The Linux Kernel Module Programming Guide

by Peter Jay Salzman, Micheal Burian, Ori Pomerantz

*Translated by **YoonMin Nam** in **PuKyung National University.***
Lab of Information Security and Internet Application** & **CERT-IS

:: zkdnsxjaos@gmail.com

Table of Contents

Chapter1. Introduction

1.1 What is A Kernel Module?

여러분들은 커널모듈을 작성하고 싶어 한다. 이미 당신은 프로세스로 프로그램을 실행하기 위한 몇 개의 프로그램들을 c를 사용해서 작성을 했었을 것이고, 이제 당신은 하나의 single pointer가 core dump와 file system을 돌아다니는 실제 활동영역을 알고 싶다..

정확하게 커널모듈이 뭘까? 모듈은 사용자의 혹은 커널의 요구로 읽혀지거나 아니면 없어지는 코드들로 이루어진 프로그램의 어떤 한 조각을 의미한다. (역주: 커널은 하나의 component와 같은 개념으로, 커널은 하나의 매우 큰 모듈들의 집합체이고, 모듈은 커널에 탑재 혹은 제거가 가능한 하나의 기능을 수행하는 코드의 모음이고 flexible 하다.) 커널은 모듈은 시스템의 reboot 없이 커널의 기능을 확장할 수 있다. 예를 들어, 하나의 모듈이 장치 드리이버 이고, 이 모듈은 시스템에 연결된 하드웨어를 커널과 연결시켜주는 것을 허용해 준다고 하자. 이 모듈이 없다면, 우리는 이 역할을 위한 하나의 단일구조로 된 다른 커널을 작성해야 하고 현 커널이미지에 이러한 새로운 기능을 직접적으로 추가해야 한다. 게다가 커널이 커진다는 것은, 즉 이러한 모듈의 개념이 없다고 생각해 본다면(모듈의 역할을 하는 것이 없으니 계속적으로 단일구조로 이루어진 작은 커널을 만들어 현 커널에 추가시켜야 하는 작업을 지속적으로 추가해야 한다면), 우리가 어떤 추가적인 기능을 커널에 사용하고 싶다면 계속 커널을 새로 만들고, 전체 시스템을 reboot 해야 하는 번거롭고 귀찮은 일들이 생기게 된다.

1.2 How Do Modules Get Into The Kernel?

여러분은 이미 커널에서 실행되고 있는 Ismod를 통해 어떤 모듈들이 이미 커널속으로 로딩되어 있나 확인해볼 수 있다. ismod는 /proc/modules 파일을 읽어 현 커널의 모듈들에 대해 알려준다.

어떻게 모듈은 커널에서의 자신이 있어야 할 곳을 찾는걸까?
커널은 자신에게 없는 새로운 기능을 필요로 할 때, 커널모듈데몬인 kmod가 mobprobe 명령어를 실행하여 필요한 모듈을 커널로 불러들인다.

Instruction - mobprobe?
커널 모듈추가, 제거하는 명령어
일반적으로 시작 스크립트에서 자동으로 호출되는 모듈 로더

※insmod 와 modprobe 의 차이점
insmod는 단일모듈 지정된 모듈만 추가하지만 modprobe를 사용하면 단일모듈, 의존성이 있는 모듈까지 모두 자동으로 추가한다.

사용 :: modprobe [option] module

옵션
-l : 사용가능한 모든 모듈을 보여줌
-r : rmmod 와 같은 기능으로 지정모듈을 제거.
    한꺼번에 여러 모듈 지정가능하고 의존성이 있는 모듈또한 자동으로 제거
-c : 기본값과 /etc/modules.conf 에 정의된 지시자를 포함한 완전한 모듈 설정을 보여줌

옵션없이 사용시에 모듈추가
예문 :: e1000 이란 모듈을 추가? -> modprobe e1000

mobprobe는 다음 두가지 형태로 된 모듈명들중 하나를 커널에 전달할수 있다.

· softdog / ppp와 같은 형식의 모듈
· char-major-10-30과 같은 좀더 포괄적인 구분자로 된 모듈

만약 mobprobe가 후자와 같은 경우를 처리한다면, mobprobe는 제일 먼저 /etc/mobprobe.conf[2]를 열어서 기록들을 찾아본다. 만일 mobprobe가 mobprobe.conf에서 다음과 같은 라인을 찾았다고 한다면? :

```
alias char-major-10-30 softdog
```

mobprobe는 이미 char-major-10-30이 softdog 모듈을 말한다는 것을 알고 있다는 뜻이다.

다음으로, mobprobe는 /lib/modules/version/modules.dep 파일을 확인해 다른 모듈이 현제 요청된 모듈을 커널로 로딩하기 전에 실행되어야 하는 다른 모듈이 있는가를 확인한다. (마치 firefox가 실행되기 위해서 XWindows가 실행되어야 하는것처럼..) 이러한 파일들은 depmod -a 명령어로부터 만들어졌고, 또한 모듈 의존성을 가지고 있다. 예를들어, msdos.ko 모듈이 실행되기 위해서 fat.ko 모듈이 먼저 커널로 로딩되어 있어야 한다. 요청된 모듈이 만일 다른 모듈이 정의한 심볼들(변수 혹은 함수)을 사용해야 한다면, 요청된 모듈은 모듈 의존성을 지니게 된다.

마지막으로, mobprobe는 우선적으로 로딩되어야 하는 모듈들을 처리하기 위해 insmod를 사용한다. 그리고나서 요청된 모듈을 처리한다. mobprobe는 insmod를 모듈에 대한 기본적인 사전적 정보가 있는 /lib/modules/version/ 폴더로 이끈다. insmod는 모듈의 위치에 대해 잘 알지못하지만, 반면에 mobprobe는 기본적인 모듈의 위치를 알고 있다. 그렇기 때문에 mobprobe는 모듈을 올바른 순서대로 로딩하기 위한 의존성과 같은 정보에 대해 추정해 낼 수 있다. 예를 들어, 여러분이 msmod 모듈을 로딩하기 위해선 다음 두가지 명령어중 하나를 입력해야 한다.

