에서 볼 수 있듯이 시스템 유휴 상태에서 4%의 오차율을 보이는 것으로 시스템 커널로 제공되는 RTAI에서 아무런 프로세스가 수행되지 않을 경우보다는 성능 차이가 있지만 5% 이하의 오차율을 보이는 것으로 크게 성능을 벗어나지 않아 실시간성을 제공할 수 있는 것을 보여준다.

시스템 부하를 적용하였을 경우에도 8%의 오차율을 보이고 있으나 그래프를 통해서 볼 수 있듯이 크게 벗어난 주기가 없이 설정 값의 10% 내에서 주기를 제공하는 것을 보여주는 것으로 실시간성을 만족하고 있다.

표 10. 1ms 주기 측정 결과

종류	구분	값
Idle 상태 RTiK 미들웨어	최대값	1.023 ms
	최소값	0.978 ms
	평균값	1.000 ms
	오차	4 %
Idle 상태 RTAI	최대값	1.002 ms
	최소값	0.996 ms
	평균값	1.000 ms
	오차	0.02 %
시스템 부하 적용 RTiK 미들웨어	최대값	1.037 ms
	최소값	0.962 ms
	평균값	1.000 ms
	오차	8 %

이를 통해 시스템의 유휴 상태에서의 성능과 시스템 부하 상태에서의 성능 측정 결과가 실시간성을 만족할 수 있음을 확인하였으며, 본 논문에서 제안한 API를 사용하였을 경우 실시간성을 제공받을 수 있음을 실험을 통해 검증하였다.

3. 기존 연구와의 비교 분석

기존 개발된 RTiK-Linux와 본 논문에서 이를 확장 구현한 RTiK미들웨어의 차이점을 다음 표를 통해 나타낸다.

표 11. RTiK-Linux와의 비교

실시간성 제공 방안		기존 RTiK-Linux	RTiK 미들웨어	RTiK (Windows)
최소 주기	Kernel	0.1 ms	0.1 ms *	0.1 ms
	User	-	1.0 ms	1.0 ms **
API 제공 여부		지원안함	지원함	지원함

[표 11]는 RTiK-Linux, RTiK미들웨어 그리고 윈도우 운영체제 기반으로 제작된 RTiK를 비교한 것으로, 모든 실시간성 제공방안에서 커널 영역에 최소 주기 0.1ms를 지원하고 있다. 본 논문에서 제안한 RTiK미들웨어는 RTiK-Linux의 커널 영역 지원을 이용하기 때문에 커널 영역에서 동일한 성능을 보여주고 있으며, 새롭게 제공되는 사용자 영역에서는 1.0 ms의 주기를 보장하는 것으로 윈도우 운영체체에서 사용되는 RTiK과 동일한 성능을 제공하고 있다. 이를 통해 윈도우에서와 동일한 성능을 보여주는 것으로 RTiK미들웨어의 성능을 검증할 수 있는 것이다.

V. 결론 및 향후 연구과제

본 논문에서는 기존 개발된 RTiK-Linux를 개선한 RTiK미들웨어를 제안하며 이를 통하여 범용 운영체제인 리눅스의 사용자 영역의 프로그램에서 실시간성을 부여받을 수 있도록 하는 연구를 진행하였다. 또한 API를 설계 및 구현하여 개발자가 본 미들웨어의 내부 구조를 전부 이해하지 못하더라도 리눅스 상에 실시간성을 제공받을 수 있도록 하였다.

구현된 RTiK 미들웨어 API의 성능을 검증하기 위해 RDTSC를 사용하였고 최소 주기인 1ms를 지정하여 실험하였을 경우 워크로드가 없을 때 약 4%의 오차 범위 내에서 동작하며, 워크로드를 적용하였을 경우 약 8%의 오차로 동작하는 것을 확인하여 빈번한 스케줄링에도 리눅스의 다른 프로세스에 영향을 받지 않고 CPU를 선점하여 동작하는 것으로 안정적인 주기를 바탕으로 실시간성을 제공하는 것을 증명하였다.

