ARX 실시간 운영체계에서의 효율적인 사용자 레벨 입출력

서양민 박정근○ 홍성수
서울대학교 전기공학부

Efficient User-level I/O in the ARX Real-Time Operating System

Yangmin Seo Jungkeun Park Seongsoo Hong
School of Electrical Engineering, Seoul National University, Seoul 151-742, Korea

요 약

사용자 레벨 입출력은 유연성 있고 효율적인 디바이스 드라이버를 만들 수 있게 해주기 때문에, 내장 실시간 시스템에서 그 중요성이 더해가고 있다. 내장 실시간 시스템에서 사용자 레벨 입출력을 지원하기 위해서는 운영체계가 입출력 장치에서 발생한 외부 인터럽트를 프로세스에 예측 가능하고 효율적인 방법으로 전달할 수 있는 방법을 제공해야 한다. 본 논문에서는 새로운 사용자 레벨 시그널 처리 기법을 기반으로 한 효율적인 사용자 레벨 입출력 기법을 제안한다. 제안된 기법은 동적 가상 쓰레드 바인딩(dynamic virtual thread binding)과 스케쥴링 이벤트 업콜(scheduling event upcall)등의 다중 쓰레딩을 지원하기 위한 커널 구조를 이용하여 기존 시그널 처리의 문제점을 해결한다.

본 논문에서는 제안된 기법을 ARX 실시간 운영체계상에 구현하여 그 성능을 측정하였다. 실험결과 제안된 기법은 적은 부하로 외부에서 발생한 인터럽트를 사용자 프로세스에 예측 가능한 시간에 전달한다.

1. 서론

입출력에 있어서 높은 성능을 요구하는 응용 프로그램의 경우 사용자 레벨 입출력은 매우 효과적이다. 예를 들어, 네트워크 환경에서 시스템 메모리와 고속 네트워크 인터페이스사이에서 데이터 전송을 할 경우 커널의 관여 없이 사용자 레벨에서 DMA가 지원된다면 데이터 전송에서 상당한 성능 향상을 기대할 수 있다. 사용자 레벨 DMA를 사용한다면 문맥교환이나, 모드전환 그리고 데이터 복사에 드는 비용을 줄일 수 있다. 위와 같은 성능향상뿐만 아니라 사용자 레벨 입출력을 사용하면 다음과 같은 이점이 있다.

- 유연성: 디바이스 드라이버가 응용 프로그램의 일부분이므로 새로운 디바이스가 추가되었을 때 커널을 다시 만들 필요가 없다.
- 효율성: 응용 프로그램에 맞도록 디바이스 드라이버를 작성할 수 있으므로 작고 효율적인 디바이스 드라이버의 작성이 가능하다.
- 이식성: MS-DOS나 UNIX와 같이 매우 상이한 운영체계간에도 디바이스 드라이버를 쉽게 이식할 수 있다.

이러한 이점들 때문에 프로그래머들은 사용자 레벨 디바이스 처리를 이용하여 쉽게 디바이스 드라이버를 작성할 수 있다.

반면에 대부분의 기존 운영체계는 사용자 레벨 입출력을 지원하지 않는다. 이는 디바이스에 대한 접근이 커널에 의해 통제되지 않으면 잘못 짜여진 프로그램에 의해서 시스템의 보안이나 일관성이 깨질 수 있기 때문이다. 이러한 경우 운영체계는 같은 디바이스에 대하여 동시에 서비스를 요구하는 사용자의 충돌을 막아주는 디바이스 매니저 역할을 한다.

그러나 일반적인 컴퓨터 시스템에서도 어떤 디바이스는 특정 프로세스에 의하여 배타적으로 사용된다. 예를 들어 X 윈도우와 같은 클라이언트/서버 윈도우 시스템에서 그래픽 디스플레이 디바이스와 마우스 입출력 디바이스는 서버 프로세스에 의해 배타적으로 사용된다. 특정 디바이스가 주로 사용되는 내장 실시간 시스템의 경우 이러한 디바이스들이 대부분 특정 프로세스에 의해 배타적으로 사용되기 때문에 프로그래머는 사용자 레벨 입출력의 이점을 최대한 살릴 수 있다.

