리눅스 디바이스 드라이버 오류 유형

류진영*, 장승주*, 임채덕**, 마유승**
*동의대학교 컴퓨터공학과, ETRI 임베디드 S/W 연구단
e-mail : ros207@naver.com, sjjang@deu.ac.kr, {cdlim,ysma}@etri.re.kr,

Error-type in Linux Device Driver

Jin-Young Ryu*, Seung-Ju Jang*, Chae Duk Lim**, Yu Sung Ma**
*Dept. of Computer Engineering, Dong-Eui University, **ETRI

요 약

현재 오픈소스인 리눅스를 기초로 많은 프로젝트들이 진행되고 있다. 그러나 하드웨어의 지식과 밀접한 관계를 가지고 있는 리눅스 디바이스 드라이버 개발 분야는 접근이 어려워 개발자들 역시 주의하고 있는 실정이다. 본 논문은 리눅스 디바이스 드라이버의 오류 유형을 토대로 오류 없이 정상적으로 동작하는 리눅스 디바이스 드라이버 개발을 위한 테스트 모듈 개발에 목적을 둔다.

1. 서론

리누스 토발즈(Linus Tovalds)라는 핀란드의 헬싱키 대학에 다니던 한 젊은 학생에 의해 시작된 리눅스는 어느덧 커널 2.6.12 까지 발전하며 그 동안 많은 기능과 하드웨어를 지원하게 되었다.

타 운영체제와 비교하여 오픈소스라는 이점에 힘입어 많은 자발적인 개발자들이 수없이 많은 프로그램들을 제작해 주었으며, 이에 타 운영체제와는 달리 많은 성능 좋은 프로그램들과 알고리즘들이 개발되었다. 그러나 아직도 리눅스 디바이스 드라이버 개발 부분은 취약하다.

물론 이는 개발자들만의 문제가 아니다. 디바이스 제작업체에서 리눅스 드라이버를 개발하지 않거나, Core 부분을 공개하지 않아 리눅스 디바이스 드라이버 개발자들은 다른 어느 개발자보다 훨씬 힘겹게 개발하고 있다.

본 논문은 리눅스 디바이스 드라이버의 오류 유형을 구조별, 유형별로 각각 제시하여 오류 없이 정상적으로 동작하는 디바이스 드라이버 개발을 위한 테스트 모듈 개발에 목적을 둔다.

본 논문의 구성은 2 장에서 디바이스 드라이버의 일반적인 에러유형을, 3 장에서 실제 디바이스 드라이버 개발 시 오류 유형을 체계적으로 분류하고, 4 장에서 결론을 각각 서술한다.

2. 디바이스 드라이버의 일반적인 오류 유형

디바이스 드라이버에서 가장 빈번하게 발생하는 오류 유형들에 대해서 간단하게 설명한다. 다음 <표 1>은 디바이스 드라이버의 일반적인 오류 유형이다.

<표 1> 디바이스 드라이버의 일반적인 오류 유형

오류유형	설명
EINTR	프로세스 수행 도중에 커널로부터 시그널을 받았을 때 사용자가 프로세스를 중단했거나 커널에서 강제로 중단시킬 경우
ENOMEM	kmalloc()을 통한 커널 메모리 영역 할당시 커널 메모리 공간에 여유가 없을 경우
ENXIO	잘못된 부장치를 호출했거나 사용할 수 없는 주소가 참조되었을 경우
ENODEV	주 장치에 딸린 부장치가 범위를 넘어서 더 이상 장치를 할당할 수 없을 경우
EAGAIN	일시적으로 드라이버에서 디바이스를 사용할 수 없거나 다른 루틴에서 이 디바이스를 사용 중일 때 잠시 기다린 후에 다시 시도할 수 있도록 할 경우
EBUSY	해당 디바이스를 이미 다른 루틴에서 사용중인 경우
ENOSYS	ioctl 등의 루틴에 없는 함수를 부를 경우
EIO	장치를 등록할 수 없거나 디바이스가 정상작동을 하지 않고 있을 때 등 포괄적인 상황인 경우

위 <표 1>은 디바이스 드라이버와 관련하여 가장 빈번하게 발생하는 오류이다. 특히 EIO, ENOMEM 등은 시스템을 정지시킬 만큼 치명적인 오류이므로 주의해야 하며, EAGAIN 은 커널에서 처리를 하고 드라이버 열기를 재시도 할 수 있다.