· insmod /lib/modules/2.6.11/kernel/fs/fat/fat.ko & insmod /lib/modules/2.6.11/kernel/fs/msdos/msdos.ko

or

· mobprobe msdos

우리가 여기서 확인할수 있는점은 insmod는 모듈의 정확한 위치와 입력순서를 요구한다는 점과, 이와는 다르게 mobprobe는 우리가 불러들이고자 하는 모듈의 이름만 입력하면 된다는 점이다. 단지 이름만 입력한다면, mobprobe는 msdos 모듈의 로딩을 위해 알아야하는 모든 것들을 /lib/modules/version/modules.dep 파일을 분석하여 실행하여 준다.

Linux distros는 mobprobe와 module-init-tools 패키지를 통해 insmod와 depmod를 제공한다. 이전버전의 module-init-tools는 modutils 라는 패키지명으로 불렸다. 다른 어떤 distros는 2.4와 2.6버전의 커널들을 올바르게 작동하기 위해 어떤 wrapper를 설정하여 두가지의 패키지를 모두 설치하게 한다. 사용자들은 이것들에 대해 걱정할 부분이 없다.

이제 여러분은 어떻게 모듈이 커널에 로딩되는가에 대해 알게 되었다. 여러분이 다른모듈에 의존성을 가지는 새로운 모듈을 작성하고 싶다면, 본 가이드에 이부분에 대한 내용이 더 있다. (우리는 위와같은 모듈을 sticky module이라 부른다.) 그러나, 다른부분을 먼저 살펴본 다음에 이 부분에 대한 조금더 세부적인 내용을 다루도록 하자.

1.2.1 Before We Begin

세부적인 커널코드 분석전에 우리는 여러 가지 부분에 대해 살펴볼 것이다. 모든 사용자의 시스템은 다르고 각자 자신만의 형태를 가지고 있다. 여러분이 "Hello World" 프로그램을 올바르게 컴파일해여 실행하기 위해선 가끔은 속임수를 쓸수 있어야 한다. 여러분이 첫 번째 관문을 통과한다면, 그이후로는 쉽게 그러한 속임수들을 사용할수 있을 것이다.

1.2.1.1 Modversioning

어떤 특정한 커널을 위해 컴파일된 모듈은 여러분이 다른 커널로 부팅을 할 경우에 CONFIG_MODVERSIONING을 허용하기 전까지 로딩되지 않는다. 우리는 이 가이드의 후반부 전까진 이러한 module versioning을 다루지 않을 것이다. 우리가 mod versioning을 다루기 전에, 만일 여러분이 mod versioning이 활성화 되어있는 커널을 사용한다면 이 가이드의 어떠한 예들은 정상적으로 작동되지 않을수 있다. 그러나, 대부분의 linux distro 커널은 mod versioning을 활성화 하고 있다. 만일 여러분이 모듈 로딩에서 versioning error 문제가 발생한다면, 커널을 modversioning을 비활성화 한 상태로 커널을 컴파일 해보자.

1.2.1.2 Using X

우리는 여러분이 우리가 다룬 모든 예를 타이핑하고, 컴파일하여 로딩해보길 권장한다. 또한 이러한 모든 작업을 콘솔모드를 통해 수행하길 추천한다. 이말 은, 여러분이 XWindow를 통한 작업을 하지 않았으면 한다는 거다.

모듈은 printf( )와 같은 출력을 하지 못한다. 그러나 모듈은 오직 콘솔에서만 여러분들의 스크린에 출력이 끝나기 전에 그에 관련된 정보와 경고들에 대해 기록을 남길수 있다. 만일 여러분이 xterm을 통해 어떤 모듈을 insmod 한다면(insmod를 통해 로딩한다면) 그에 관련된 정보와 경고들은 오직 로그파일을 통해 기록될 것인데, 여러분은 그 로그파일을 열지 않는이상 그러한 정보와 경고를 확인하지 못한다. 이러한 정보에 빠르게 접근하기 위해, 모든 작업은 콘솔을 통해서 하자.

1.2.1.3 Compiling Issues and Kernel Version

리눅스 배포판(역주: Radhat Linux, Fedora, Ubuntu와 같은..) 은 문제의 여지가 있는 것들에 대해 패치한 새로운 커널버전을 자주 분배한다.

종종 리눅스 배포판은 불완전한 커널 헤더를 분배하는 경우가 있는데, 여러분은 리눅스 커널의 다양한 헤더파일을 통해 여러분의 코드를 컴파일 해야 할 필요가 있다. Murphy's Law는 "문제가 발생한 헤더는 커널헤더에서 정확하게 당신의 모듈을 동작시키기 위한 부분이다." 라고 말한다.

이러한 두가지 문제를 피하기 위해, 여러분들만의 순정의 리눅스 커널을 다운로드하여 설치하고 그것을 통해 작업하길 권장한다.

역설적이게도, 이것 역시 문제를 발생 시킬수 있다. 기본적으로, 여러분의 시스템에 있는 gcc는 여러분이 새롭게 설치한 커널보다는(/usr/src/) 기본적인 위치의 커널 헤더를 우선적으로 찾기 때문인데, 이 문제는 gcc's -i switch를 통해 해결할수 있다.

Chapter2. Hello World

2.1 Hello, World (part 1) : The Simplest Module

아직 숙달되지 못한 초보 프로그래머들이 프로그래밍을 할 때 제일 처음으로 제작하는 프로그램은 다들 알고 있는거와 같이 "Hello World" 프로그램이다. 로마의 프로그래밍 교재는 "Salut, Mimdi" 프로그램으로 시작하는데, 이 전통을 누가 깨뜨렸는지 저자도 알지 못한다.. 다만 나의 생각엔 누가 그 전통을 깻는지 찾지 않는 것이 좋을듯하다.. :)

우리는 몇가지 "Hello World" 프로그램들로 커널모듈 작성의 여러 가지 다른 측면들을 살펴볼 것이다.