내장 실시간 시스템에서 사용자 레벨 입출력을 지원하기 위해서는 운영체계에서 외부 인터럽트를 예측 가능하고 효율적인 방법으로 사용자 인터럽트 핸들러에 전송해야 한다. 이러한 방법으로 기존의 UNIX 시그널 기법을 사용할 수 있다. 그러나 기존의 시그널 기법은 시그널을 받는 프로세스가 인터럽트 될 수 없는 모드에서 블록되면 시그널의 전달이 무한히 연기 되기 때문에 내장 실시간 시스템에서의 외부 인터럽트 전달 방법으로 사용되기 부적합하다. 본 논문에서는 위와같은 문제점을 해결하기 위해 새로운 시그널 전달 방법을 제안한다. 제안된 방법은 ARX 커널의 다중 쓰레딩 기법을 이용하여 적은 지연시간으로 예측 가능하게 시그널을 전달한다. 본 논문에서는 제안된 시그널 처리 기법을 ARX상에 구현하여 그 성능을 측정하였다.

2. 관련연구

사용자 레벨 입출력을 가능하게 하기 위해서 커널은 사용자 프로세스가 입출력 디바이스를 직접 접근할 수 있도록 해야한다. 이뿐만 아니라 입출력 디바이스의 인터럽트를 사용자 인터럽트 핸들러에게 전달할 수 있어야 한다. Linux같은 대부분의 UNIX 시스템은 사용자 레벨 인터럽트를 지원하지 않는다. 따라서 사용자 레벨 입출력을 위해서는 커널이 제공하는 디바이스 레지스터 접근 경로를 통하여 폴링에 기초한 입출력을 할 수 밖에 없다[2].

반면에, 실시간 기능을 제공하는 SGI의 Irix REACT의 경우 커널 디스패치 사용자 핸들러를 사용할 수 있게 해준다[4]. Irix의 커널 인터럽트 핸들러는 커널 모드와 사용자 모드간의 변환을 일으키고, 사용자 인터럽트 핸들러를 실행시킨다. 이 방법은 간단하기 때문에 적은 인터럽트 지연시간을 갖게 되지만 사용자 인터럽트 핸들러의 실행이 엄격하게 제한되는 심각한 단점을 가진다. 사용자 인터럽트 핸들러는 시스템 콜이나, 부동 소수점 연산을 할 수 없으며 잘못된 메모리 참조에 의한 페이지 폴트를 내서도 안된다.

커널 디스패칭의 대안으로 UNIX 시그널과 같은 이벤트 전달 방법을 사용할 수 있다. 시그널은 이벤트의 발생에 대해 커널이 생성하여 사용자에게 전달되는 방법이다. 시그널은 사용자가 등록한 시그널 핸들러를 통해 사용자 프로세스에 전달되어 처리된다. 그러나 기존의 시그널 기법은 내장 실시간 시스템에서 사용될 수 없는 단점이 있다. 첫째, 시그널을 받는 프로세스가 인터

럽트될 수 없는 모드에서 블록되어 있다면 시그널의 전달이 무한히 지연될 수 있다. 둘째, 기존의 시그널 전달방식은 커널의 시그널 마스크를 처리하기 위해 부가적인 모드전환과 문맥교환이 필요하여 오버헤드가 크다.

ARX 운영체계는 다중 쓰레딩 구조를 통해서 이런 문제를 해결하였다. 먼저, 외부 인터럽트에 의해 발생한 비동기적인 시그널을 처리하는 별도의 쓰레드를 두었다. 따라서 비동기적인 시그널은 쓰레드가 커널 모드에서 멈추어 있는 경우라도 즉시 전달되고 처리될 수 있다. ARX 커널은 새로 동작하는 쓰레드에 대해서 동적으로 커널 스택을 할당하도록 설계되었기 때문에 이러한 것이 가능하다. 둘째로, ARX 커널은 시그널을 단지 사용자 레벨 스케쥴러에 전달하고, 사용자 레벨 스케쥴러가 핸들러 쓰레드를 동작시키도록 한다. 따라서, ARX의 시그널 처리는 사용자 레벨에서 구현될 수 있고 사용자 레벨 입출력에 대해서 최적화 될 수 있다. 실험 결과에서 기존의 시그널 기법을 사용한 것에 비해 상당한 수준의 성능 향상이 이루어 진 것을 확인할 수 있다.