본 논문은 한국전자통신연구원 2005 년도 연구비 지원에 의하여 연구되었습니다.

3. 실제 디바이스 드라이버 개발 오류

3.1. 디바이스 드라이버 초기화 오류

디바이스 드라이버 초기화 함수에서는 다음 <표 2>와 같은 과정을 수행하게 된다.

<표 2> 드라이버 초기화 함수의 처리 내용

No	처리 과정
1	드라이버 초기화에 필요한 인자의 파싱
2	디바이스 드라이버 등록
3	file_operation 구조를 이용한 디바이스 접근
4	디바이스가 사용할 번지 주소가 유효한지 검사
5	인터럽트 사용시 요청
6	디바이스가 실제로 장착되어있는지 검사
7	사용할 데이터/주소 메모리 영역 할당

드라이버 초기화 함수에서는 드라이버에 특정 주소를 값으로 보내거나, BIOS 정보를 다운로드 및 지원 디바이스 중에서 장착된 디바이스를 구별하는 등의 작업을 수행 한다.

이러한 경우 점검해 보아야 할 부분은 지정한 번지를 파싱하는 부분(check_region)과 인터럽트가 유효한 것인지를 검사하는 부분(request_irq)이다

위 <표 2>에 나타난 과정 중 설정 값을 잘못 지정했거나, 장착된 디바이스보다 더 많은 디바이스를 지정했거나 디바이스가 작동하지 않는 경우 오류 값을 검출해 보아야 할 것이다.

디바이스 드라이버 초기화 시 디바이스 드라이버에 인식 및 접근 하기 위해서 다음 <표 3>과 같은 함수를 사용하기도 한다..

<표 3> 디바이스 드라이버 초기화 시 접근 함수

함수명	용도
outb/inb outw/inw	물리 주소에 쓰기/읽기 함수
readb/writeb	memory mapped I/O 장비에 읽고 쓰는 함수
memcpy_toio memcpy_fromio	특정 주소에 데이터를 인자로 준 바이트만큼 읽고 쓰는 함수

위 <표 3>의 함수 인자가 잘못 설정될 경우 임의적인 에러가 발생한다. 또한 특히 주의해야 할 점은 x86 Architecture 에서는 디바이스의 물리적인 주소와 memory mapped 주소가 동일하지만 다른 Architecture 에서는 물리적인 주소와 memory mapped 주소가 차이가 있을 가능성이 있으므로 호환성을 위해서 상당히 주의해야 한다.

3.2. 디바이스 드라이버 삭제 오류

디바이스 드라이버 삭제 함수에서는 초기화 함수에서 할당 받은 부분을 해제하는 역할을 수행한다. 일반적인 자원 반납은 다음 <표 4>와 같다

<표 4> 디바이스 드라이버 삭제 시 자원의 반납

할당 함수	관련 영역	반납 함수
kmalloc()	Kernel Memory	kfree()
request_region()	Kernel Address	release_region()
request_irq()	Interrupt	free_irq()
register_dev()	Major Minor Number	unregister_dev()

디바이스 드라이버는 커널에 종속적인 프로그램이므로 미 반납된 자원을 자동으로 처리하지 못한다. 만약 자원을 제대로 반납하지 않고 모듈을 삭제한다면 반납되지 못한 영역의 포화에 의해 시스템 전체에 문제를 일으킨다.

3.3. 디바이스 드라이버 open 오류

디바이스 드라이버를 커널 모듈로 컴파일하여 동적으로 사용할 경우, 만약 open 및 read/write 루틴의 수행 중 'rmmod'를 이용하여 모듈을 삭제한다면 디바이스 드라이버 수행 및 정상적 중단이 불가능해져 커널이 정지한다. 그러므로 디바이스 드라이버 루틴을 수행 중인 프로세스가 있으면 이를 알려주는 변수를 증가(MOD_INC_USE_COUNT)시키고 모듈 삭제 함수에서 감소(MOD_DEC_USE_COUNT)시켜야 한다.

커널이 open()을 호출할 때 주 장치번호를 토대로 호출하므로 내부에서 동작을 위해서 MINOR(inode->i_rdev)를 통해 정확한 부 장치번호를 구해야 한다.