향후 연구과제로는 현재 널리 사용되고 있는 솔루션에서 제공하는 것과 유사하게 리눅스와 완전 분리된 실시간 스케줄러를 개발하는 것이다. 이는 현재 사용자 레벨에서 분리 사용되고 있는 CPU를 RTiK미들웨어를 위한 모든 기능을 위해 분리 하는 것으로 완전한 실시간성 지원 솔루션으로 발전이 가능할 것이다. 또한 x86 PC에 적용되어 있는 본 RTiK미들웨어를 임베디드 환경에서 사용 가능하도록 ARM 플랫폼에 적용하는 연구를 통해 더욱 확장된 이식성을 갖추도록 하는 연구가 필요하다.

참 고 문 헌

[1] http://marketshare.hitslink.com, 2015.12.01
[2] “공개 SW 활성화 정책 토론회,” 국가정책토론회,

2014.06.25.

[3] M. Tim Jones, *Inside the Linux 2.6 Completely Fair Scheduler*, IBM developerWorks, 2009.

[4] Rover Love, *Linux Kernel Development* (3rd Edition), Addison-Wesley, 2011.

[5] 로버트 러브 저, 황정동 역, *리눅스 커널 심층 분석*, 에어콘출판, 2013.

[6] 김주만, 송창인, 이철훈, "RTiK-Linux: 리눅스용 실시간 이식 커널의 설계," 한국콘텐츠학회논문지, 제11권, 제9호, 2011.

[7] 주민규, 이진욱, 김종진, 조한무, 박영수, 이철훈, "x86 기반의 윈도우 상에서 실시간성 지원 방법," 한국차세대컴퓨팅학회논문지, 제11권, 제4호, 2011.

[8] 조아라, 송창인, 이철훈, "윈도우즈 상에서 실시간 디바이스 드라이버를 위한 통합 미들웨어," 한국콘텐츠학회논문지, 제13권, 제3호, 2013.

[9] 박지윤, 조아라, 김효중, 최정현, 허용관, 조한무, 이철훈, "태블릿 PC 환경의 실시간 처리 기능 지원," 한국콘텐츠학회논문지, 제13권, 제11호, 2013.

[10] 이상길, 이철훈, "멀티프로세서 기반 리눅스에 실시간성 지원 방안 연구," 한국콘텐츠학회 종합학술대회 논문집, pp.57-58, 2015(5).

[11] C. M. Krishna and Kang G. Shin, *Real-Time Systems*, McGraw-Hill, 1997.

[12] TPI KOREA, 리얼타임 OS(Real time Operating System)

[13] http://www.rtlinuxfree.com

[14] http://www.rtai.org

[15] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Vol.3: System Programming Guides*, Intel, 2009.

저 자 소 개

이 상 길(Sang-Gil Lee) 정회원

- 2014년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2016년 2월 : 충남대학교 컴퓨터공학과(공학석사)
- 2016년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템

이 승 율(Seung-Yul Lee) 준회원

- 2015년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2015년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템

이 철 훈(Cheol-Hoon Lee) 정회원

- 1983년 2월 : 서울대학교 전자공학과(공학사)
- 1988년 2월 : 한국과학기술원 전기 및 전자공학과(공학석사)
- 1992년 2월 : 한국과학기술원 전기 및 전자공학과(공학박사)
- 1983년 3월 ~ 1986년 2월 : 삼성전자 컴퓨터 사업부 연구원
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터 사업부 선임연구원
- 1994년 2월~1995년 2월 : Univ. of Michigan 객원 연구원
- 1995년 5월 ~ 현재 : 충남대학교 컴퓨터공학과 교수
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 초빙 연구원

<관심분야> : 실시간시스템, 운영체제, 고장허용 컴퓨팅, 로봇 미들웨어