3. ARX 실시간 운영체계

ARX는 실시간 및 멀티미디어 응용을 위해 개발한 운영체계이다[3]. ARX는 두 단계 스케쥴링, 다중 쓰레딩, 효율적인 사용자 레벨 입출력등을 지원한다. ARX에서의 스케쥴링은 커널은 프로세스를, 사용자 레벨 스케쥴러는 쓰레드를 스케쥴링하는 두 단계 스케쥴링을 한다. 커널은 프로세스를 프로세서 용량 할당방식에 의해 스케쥴링 한다. 반면 사용자 레벨 스케쥴러는 응용 프로그램에 맞는 다양한 방식으로 쓰레드를 스케쥴링 할 수 있다. 사용자 레벨에서 실시간 스케쥴링이 가능하게 하기 위해 ARX는 동적 가상 쓰레드 바인딩(dynamic virtual thread binding)과 스케쥴링 이벤트 업콜(scheduling event upcall)을 제공한다.

사용자 레벨 쓰레드의 경우 커널이 쓰레드의 존재를 모르기 때문에 하나의 커널 스택만이 프로세스에 할당된다. 따라서 쓰레드가 커널 내에서 블록당하면 프로세스내에 수행할 수 있는 다른 쓰레드가 있음에도 불구하고 커널은 프로세스 전체를 블록시킨다. 이러한 문제를 해결하기 위하여 ARX는 사용자 레벨 쓰레드가 커널에서 블록당할 경우 동적 가상 쓰레드 바인딩을 통하여 프로세스에 새로운 커널 스택을 할당하여 다른 쓰레드가 다시 커널로 들어올 수 있도록 한다.

스케쥴링 이벤트 업콜은 커널에서 발생한 이벤트를 사용자 레벨 스케쥴러에 전달하여 사용자 레벨에서 실시간 스케쥴링이 가능하도록 한다. 이벤트가 발생하면 커널은 이를 이벤트 큐에 넣고, 사용자 레벨 스케쥴러를 호출한다. 사용자 레벨 스케쥴러는 큐에 들어온 이벤트의 종류에 따라 이벤트를 처리한다. 현재 ARX 커널은 `BLOCK`, `WAKEUP`, `TIMER`, `SIGNAL`, `EXIT`, `RESUME`의 여섯가지 이벤트를 지원한다.

커널과 사용자 레벨 스케쥴러와의 이벤트 전달에 따르는 오버헤드를 줄이기 위해 ARX에서는 스케쥴러 제어 블록(scheduler control block)을 커널과 프로세스간의 공유 메모리 상에 두어 이를 해결한다.

4. ARX의 사용자 레벨 시그널 처리

4.1 시그널의 종류

ARX는 실시간 확장을 포함한 POSIX 1003.1[1] 표준의 시그널 큐와 신뢰성 있고 정확한 시그널 전달 기법을 지원한다. 1번부터 20번까지의 시그널은 표준에 의해 각각 용도가 정해져 있고 나머지 21 (`SIGRTMIN`)에서 31 (`SIGRTMAX`)까지를 실시간 응용 프로그램을 위해 사용한다.

다중 쓰레딩 기법을 쓰지 않는 프로세스에 대해서 ARX 커널은 다른 UNIX와 동일하게 커널이 직접 해당 시그널 핸들러를 호출한다. 반면, 다중 쓰레딩 기법을 사용하는 프로세스에 대해서는 커널은 단지 해당 시그널을 사용자 레벨 스케쥴러에게 알려주고 사용자 레벨 스케쥴러가 시그널 핸들러를 호출한다.