아래에 가장 간단한 모듈의 예제가 있다. 다음장 에서 커널 컴파일에 대하여 살펴볼 것이기 때문에 아직 컴파일은 안해도 되겠다.

```
/*
* hello-1.c - The simplest kernel module.
*/
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void) {
        printk(KERN_INFO "Hello world 1.₩n");
        /*
        * A non 0 return means init_module failed; module can't be loaded.
        */
        return 0;
}

void cleanup_module(void) {
        printk(KERN_INFO "Goodbye world 1.₩n");
}
```

커널모듈은 최소한 두가지 함수를 갖추어야 한다. :: 하나는 "start"함수 (초기화) 로 이것을 init_module( ) 이라 부르는데, 이 함수는 작성된 모듈이 insmod 명령어를 통해 커널에 로딩될 때 호출된다. 그리고 나머지 하나는 "end" 함수로 cleanup_module( )이라 불리고 이것은 rmmod 명령어를 통해 모듈이 종료되기 전에(커널에서 제거될 때) 호출된다. 사실 커널 2.3.13 버전이후로 이러한 함수들은 변하였다. 여러분은 모듈이 시작되고 종료될 때, 함수 이름에 제한없이 이러한 시작과 종료함수를 작성할수 있는데, 우리는 이것을 2.3에서 살펴볼 것이다. 사실, 이러한 새로운 방식은 최근에 선호되는 방식이다. 그러나 많은 사람들이 아직 init_module( )과 cleanup_module( )을 사용한다.

일반적으로 init_module( )은 커널에 handler를 등록하거나, 혹은 커널함수의 한 부분을 자신의 코드로 대체한다.(보통 자신의 코드를 실행한 후 원본함수를 다시 불러온다) cleanup_module( )은 init_module( )이 수행한 기능을 되돌리는 역할을 하는데 이를 통해 모듈이 안전하게 커널의 한부분으로부터 원본으로 대체된다.

마지막으로, 모든 커널모듈은 linux/module.h 헤더파일을 필요로 한다. 우리는 printk( ) log level marco expansion과 KERN_ALERT을 위해서만 linux/kernel.h 헤더파일을 로드해야 하는데, 우리는 2.1.1에서 이것에 대해 살펴볼 것이다.

2.1.1 Introducing printk( )

여러분이 어떤 생각을 하던지간에, printk( )함수는 유저를 위한 통신수단이 아니다. 심지어 우리가 hello-1 에서와

같이 정확히 출력의 목적으로 사용을 했지만 말이다. 이 함수를 실행했을 때, 커널의 logging mechanism 이 수행되고, 이 logging mechanism은 수행한 함수의 정보를 기록하거나 혹은 경고를 알린다. 그리므로, 각각의 printk( ) 선언은 우선순위를 통해 제공되는데, 아까 살펴봤던 <1>과 KERN_ALERT이다. printk( ) 함수에는 8가지의 속성과 커널이 매크로를 가지고 있어, 여러분은 이상한 숫자를 사용하지 않아도 되고, 또한 매크로에 대한 정보를 linux/kernel.h에서 확인할수 있다. 만일 여러분이 우선순위 수준을 명시하지 않았을 경우, 기본 우선순위인 DEFAULT_MESSAGE_LOGEVEL이 사용될 것이다.

우선순위 매크로를 천천히 읽어보자. 위에서 언급한 헤더파일은 역시 각각의 우선순위가 의미하는 바를 기술해 놓았다. 실제 모듈작성시에, <4>와 같은 숫자를 사용하지 말고 항상 KERN_WARNING 과 같은 매크로를 사용하도록 하자.

만일 우선순위가 int_console_loglevel보다 낮은 경우, 메시지가 여러분의 터미널에 출력될 것이다. 만일 syslogd와 klogd이 동시에 실행중이라면, 출력된 메시지는 콜솔 출력여부와 관계없이 /var/log/messages에 기록될 것이다. 우리는 단순히 메시지들이 파일로 기록되는 것을 원치 않기 때문에, KERN_ALERT와 같이 높은 우선순위를 가진 매크로를 사용하여 printk( ) 함수가 여러분의 콘솔에 확실히 출력되게 할 것이다. 여러분이 실제 모듈을 작성할 때, 여러분은 현 상황에 의미가 있는 우선순위를 사용하길 원할 것이다.

2.2 Compiling Kernel Modules

커널 모듈은 일반적인 사용자 프로그램과는 약간 다르게 컴파일 되어야 한다. 이전의 커널버전은 이러한 차이점에 대해 좀더 상세하게 다루도록 요구하였다. (이러한 상세한 옵션들은 보통 MakeFile에 저장되어 있다.) 비록 계층적으로 조직되었다 하더라도, 많은 불필요한 설정들은 MakeFiles의 sublevel에 집중되어 있고 이 때문에 커널 모듈을 커지게 하고 유지하기 어렵게 한다. 운좋게도, 이러한 불행한 경우를 없애줄 새로운 방식이 등장하였는데, 우리는 이것을 kbuild라고 부르고, 외부에서 읽어올수 있는 모듈을 위한 build process는 이제 일반적인 kernel build mechanism에 포함되어 있다. 공식적인 커널의 일부분이 아닌 모듈을 컴파일 하는 방법에 대해선 (우리가 작성하고 연습하는 모든 이 가이드의 예제들을 말함) linux/documentation/kbuild/modules.txt를 참조하자.

이제 MakeFile을 이용해 간단한 hello-1.c의 모듈을 컴파일 해보자.

Example 2-2. Makefile for a basic kernel module

```
obj-m += hello-1.o

all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

첫 번째 라인이 정말 필요하다는 기술적인 관점에서 볼 때, all과 clean 명령어는 단지 순수한 편의를 위해서 추가되었다고 볼 수 있다. 이제 여러분은 make 명령어를 통해 모듈을 컴파일 할수 있게 되었다. make 명령어로 컴파일을 했기 때문에 여러분은 다음과 같은 출력결과를 얻어야 한다.