또한 커널은 부장치가 다른 루틴에서 사용 중인지 검사하지 않고 open()이 발생하면 무조건 이동하므로 다른 루틴에서 사용하지 못하게 하거나 사용이 끝나는 시점을 알려주기 위해 device 구조체 안에 busy 필드를 정의 및 사용한다. 이때 각 포트로 필요한 정보와 어느 카드의 포트인지 알려주는 정보 및 포트의 정의 정보를 포함 시켜야 한다.

3.4. 디바이스 드라이버 read/write 오류

디바이스 드라이버의 read/write 는 커널 메모리 영역의 사용을 의미하므로 memcpy 를 사용할 수 없다.

또한 한번에 buf 로 넘어오는 데이터의 최대값은 4096 바이트이며 이를 넘어서는 안된다.

그리고 read/write 에서도 부 장치번호를 체크해야 한다. 만약 사용할 수 없는 부 장치에 접근 요구가 오면 오류 값을 리턴해야 한다.

문자디바이스일 때 한 번에 한 바이트씩 보내야 하며, 블록 디바이스라면 자체적으로 데이터를 보낼 수도 있고 block_write 라는 블록 디바이스 공용 루틴을 사용할 수도 있다.

쓰기나 읽기 도중 시그널을 받았으면 -EINTR 값을 사용한다.

그 외에 오류가 발생하였으나 커널 쪽에서 재시도를 하기를 원한다면 그 때까지 읽고 쓴 데이터 양을 인수로 하여 리턴한다.

만약 커널 쪽에서 읽기 및 쓰기를 재시도 할 수 없는 오류면 커널에서 release()를 스스로 호출하게 된다.

3.5 디바이스 드라이버 release 오류

디바이스 드라이버 release()에서는 open()에서 할당한 값들을 역으로 복구시켜야 한다. 아래 <표 5>는 release() 시 복구해야 할 주요사항이다.

<표 5> release 시 복구해야 할 주요사항

No	복구 내용
1	메모리영역을 할당 받아서 사용했다면 반납
2	부장치에 대해서 busy 세팅을 했다면 이를 해제
3	모듈을 사용했다면 MOD_DEC_USE_COUNT 를 실행해서 모듈 사용값을 감소
4	인터럽트를 할당 받았다면 해제
5	리턴값의 체크(kernel 2.2.x 이상)
6	차후의 open()이 가능하도록 복구

위 <표 5>의 내용 중 가장 중요하게 처리해야 할 부분은 '차후의 open()이 가능하도록 복구'부분이다.. open()이 여러 번 호출되더라도 사용할 변수나 값들이 최초의 open()이 실행되었던 경우와 동일해야 하므로 open(), read(), write()에서 사용된 변수들을 조사해 원래대로 복구시켜야 한다. 그렇지 않을 경우 사용할 수 있는 디바이스라도 단 한번 사용하고 나서 더 이상 쓸 수 없는 오류가 발생한다.

3.6. 디바이스 드라이버 메모리 오류

3.6.1 커널 메모리 영역 할당 오류

디바이스 드라이버의 경우 커널 메모리를 사용하므로 kmalloc()/kfree()를 통해 메모리를 할당 및 해제해야 한다. 이때 상황에 맞는 FLAG 를 설정이 필요하다. 아래 <표 7>은 kmalloc() 할당 GFP_FLAG 의 종류이다.

<표 7> kmalloc() 할당 GFP_FLAG 의 종류

GFP_ATOMIC	GFP_HIGHUSER
GFP_BUFFER	__GFP_DMA
GFP_KERNEL	__GFP_HIGHMEM
GFP_NOIO	__GFP_REPEAT
GFP_NOFS	__GFP_NOFAIL
GFP_USER	__GFP_NORETRY

kmalloc()이 한 번에 할당할 수 있는 메모리 양은 4KB 페이지로 제한되어 있으며 넘을 경우 오류가 발생한다. 그 이상의 메모리를 할당하기 위해서는 vmalloc()을 사용해야 하며 64MB 까지 할당 가능하다.

3.6.2 커널 메모리 영역 사용 오류

할당 받은 메모리는 이전 내용이 남아있으므로 사용 전 반드시 memset()으로 초기화 시켜야 한다.

메모리간의 데이터를 복사할 경우 memcpy()를 사용하고 커널 영역과 사용자 영역 간의 데이터를 복사할 경우 copy_from_user/copy_to_user()를 사용해야 한다. 잘못 사용하면 사용자 메모리 영역의 주소와 커널 메모리 영역의 주소가 다르기 때문에 오류가 발생한다.