시그널은 동기적 시그널과 비동기적 시그널로 나눌 수 있다. 동기적 시그널은 `SIGFPE`, `SIGILL`, `SIGSEGV`, `SIGV`등으로 프로세스의 수행 중 나타나는 예외에 의해 시그널이 생긴 경우이다. 비

```
void signal_thread(int signum)
{
    while (1) {
        q = find signal in the signal queue;
        if (q) {
            remove q from the signal queue;
            if (need siginfo)
                (sighandler[signum])(signum, q);
            else
                (sighandler[signum])(signum);
        } else
            thread_suspend(thread_self());
    }
}
```

그림 1: 시그널 핸들링 쓰레드의 구조

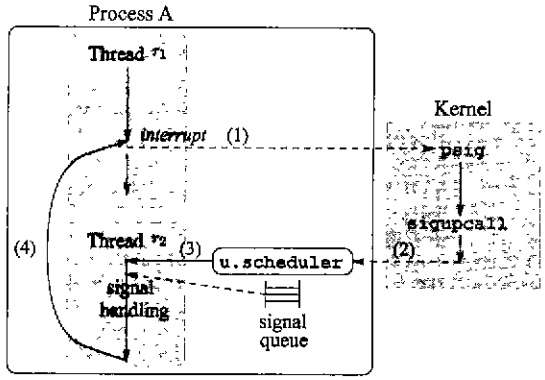

그림 2: ARX에서의 비동기적 시그널의 처리

동기적 시그널은 외부적인 요인에 의해 발생하는데, 위의 4가지 시그널을 제외한 시그널은 모두 비동기적이다.

4.2 동기적 시그널과 비동기적 시그널의 처리

ARX는 동기적 시그널과 비동기적 시그널을 다르게 처리한다. 동기적 시그널에 대해서는 시그널을 생성한 쓰레드가 직접 이를 다룰 수 있어야 한다. 그러므로 각각의 쓰레드는 자신만의 고유한 시그널 마스크를 가진다. 반면 비동기적 시그널은 그림 1의 구조를 가지고 있는 별도의 핸들러 쓰레드가 이를 처리한다. 사용자 프로세스가 `sigaction()` 함수를 부르는 시점에 이 핸들러 쓰레드가 생성된다. 핸들러 쓰레드는 시그널 큐에서 시그널을 하나 읽어오고 필요한 핸들러 함수를 부른다. 동기적 시그널과는 달리 비동기적 시그널에 대해서는 모든 쓰레드가 프로세스의 시그널 마스크를 공유한다.

4.3 시그널의 전달과 핸들러의 호출

UNIX에서는 커널 모드에 있는 프로세스가 시그널의 발생을 알 수 있는 시점이, (1) 프로세스가 스케쥴러에게 선택되어 수행이 다시 시작되는 시점, (2) 인터럽트에서 돌아오는 시점, (3) 시스템 콜을 끝내고 돌아오는 시점의 세 가지로 제한되게 된다. 반면 ARX에서는 `SIGNAL` 업콜을 쓰기 때문에 현재 쓰레드의 상태와 무관하게 발생 즉시 시그널을 전달받을 수 있다.

커널은 사용자 레벨 스케쥴러의 스케쥴러 제어 블록에 있는 시그널 큐에 해당 시그널에 대한 정보를 넣고, 해당 시그널 마스크를 지운 후 `SIGNAL` 업콜을 통하여 사용자 레벨 스케쥴러를 호출한다. 사용자 레벨 스케쥴러는 시그널 큐에서 한 엔트리를 꺼내고, 동기적 시그널인지 비동기적 시그널인지 판단한다. 비동기적 시그널인 경우 핸들러 쓰레드를 수행시키고, 동기적 시그널인 경우는 해당 쓰레드가 예외 처리를 하도록 한다.