```
hostname:~/lkmpg-examples/02-HelloWorld# make
make -C /lib/modules/2.6.11/build M=/root/lkmpg-examples/02-HelloWorld modules
make[1]: Entering directory `/usr/src/linux-2.6.11'
CC [M] /root/lkmpg-examples/02-HelloWorld/hello-1.o
Building modules, stage 2.
MODPOST
CC /root/lkmpg-examples/02-HelloWorld/hello-1.mod.o
LD [M] /root/lkmpg-examples/02-HelloWorld/hello-1.ko
make[1]: Leaving directory `/usr/src/linux-2.6.11'
hostname:~/lkmpg-examples/02-HelloWorld#
```

커널 2.6버전은 새로운 파일 이름에 관한 내용을 담고 있다는 것을 기억하자 : 커널모듈은 이제 .ko 확장자를

가지는데 (이전의 .o 확장자를 대채함) 이를 통해서 이전 object file들로부터 쉽게 구분할수 있다. 이러한 확장자 변경의 이유는 .modinfo에 관한 추가적인 부분을 담고 있기 때문인데, 모듈이 어디에 보관되에 있는지에 대한 정보이다. 우리는 이 정보가 왜 좋은지에 대해 살펴볼 것이다.

modinfo hello.*ko를 사용하여 보자.

```
hostname:~/lkmpg-examples/02-HelloWorld# modinfo hello-1.ko
filename: hello-1.ko
vermagic: 2.6.11 preempt PENTIUMII 4KSTACKS gcc-3.3
depends:
```

딱히 특별한 것이 없는데, 우리의 마지막 예제인 hello-5.ko를 살펴볼 때 달라진다.

```
hostname:~/lkmpg-examples/02-HelloWorld# modinfo hello-5.ko
filename: hello-5.ko
license: GPL
author: Peter Jay Salzman
vermagic: 2.6.11 preempt PENTIUMII 4KSTACKS gcc-3.3
depends:
parm: myintArray:An array of integers (array of int)
parm: mystring:A character string (charp)
parm: mylong:A long integer (long)
parm: myint:An integer (int)
parm: myshort:A short integer (short)
hostname:~/lkmpg-examples/02-HelloWorld#
```

이 출력물엔 많은 유용한 정보들이 제공되어 있다.

커널모듈을 위한 MakeFiles에 관한 추가적인 정보는 linux/Documentation/kbuild/makefiles.txt에 기술되어 있다. MakeFiles를 파보기 전에 이 문서를 확실히 읽어두면 많은 도움이 될 것이다.

이제 insmod ./hello.1-lo 명령어를 통해 컴파일한 여러분의 커널모듈을 커널에 심어볼 시간이다.

커널로 불러들여진 모든 모듈들은 /proc/modules 에 기술되어 있다. 이 파일을 살펴본다면 실제로 커널을 구성하고 있는 모듈들이 어떤것인가에 대한 정보를 얻을수 있다. 축하한다! 여러분은 이제 리눅스 커널코드의 저자가 되었다. 또한 rmmod hello-1 명령어를 통해 여러분이 작성한 커널모듈을 커널로부터 제거할수 있다. /var/log/messages를 통해 여러분의 시스템 로그파일을 살펴보아라.

여기 다른 예제가 제공되어 있다. init_module( )의 리턴문 위의 내용이 보이는가? 리턴값을 바꾸어 새로 컴파일을 하고 커널로 넣어보자. 어떠한 일이 일어날까?

2.3 Hello, World (part 2)

2.4 리눅스에서는 init과 cleanup 함수를 재명명 할수 있다. 즉, init_module( ) 혹은 cleanup_module( ) 같은 이름으로 불려지지 않아도 된다는 것이다. 위의 함수들은 module_init( )과 module_exit( ) 매크로를 통해 수행된다. 이러한 매크로들은 linux/init.h 헤더파일에 정의되어 있다. 이 매크로들을 사용할 때 여러분이 주의해야 할 유일한 부분은 init과 cleanup이 매크로 호출 전에 정의가 되어 있어야 한다는 점이다. 그렇지 않다면 컴파일 에러가 발생할 것이다. 여기에 이러한 방법에 대한 예제가 제공되어 있다.

Example 2-3. hello-2.c

```
/*
* hello-2.c - Demonstrating the module_init() and module_exit() macros.
* This is preferred over using init_module() and cleanup_module().
*/
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h> /* Needed for the macros */
static int __init hello_2_init(void)
{
        printk(KERN_INFO "Hello, world 2₩n");
        return 0;
}
static void __exit hello_2_exit(void)
{
        printk(KERN_INFO "Goodbye, world 2₩n");
}
module_init(hello_2_init);
module_exit(hello_2_exit);
```

이제 여러분은 두 개의 실제 커널모듈을 다룰수 있게 되었다. 다른 모듈을 커널로 불러들이는것도 아래와 같이 쉽다 :

Example 2-4. Makefile for both our modules

```
obj-m += hello-1.o
obj-m += hello-2.o
all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

linux/drivers/char/Makefile를 살펴보자. 여러분이 본것과 같이 뭔가가 커널에 내장되어 있지만 이것들의 obj-m 들은 어디로 사라진걸까? 그러한 쉘스크립트와 유사한 파일은 찾아내기가 쉽다. 그렇지 않다면, obj-$(CONFIG-FOO)파일들은 obj-y 혹은 obj-m 으로 확장되어 있을것인데, 이것은 CONFIG-FOO 변수가 어떻게 설정되어 있느냐에 달려있다.(y 혹은 m으로) 우리가 언급하고 있지만, 이러한 것들은 여러분들이 make menuconfig 혹은 그러한 명령어를 통해 설정하는 linux/config 파일의 변수들의 한 종류이다.