3.7. 디바이스 드라이버 동기화 오류

같은 자원에 대해서 동시에 여러 프로세스가 접근한다면 충돌이 일어난다. 이를 해결하기 위해 세마포어를 사용한다. 여러 부 장치가 경쟁적으로 한 자원을 접근한다면 다음의 (그림 1)과 같이 앞뒤로 세마포어를 설정한다

```
down(&card->semaphore);
중요루틴 실행
up(&card->semaphore);
```

(그림 1) 세마포어의 사용

특수한 경우 사용자가 만든 세마포어를 사용할 수도 있으나 이것이 동기화 문제를 일으키지 않는지 점검해 보아야 한다.

인터럽트와 관련하여 cli()를 사용하여 인터럽트를 금지 시키고 중요한 작업을 한 다음에 sti()로 인터럽트를 가능하게 만든다. 이때 디바이스 드라이버의 호출 전후에 FLAG 상태가 변화한다면 문제가 발생할 수 있다. 그러므로 FLAG 보존을 위해 save_flags(flags); cli(); sti(); restore_flags(flags); 형식을 사용한다. 함수가 조건문 등으로 분기할 때 restore_flags()가 수행되지 않는 등의 오류가 있다면 시스템이 정지할 수 있다.

디바이스 드라이버에 어떤 조작을 하고 일정 시간을 기다릴 때 절대적인 대기 함수 __delay()를 사용하는데 문맥교환이 일어나지 않아 실행되는 동안 시스템이 정지해 있게 된다. 일반 작업에 사용하면 효율이 엄청나게 떨어지므로 절대적인 시간이 필요한 초기화 작업 같은 경우를 제외하고는 이 함수를 사용해서는 안 된다. 일반 작업에서 대기 함수는 다음의 (그림 2)의 함수를 사용한다.

```
current->state = TASK_INTERRUPTIBLE;
current->timeout = jiffies + HZ * second;
schedule();
```

(그림 2) 일반적인 대기 함수

위 (그림 2)는 인터럽트를 허용하면서 second 후에 자동적으로 깨어나게 하는 대기 함수의 예이다.

대기하고 있는 동안에 다른 루틴에서 조건을 변화시키고 대기 프로세스를 깨워 준다면 조건이 만족하기 전에 깨어났다가 다시 잠드는 오버헤드를 줄일 수 있다. 이를 위해 sleep/wake_up 함수를 사용한다..

interruptible_sleep_on/wake_up_interruptible 함수는 인터럽트를 사용하는 드라이버에서 디바이스에 인터럽트를 걸어 놓고 잠들면 디바이스가 처리를 끝내고 인터럽트를 걸어 깨워 주는 경우에 많이 사용한다. 그러나 이 함수도 schedule 과 마찬가지로 시그널을 받으면

무조건 깨어 나기 때문에 꼭 상태를 체크해야 한다.

sleep_on()으로 잠들면 타임아웃이 지나거나 시그널이 발생해도 깨어 나지 않고 오로지 wake_up()만이 깨울 수 있다. wake_up()가 수행되지 않는다면 시스템이 교착상태에 빠질 수 있다.

3.8. 디바이스 드라이버 인터럽트 오류

인터럽트를 사용하기 위해서는 일단 그 인터럽트가 유효한지 검사해야 한다. 이때 request_irq()를 사용한다. 다음 <표 8>은 request_irq()의 세 번째 인자로 인터럽트 레벨을 나타낸다.

<표 8> 인터럽트 레벨

인자	기능
SA_INTERRUPT	다른 인터럽트를 금지
0	다시 인터럽트가 발생가능
SA_SHIRQ	PCI 디바이스가 인터럽트를 공유

위 <표 8>의 인터럽트 레벨을 잘못 사용하면 오류가 발생한다.. 특히 SA_INTERRUPT 의 경우 문맥교환, 재인터럽트, 프로세스 블록킹이 일어나지 않는다.

인터럽트 내에서는 메모리를 할당 받는 kmalloc()등을 사용할 수 없다. kmalloc()은 할당에 실패했을 경우 스스로 필요한 시간만큼 기다리고 재시도한다면 시스템을 정지 시킬 가능성이 있다. 또한 인터럽트 중에 schedule 같은 프로세스 블록 함수도 사용할 수 없다. 인터럽트라는 특성상 블록이 가능하지 않기 때문이다.

save_flags(flags); cli(); sti(); restore_flags(flags);도 사용할 수 없다. 인터럽트 수행 도중 또 다시 인터럽트가 걸릴 경우 오류가 발생할 수도 있다. 인터럽트는 한번에 수행되고 그 동안 방해 받아서는 안 되는 루틴이기 때문에 이 경우 수많은 오류의 발생이 가능하다.