5. 사용자 레벨 입출력 인터페이스와 구현

ARX 커널은 프로그래머가 각각의 디바이스에 대하여 가장 적합한 입출력 처리 기법을 사용할 수 있도록 하기 위하여 커널 디

스패치 사용자 인터럽트 핸들러와 시그널 기반 사용자 인터럽트 핸들러 모두를 제공한다. 커널 디스패치 사용자 인터럽트 핸들러는 디바이스가 짧은 대기시간을 요구하고 핸들러의 수행시간이 매우 짧은 경우에만 사용한다. 그러나 이 경우 핸들러는 커널의 특수한 문맥위에서 수행되므로 인터럽트 핸들러는 매우 제한적이다. 따라서 안전하고 유연성있는 인터럽트 핸들러나 소수점 연산과 같은 복잡한 연산을 필요로 하는 경우에는 시그널 기반의 사용자 인터럽트 핸들러를 작성해야 한다. 다음은 ARX에서 구현된 사용자 레벨 입출력의 프로그래밍 인터페이스(API)를 기술한다.

사용자 레벨 입출력을 위해서 우선 입출력 디바이스에 대한 접근 권한을 요청해야 한다. 다음의 함수들이 접근 권한을 요청하고 해제하기 위하여 사용된다.

- **uio_request(ioa_start, ioa_end, [type])**: 이 함수는 **ioa_start**에서 **ioa_end**까지의 입출력 주소를 접근할 수 있는 권한을 사용자 프로세스에게 부여한다. **type**은 **IOMAP**이나 **MEMORYMAP**중 하나의 값으로 주어야 한다. 이들 주소가 memory-mapped 디바이스에 해당한다면 반드시 페이지단위로 정렬되어 있어야 한다. 커널은 사용자 프로세스에 권한이 부여된 디바이스 주소를 관리하여 다른 프로세스와의 충돌이 있는지 검사한다. 충돌이 없다면 권한이 부여된 주소를 나타내는 정수 **handle**을 리턴하고 그렇지 않으면 0을 리턴한다. ARX 커널은 마이크로 프로세서의 메모리 보호 메카니즘을 사용하여 입출력 주소 보호를 구현한다. 만약 몇 개의 다른 디바이스의 주소가 한 페이지에 매핑되었다면 어떤 프로세스가 다른 프로세스에게 할당된 디바이스의 주소공간을 접근할 수도 있다. 따라서 memory-mapped 디바이스들이 페이지단위로 정렬되어 있어야 한다.
- **uio_release(handle)**: 이 함수는 **handle**에 해당하는 디바이스 주소에 대한 접근 권한을 해제한다.

두 번째로 입출력 디바이스에 대하여 사용자 인터럽트 핸들러를 설치해야 한다. 다음의 함수가 사용자 레벨에서 인터럽트를 제어하는데 사용된다.

- **uio_int_request(irq, handler, type, flag)**: 이 함수는 사용자 레벨 인터럽트 핸들러를 등록한다. **type**은 **KERNEL_DISPATCH**나 **USER_DISPATCH**중의 하나이다. **USER_DISPATCH**에 대해 이 함수는 인터럽트에 **SIGRTMIN**과 **SIGRTMAX**사이의 시그널 번호를 할당한다. 만약 호출하는 프로세스가 다중 쓰레드 응용 프로그램이라면 이 함수는 핸들러 쓰레드도 함께 생성한다. **flag**가 0이라면 인터럽트를 금지한 뒤에 리턴한다. 성공적으로 수행되면 **irq** 번호, 시그널 번호, 쓰레드 아이디를 가지고 있는 **irq_info** 구조체의 포인터를 리턴하고 그렇지 않으면 0을 리턴한다. 쓰레드 아이디는 프로세스가 핸들러 쓰레드의 속성을 제어하기 위해 필요하다.
- **uio_int_release(irq, flag)**: 이 함수는 등록된 사용자 인터럽트 핸들러의 등록을 없앤다. **flag**가 1이면 관련된 핸들러 쓰레드도 함께 없앤다.
- **uio_int_enable(irq)**, **uio_int_disable(irq)**: 이 함수는 인터럽트를 가능하게 하거나 금지시킨다. 만약 등록된 핸들러가 시그널 기반 핸들러라면 실제 인터럽트를 막지 않고 단순히 관련된 시그널을 마스크한다.

6. 성능

ARX 사용자 레벨 입출력의 성능은 절대적으로 ARX 시그널 처리 성능에 달려있다. 본 논문에서는 UNIX 시그널에 대한 ARX 시그널의 성능 향상을 보이기 위하여 ARX와 UNIX에서의 시그널 처리 지연시간을 측정하였다. 시그널 처리 지연시간은 커널 인터럽트 핸들러가 수행을 시작할 때부터 시그널 핸들러가 수행을 시작할 때까지의 시간으로 정의하였다.