2.4 Hello World (part 3) : The _init and _exit Macros

이 부분은 커널 2.2버전 이후에 관해 실제 예를 들어 설명한다. init 과 cleanup 함수 정의의 변화를 알아보자. _init 매크로는 사용가능한 모듈이 아닌 내장된 드라이버에 대해 init 함수를 불러온후 후 메모리로부터 제거된다. init 함수가 불려졌다고 생각한다면 이해가 될 것이다.

또한 함수보단 init 변수를 위한 _initdata가 존재한다.

_exit 매크로는 커널에 모듈이 로딩되었을 때 함수의 누락을 야기하는데, _init과 마찬가지로 사용가능한 모듈에는 효과가 없다. 다시말해 여러분이 cleanup 함수가 동작할때를 생각해 본다면, 실행가능한 모듈과는 반대로 내장된 드라이버(built-in driver) 는 cleanup 함수가 필요하지 않다는 점이 완벽하게 이해가 될 것이다.

이러한 매크로는 linux/init.h 에 정의되어 있고 커널메모리의 공간을 확보하여 준다. 여러분이 커널을 부팅하고 "Freeing unused kernel memory : 236k frred" 와 같은 메시지를 본다면, 커널메모리가 최적화됬음을 명확하게 알수 있을 것이다.

Example 2-5. hello-3.c

```
/*
* hello-3.c - Illustrating the __init, __initdata and __exit macros.
*/
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h> /* Needed for the macros */
static int hello3_data __initdata = 3;
static int __init hello_3_init(void)
{
        printk(KERN_INFO "Hello, world %d₩n", hello3_data);
        return 0;
}
static void __exit hello_3_exit(void)
{
        printk(KERN_INFO "Goodbye, world 3₩n");
}
module_init(hello_3_init);
module_exit(hello_3_exit);
```

2.5 Hello World (part 4) : Licensing and Module

Documentation

만약 여러분이 커널 2.4버전 이후의 커널을 사용하고 있는 경우에, 여러분이 제작자가 명시된 모듈을 로딩했을 경우에 다음과 같은 메시지를 봤을 것이다.

```
# insmod xxxxxx.o
Warning: loading xxxxxx.ko will taint the kernel: no license
See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Module xxxxxx loaded, with warnings
```

커널 2.4 버전 이후에, GPL(General Public License) 에 영향을 받아 코드의 라이센스를 명시하는 메커니즘이 고안되었다. 그래서 사용자들은 위와 같은 모듈을 사용할 때 그 모듈이 오픈소스가 아니라는 것에 대한 경고메시지를 보게 된다. 라이센스에 대해 GPL을 적용하게 되면 위와같은 경고메시지는 출력되지 않는다. 위와 같은 라이센스 메커니즘은 linux/module.h 헤더파일에 상세히 정의되어 있으니 참고하길 바란다.

```
/*
* The following license idents are currently accepted as indicating free
* software modules
*
* "GPL" [GNU Public License v2 or later]
* "GPL v2" [GNU Public License v2]
* "GPL and additional rights" [GNU Public License v2 rights and more]
* "Dual BSD/GPL" [GNU Public License v2
* or BSD license choice]
* "Dual MIT/GPL" [GNU Public License v2
* or MIT license choice]
* "Dual MPL/GPL" [GNU Public License v2
* or Mozilla license choice]
*
* The following other idents are available
*
* "Proprietary" [Non free products]
*
* There are dual licensed components, but when running with Linux it is the
* GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL
* is a GPL combined work.
*
* This exists for several reasons
* 1. So modinfo can show license info for users wanting to vet their setup
* is free
* 2. So the community can ignore bug reports including proprietary modules
* 3. So vendors can do likewise based on their own policies
*/
```

비슷하게, MODULE_DESCRIPTION( )은 모듈이 어떤 역할을 수행하는지에 대해 기술할수 있는 명령어 이고, MODULE_AUTHOR( ) 명령어는 모듈의 저자를, MODULE_SUPPORTED_DEVICE( )는 모듈이 지원하는 장치에 대해 기술하는 명령어이다.

이러한 매크로들은 linux/module.h 헤더파일에 정의되어 있으며 커널 스스로 이러한 매크로들을 사용하지 않는다. (즉, 사용자가 코드상에 적용해야 하는 부분이다.) objdump 와 같은 툴을 사용하면 위와 같은 매크로들을 손쉽게 확인할수 있다. 여러분 스스로 linux/driver에서 위와같은 툴을 사용하여 모듈의 저자가 어떻게 자신들의 모듈에 위와 같은 정보들을 기술 하였는지 쉽게 확인해 볼수 있을 것이다.

grep -inr MODULE_AUTHOR 과 같은 명령어도 추천할만한 구문이다. 이런 구문에 익숙하지 않은 사용자라면 아마 LXR과 같은 웹기반의 솔루션을 좋아할 것이다.

vi나 emacs와 같은 유닉스의 전통적인 문서편집기를 주로 사용하는 유저라면, 그러한 편집기들이 tag files를 찾는데 유용하게 사용될 것이다. 그러한 파일들은 make tags 나 make TAGS 와 같은 명령어를 통해 usr/src/linux-2.6.x/ 에 생성되는데, 일단 여러분들의 커널 트리에서 그러한 태그파일들을 찾았다면, 커서를 이용하여 어떤 함수를 호출하거나 특정 키 조합을 통해 함수의 정의부로 바로 이동할수 있을 것이다.