인터럽트를 주고 받을 때의 규약을 정확히 지키지 못한다면 디바이스가 인터럽트를 돌려 주지 못하는 오류가 발생한다. 만약 다른 운영체제에서 정상적으로 사용하던 하드웨어일 경우 인터럽트 규약만 정확히 지킨다면 리눅스에서 아무 오류없이 사용할 수 있다.

3.9 시그널 및 타임 아웃 관련 오류

커널은 프로세스에게 각종 시그널을 줄 수 있다. 여러 시그널을 받은 프로세스는 가능한 신속하게 작업을 끝내기 위해서 블록되지 않으므로 schedule 이 무시되기 때문에 코딩을 할 때 이 것을 염두에 두고 깨어났을 때 정상상태에서 깨어난 것인지 시그널을 받았는지 구별해서 동작하게 해야 한다.

```
if(current->signal & ~current->blocked) //signal
else // no signal
```

(그림 3) 프로세스 wake 상태 판별

위 (그림 3)은 프로세스 wake 상태 판별을 위한 코드이다. 프로세스가 정상상태에서 깨어난 것인지 시그널을 받았는지 구별해서 동작하도록 구현되어 있다.

대기 후에 활성화된 프로세스는 프로세스가 깨어난 상태에 대한 점검이 필요하다. 시그널에 의해서 깨어난 프로세스는 작업을 중단하고 자신을 부른 상위 루틴으로 복귀해야 한다. 물론 세마포어에 대한 정리작업 같은 중요한 작업은 반드시 수행해야 그 다음 프로세스가 영향을 받지 않는다.

만약 타임아웃이 지나 활성화되었고 시그널이 없다면 디바이스에 이상이 있는 것인지 타임아웃 값이 충분하지 않았는지 확인해야 한다. 타임아웃 값이 적다면 늘여 주고 디바이스가 인터럽트를 돌려 주지 못한 것이라면 이에 따른 처리를 해야 한다.

3.10 기타 프로그래머의 실수

커널 프로그램의 경우 Standard Library 를 사용할 수 없으며, 잘못된 코드 작성 역시 오류를 일으킨다.

4. 결론

본 논문에서는 디바이스 드라이버의 개발 관련 오류 유형에 대하여 연구하였다. 디바이스 드라이버 개발 시 가장 빈번히 발생하는 오류 유형을 정리하여 디바이스 드라이버의 일반적인 오류 유형에 대하여 연구하였다.

실제 디바이스 드라이버 개발 오류 부분에서 디바이스 드라이버의 구조별로 “초기화 오류, 삭제 오류, open 오류, read/write 오류, release 오류”라는 다섯 부분으로 세분화하여 오류 유형에 대한 설명 및 해결책을 제시하였다. 서로 관련된 오류 유형별로 “메모리 오류, 동기화 오류, 인터럽트 오류, 시그널 및 타임 아웃 관련 오류”라는 네 부분으로 세분화하여 오류 유형에 대한 설명 및 해결책을 제시하였다. 마지막으로 프로그래머의 실수에 대하여 설명하였다.

본 논문에서 제안하는 오류 유형을 이용하여 디바이스 드라이버 개발에서 발생 가능한 오류를 사전에 점검하여 디바이스 드라이버의 안정성을 도모할 수 있다.

참고문헌

[1] A. Rubini, J. Corbet, Linux Device Driver, 2nd Edition, O’Reilly, 2001.
[2] David A Rusling, The Linux Kernel, http://linuxkernel.net/kernel/tlk/origtext/tlk-0.8-3.pdf
[3] 유영창, “IT EXPERT 리눅스 디바이스 드라이버”, 한빛미디어, 2004
[4] Peter Jay Salzman, Ori Pomerantz, Linux Kernel Module Programming Guide, http://www.faqs.org/docs/kernel/
[5] How to NOT write kernel driver, http://people.redhat.com/arjanv/olspaper.pdf
[6] Kernel development, http://lwn.net/Articles/22699/
[7] 리눅스 커널 디바이스 드라이버 만들기, http://www.kesl.org/pds/data/linuxdevicedriver.doc