UNIX 시스템으로서 Linux ver. 2.0.31을 선택하였다. Linux에서는 사용자 레벨 인터럽트가 제공되지 않으므로 임의의 인터럽트를 시그널과 연관시킬 수가 없다. 따라서 본 논문에서는 alarm 시그널을 사용하여 클럭 인터럽트로 부터의 지연시간을 측정하였다. 그리고 실제 시그널 처리 대기시간을 측정하기 위해 클럭 인터럽트 핸들러의 수행시간을 제하였다.

표 1: Linux 와 ARX상에서의 시그널 처리 지연시간

	Linux signal	ARX signal	Performance increase
In user-mode	19.15	17.07	11 %
In kernel-mode	30.80	17.07	45 %
Return from handler	11.97	1.85	85 %

(unit: μs)

실험을 위하여 간단한 오디오 플레이어 프로그램을 작성하여 100MHz 펜티엄 프로세서, 256KB 이차 캐쉬, 32MB EDO DRAM, Sound Blaster 카드를 장착한 PC에서 수행하였다. 오디오 플레이어 프로그램은 주기적으로 4KB의 오디오 데이터를 사용자 레벨 DMA를 사용하여 오디오 디바이스에 전달한다. 디바이스는 각 DMA 종료후 인터럽트를 발생시키고 플레이어는 다음 DMA를 시작한다.

표 1은 ARX와 Linux에서 측정한 시그널 처리 지연시간을 보여준다. 전달받은 프로세스가 사용자 모드와 커널 모드에서 수행되는 두 가지 경우에 대하여 지연시간을 표시하였다. 표 1에서 볼 수 있듯이 Linux 비하여 ARX가 훨씬 작은 시그널 지연시간을 가진다. 특히 ARX의 시그널 처리 지연시간이 전달받는 프로세스의 상태에 무관하게 일정한 반면 Linux는 61%까지 크게 변화한다. 이것은 ARX에서는 인터럽트가 전용의 핸들러 쓰레드에 의하여 서비스되기 때문이다. 이러한 결과는 ARX의 사용자 레벨 입출력이 내장 실시간 시스템에 필요한 예측 가능한 시그널 처리를 할 수 있음을 보여준다. 표 1의 마지막 줄은 시그널 핸들러로부터 리턴하는데 걸리는 시간을 보여준다. Linux 시그널 핸들러는 시스템 콜을 통하여 커널의 시그널 마스크를 복귀시켜야 하기 때문에 ARX보다 훨씬 더 큰 오버헤드를 가진다.

7. 결론

본 논문에서는 ARX 실시간 운영체제의 사용자 레벨 시그널 기법에 기반한 효율적인 사용자 레벨 입출력을 제시하였다. 제안된 기법은 다중 쓰레드를 지원하기 위한 커널 구조를 이용하여 적은 지연시간을 가지는 예측 가능한 시그널 전달을 가능하게 한다. 제안된 기법은 업콜을 통하여 커널이 시그널을 사용자 레벨 스케쥴러에 전달하고 사용자 레벨 스케쥴러가 핸들러 쓰레드를 디스패치하게 하며, 각 비동기적 시그널을 핸들러 쓰레드가 전담하여 처리하도록 하여 기존의 시그널 구현의 문제점을 해결한다.

본 논문에서는 제안된 기법을 ARX 실시간 운영체계상에 구현하여 실험을 통하여 그 성능을 확인하였다. 실험 결과 제안된 기법은 기존의 시그널 처리 기법에 비해 월등한 성능 향상을 보여준다.

참고 문헌

[1] Institute for Electrical and Electronic Engineers. POSIX Part 1: System application program interface, 1996.

[2] M.K. Johnson. The Linux kernel hackers' guide, 1997.

[3] 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.

[4] Silicon Graphics, Inc. *REACT™* in *IRIX™* 6.4. Technical report, Jan 1997.