Example 2-6. hello-4.c

```
/*
* hello-4.c - Demonstrates module documentation.
*/

#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h> /* Needed for the macros */

#define DRIVER_AUTHOR "Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC "A sample driver"

static int __init init_hello_4(void)
{
        printk(KERN_INFO "Hello, world 4₩n");
        return 0;
}

static void __exit cleanup_hello_4(void)
{
        printk(KERN_INFO "Goodbye, world 4₩n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);

/*
* You can use strings, like this:
*/

/*
* Get rid of taint message by declaring code as GPL.
*/

MODULE_LICENSE("GPL");

/*
* Or with defines, like this:
*/
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */

/*
* This module uses /dev/testdevice. The MODULE_SUPPORTED_DEVICE macro might
* be used in the future to help automatic configuration of modules, but is
* currently unused other than for documentation purposes.
*/
MODULE_SUPPORTED_DEVICE("testdevice");
```

2.6 Passing Command Line Arguments to a Module

모듈은 커맨드라인의 인자들을 취할수 있는데, 여러분들이 이미 사용할지 모르는 argc/argv 는 사용할수 없다. 여러분들의 모듈에 인자들이 사용될수 있게 하기위해, 커맨드라인 인자들을 전역변수로 선언하고, module_parm( ) 매크로를 사용하여 (linux/moduleparm.h 헤더파일에 정의되어 있다.) 위의 인자들의 메커니즘을 설정할수 있다. 실행시에 insmod 는 ./insmod mymodule.ko myvariable=5 와 같은 명령어를 통해 주어진 커맨드라인의 인자들을 이용해 변수들을 채울 것이다. 변수정의와 매크로는 가독성을 위해 모듈의 앞부분에 위치하는 것이 좋다. 아래에 있는 예제가 위에서 설명한 조건들을 잘 반영하고 있다.

module_parm( ) 매크로는 3개의 인자를 취한다 : 변수의 이름과 타입, 권한. (sysfs 에 대응하는 파일을 위해) Integer 타입의 변수는 unsigned나 일반적인 보통의 signed 로 정의될수 있다. 여러분들이 배열형 Integer 변수를 사용하고 싶다면, module_param_array( )과 module_param_string( ) 매크로를 사용하면 된다.

```
int myint = 3;
module_param(myint, int, 0);
```

위에서 언급했지만, 배열은 지원가능하나 2.4버전과는 약간 차이점이 있다. 인자들의 개수를 파악하기 위해, 여러분들은 세 번째 인자로 count variable에 대한 포인터를 넘겨주어야 한다. 여러분들은 count를 무시하고 NULL을 대신 넘겨줄수도 있다. 아래에 두가지 모두에 대해 예시를 제공하였다.

```
int myintarray[2];
module_param_array(myintarray, int, NULL, 0); /* not interested in count */

int myshortarray[4];
int count;
module_parm_array(myshortarray, short, , 0); /* put count into "count" variable */
```

좋은 사용예로는, 모듈 변수를 포트번호 혹은 IO 주소와 같은 기본값으로 설정하는 것이다. 만약 변수들이 기본값을 가지고 있다면 autodetection을 수행하라. 아니면, 현 값을 유지해라. 이부분에 대해 추후에 명확하게 설명할 것이다.

마지막으로 MODULE_PARM_DESC( ) 매크로 함수가 있는데, 모듈이 취하는 변수의 기술에 사용된다. 위 매크로 함수는 2가지의 인자를 취하는데, 변수 이름과 string으로 이루어진 변수에 대한 설명문이다.

Example 2-7. hello-5.c

```
int myint = 3;
module_param(myint, int, 0);
int myintarray[2];
module_param_array(myintarray, int, NULL, 0); /* not interested in count */
int myshortarray[4];
int count;
module_parm_array(myshortarray, short, , 0); /* put count into "count" variable */

/*
* hello-5.c - Demonstrates command line argument passing to a module.
*/

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";
```

```
static int myintArray[2] = { -1, -1 };
static int arr_argc = 0;

/*
* module_param(foo, int, 0000)
* The first param is the parameters name
* The second param is it's data type
* The final argument is the permissions bits,
* for exposing parameters in sysfs (if non-zero) at a later stage.
*/
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");

module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");

module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");

module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

/*
* module_param_array(name, type, num, perm);
* The first param is the parameter's (in this case the array's) name
* The second param is the data type of the elements of the array
* The third argument is a pointer to the variable that will store the number
* of elements of the array initialized by the user at module loading time
* The fourth argument is the permission bits
*/
module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
static int __init hello_5_init(void)
{
        int i;
        printk(KERN_INFO "Hello, world 5₩n=============₩n");
        printk(KERN_INFO "myshort is a short integer: %hd₩n", myshort);
        printk(KERN_INFO "myint is an integer: %d₩n", myint);
        printk(KERN_INFO "mylong is a long integer: %ld₩n", mylong);
        printk(KERN_INFO "mystring is a string: %s₩n", mystring);
        for (i = 0; i < (sizeof myintArray / sizeof (int)); i++)
        {
                printk(KERN_INFO "myintArray[%d] = %d₩n", i, myintArray[i]);
        }
        printk(KERN_INFO "got %d arguments for myintArray.₩n", arr_argc);
        return 0;
}
static void __exit hello_5_exit(void)
{
        printk(KERN_INFO "Goodbye, world 5₩n");
}
module_init(hello_5_init);
module_exit(hello_5_exit);
```

저자는 여러분이 아래와 같은 코드와 유사하게 사용하길 권장한다.

```
satan# insmod hello-5.ko mystring="bebop" mybyte=255 myintArray=-1
mybyte is an 8 bit integer: 255
myshort is a short integer: 1
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: bebop
myintArray is -1 and 420
satan# rmmod hello-5
Goodbye, world 5
satan# insmod hello-5.ko mystring="supercalifragilisticexpialidocious" ₩
> mybyte=256 myintArray=-1,-1
mybyte is an 8 bit integer: 0
myshort is a short integer: 1
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
myintArray is -1 and -1
satan# rmmod hello-5
Goodbye, world 5
satan# insmod hello-5.ko mylong=hello
hello-5.o: invalid argument syntax for mylong: 'h'
```

2.7 Module Spanning Multiple Files

때로는 커널모듈을 여러개의 파일로 분리해야 할 경우가 생긴다.
아래에 위에 대한 예를 실었다.

Example 2-8. start.c
The next file:
Example 2-9. stop.c
And finally, the makefile:
Example 2-10. Makefile

```
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
myintArray is -1 and -1
satan# rmmod hello-5
Goodbye, world 5
satan# insmod hello-5.ko mylong=hello
hello-5.o: invalid argument syntax for mylong: 'h'

/*
* start.c - Illustration of multi filed modules
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
int init_module(void)
{
        printk(KERN_INFO "Hello, world - this is the kernel speaking₩n");
        return 0;
}

/*
* stop.c - Illustration of multi filed modules
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
void cleanup_module()
{
        printk(KERN_INFO "Short is the life of a kernel module₩n");
}

obj-m += hello-1.o
obj-m += hello-2.o
obj-m += hello-3.o
obj-m += hello-4.o
obj-m += hello-5.o
obj-m += startstop.o
startstop-objs := start.o stop.o

        all:
                make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
        clean:
                make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

첫 번째 다섯 개의 라인까진 별다른 특별한 점이 없지만, 마지막 예제에서 우리는 두 개의 명령문을 필요로 한다.
첫 번째는 여러개의 파일들로 결합된 모듈을 위해 object file을 만드는 것이고, 두 번째는 make 명령어를 통해 각각의 object 파일들이 모듈에서 어떤 부분인가를 명시해 주는 것이다.

2.8 Building modules for a precompiled kernel

명백하게 저자는 forced module unloading (MODULE_FORCCE_UNLOAD) : "위의 옵션이 활성화 되었다면 rmmod -f module의 명령어를 통해 로딩한 모듈이 안전하지 않다고 생각했을 때 강제로 종료할수 있다." 와 같은 유용한 것들을 여러분들의 커널에서 사용할수 있게끔 하기위해 여러분의 커널을 재컴파일 할 것을 추천했다. 위와 같은 옵션의 설정은 여러분들의 수고를 덜어주고, 모듈의 개발과정에서의 커널의 재부팅횟수를 획기적으로 줄여줄 것이다.

반면에 이전에 컴파일한 커널이나, 공통적인 Linux 배포판들을 통해 딸려온 것들과 같은 이미 컴파일된 커널들에 여러분들의 모듈을 집어넣고 싶을 경우가 생길 것이다. 위와 같은 상황에서 여러분들은 재컴파일이 허용되지 않은 커널 혹은 재부팅하고 싶지 않은 머신에서 실행중인 커널에 여러분들의 모듈을 컴파일하고 탑재할수 있다. 만약 여러분이 위와 같은 경우의 필요성을 느끼지 못한다면, 이 부분은 그냥 넘기고 필요할 때 다시 공부해도 좋다.

이제, 만약 여러분이 커널 source tree를 설치해서 여러분들의 커널모듈 컴파일에 사용하고 커널에 탑재하길 원한다면 대부분의 경우에 아래와 같은 에러를 만나게 될 것이다.

```
insmod: error inserting 'poet_atkm.ko': -1 Invalid module format
```

좀더 명확한 정보는 /var/log/messages 에 기록된다.

```
Jun 4 22:07:54 localhost kernel: poet_atkm: version magic '2.6.5-1.358custom 686
REGPARM 4KSTACKS gcc-3.3' should be '2.6.5-1.358 686 REGPARM 4KSTACKS
gcc-3.3'
```

위를 해석해보면, "여러분들이 탑재하고싶은 모듈의 version string (정확히 말해 version magic) 이 여러분의 커널과 일치하지 않아 탑재가 허용되지 않았다" 라는 에러메세지이다. 명백히 version magic은 vermagic으로 시작하는 static string의 형태로 module obeject에 저장되어 있다. 주어진 모듈의 version magic과 다른 문자들을 조사하기 위해 modinfo module.ko 명령어를 사용하자.

```
[root@pcsenonsrv 02-HelloWorld]# modinfo hello-4.ko
license: GPL
author: Peter Jay Salzman <p@dirac.org>
description: A sample driver
vermagic: 2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3
depends:
```

위와 같은 경우를 극복하기 위해 우리는 --force-vermagic과 같은 명령어를 사용해야 하지만 이와 같은 해결책은 잠재적으로 안전하지 못하고 일반적으로 모듈 제작에서 허용되지 않는다. 결과적으로볼 때, 컴파일된 커널과 동일한 환경에서 우리의 모듈을 컴파일 하는 것이 해결책이 된다는 말이다. 이 부분에 대해선 이 챕터의 나머지부분의 주제로 두겠다.

무엇보다도, 현제 커널과 동일한 버전의 커널 source tree가 사용가능하도록 하자. 그리고나서 우리들의 커널의 설정에 사용되는 파일을 찾자. (/boot 폴더에 config-2.6.x 와 같은 형태로 있을 것이다.) 그리고 위의 설정파일을 여러분의 커널 source tree에 복사하자. (cp /boot/config-'uname-r' /usr/src/linux-'uname-r'/config.)

다시 이전의 에러메세지에 초점을 맞추어보자. : 주어진 설정파일의 version magic 부분을 자세히 살펴보자. 심지어 두가지의 설정파일이 정확하게 일치할 경우도 있을 것이다. 그러나 아주 조그마한 version magic의 차이도 있을 것인데, 이러한 작은 차이는 모듈이 커널로 탑재되는 것을 막기에 충분한 차이이다. 이러한 자그마한 차이를 custom 이라 하는데 모듈의 version magic 엔 존재하지만 커널엔 존재하지 않는 부분이다. (custom은 makefile이 만들어내는 경우인데, 원본 커널을 바탕으로 수정을 가할 경우에 발생한다.) 그리고 우리의 /usr/src/linux/makefile을

살펴서 명시된 버전정보가 우리가 사용하는 커널의 버전정보와 일치하는지를 확인하자. 예를 들어 여러분의 makefile은 다음과 같이 시작될 것이다.

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 5
EXTRAVERSION = -1.358custom
...
```

이러한 경우에, 여러분은 EXTRAVERSION의 값을 -1.358로 변경해야 한다. 우리는 여러분들의 커널 컴파일에 사용된 makefile을 백업할 것을 권장한다. (/lib/modules/2.6.5-1.358에 위치해 있고,
간단히 cp /lib/modules/'uname-r'/build/Makefile /usr/src/linux-'uname-r' 명령어를 통해 백업을 수행할수 있다.)
추가로, 만약 여러분이 이전의 makefile 로 커널을 시작하였다면, make 명령어를 다시 실행하거나 혹은 직접적으로 /usr/src/linux-2.6.x/include/linux/version.h 헤더파일의 UTS_RELEASE를 수정하거나 정상적으로 설정된 파일을 덮어 씌우는 방법도 있다.

이제 make 명령어를 통해 object, version header와 설정을 업데이트 하자 :

```
[root@pcsenonsrv linux-2.6.x]# make
CHK include/linux/version.h
UPD include/linux/version.h
SYMLINK include/asm -> include/asm-i386
SPLIT include/linux/autoconf.h -> include/config/*
HOSTCC scripts/basic/fixdep
HOSTCC scripts/basic/split-include
HOSTCC scripts/basic/docproc
HOSTCC scripts/conmakehash
HOSTCC scripts/kallsyms
CC scripts/empty.o
...
```

만약 여러분이 커널을 컴파일 해야하는 부분에 대해서 망설여 진다면, CRTL+C를 통해 build process를 중단할수 있다. (SPLIT 라인이 나타나기 전에 -> 이미 여러분이 필요한 파일이 준비되어 있다. (version.h. 헤더파일))
이제 여러분들의 모듈을 다시 컴파일해보자. 다른 에러없이 정상적으로 컴파일되어 커널에 탑재될 것이다.

Chapter3. Preliminaries

3.1 Modules vs Programs

3.1.1 How modules begin and end

프로그램은 보통 메인함수로부터 시작되고, 여러 가지의 명령어들을 실행하고 종료한다. 하지만 커널모듈은 이와는 조금 다른방식으로 동작한다. 커널모듈은 항상 _init_module 혹은 module_init 와 같은 함수의 호출로 시작된다. (우리는 위와 같은 함수를 "entry function" 이라 부른다.)위의 함수는 모듈의 시작을 의미하는데, 기능적으로 모듈이 어떤 역할을 하는지, 그리고 우리가 필요로 할 때 모듈이 작동될수 있도록 커널을 설정하는 역할을 한다. 위의것들이 수행된 후, 모듈이 제공하는 코드들이 커널로부터 사용되기 전까지 모듈은 아무것도 하지 않고, 위의 entry function은 종료된다.

모듈은 cleanup_module 혹은 module_exit 의 이름으로 사용자가 구체화한 함수의 호출을 통해 종료된다. 이와 같은 종료함수는 entry function 이 수행한 모든 기능들을 수행하기 전의 상태로 되돌린다. (Undo) 즉, entry function이 register 한 모든 기능들을 unregister 하는 것이다.

모든 모듈들은 위의 두가지, entry 와 exit 기능을 하는 함수를 가지고 있어야 한다. 위의 두가지 기능을 하는 함수를 구체화 하는 다양한 방법이 존재하지만, 우리는 "entry function" 과 "exit function" 라는 용어를 사용하도록 하자. (각각의 용어는 init_module 과 cleanup_module 함수를 의미한다.)

3.1.2 Functions available to modules

프로그래머들은 항상 자신들이 정의하지 않은 함수들을 사용한다. printf( ) 함수가 좋은 간단한 예가 될 수 있겠다. 우리는 이러한 라이브러리 함수들을 사용하여 프로그램을 작성한다. 사실 위와 같은 함수들은 linking 과정을 거치기 전까지 우리들의 프로그램에 실제로 호출되지 않는다. (linking 과정은 printf( )와 같은 라이브러리 함수들을 실제로 우리가 작성한 프로그램에 연결시켜서 호출하는 과정을 말한다.)

커널모듈은 이런 부분에서 또 일반적인 프로그램과 다르다. 우리가 작성한 "hello world" 모듈을 예를 들어보자면, printk( ) 라는 라이브러리 함수를 사용했지만 실제로 I/O Library를 include 하지는 않았다. 그 이유는 모듈은 insmod가 실행되는 중에 printk( )와 같은 함수들의 symbol이 결정되는 object file 이기 때문이다. symbol 들의 함수적 정의는 커널 스스로가 제공한다. 즉, 우리가 사용한 외부함수 (라이브러리 함수들을 의미) 는 커널 자체에서 제공되는 함수인 것이다. 만약 여러분이 커널로부터 추출되는 이러한 symbol 들에 대하여 궁금하다면, /proc/kallsyms 에서 확인 할 수 있다.

일반적인 라이브러리 함수와 system call은 서로 다르다는 것을 기억하자. 라이브러리 함수들은 조금 더 높은 수준의 함수, 즉 system call 보다 조금더 편하게 프로그래머들이 사용할수 있고 완전한 사용자의 환경에서 작동되기 때문에 system call 보다 높은 수준의 함수라 말할수 있다. 반면에 system call은 커널 자체에서 제공되며 커널 과 유저 환경의 중간 단계에서 실행된다. printf( ) 함수는 것보기엔 매우 일반적인 수준의 출력함수로 보이지만, 실제로는 낮은수준의 system call인 write( )를 통해 데이터를 string으로 format화 하고 출력한다.

실제로 printf( ) 함수를 사용한 매우 간단한 소스코드를 아래에 구현해 보았다. 이를 통해 실제로 어떠한 system call 이 사용되는지 확인해 보자.

```
#include <stdio.h>
int main(void)
{ printf("hello"); return 0; }
```

gcc -Wall -o hello hello.c 명령어를 사용해 컴파일 하자. 그 후 strace ./hello 명령어로 실행하여 보자. 놀랍지 않은가? 출력되는 모른 라인은 printf( ) 함수를 사용할 때의 호출되는 system call을 의미한다. strace 명령어는 프로그램이 만들어낸 system call과, 리턴되는 인자들이 무엇인가 에 관해 상세히 출력하여 준다. 그래서 실제로 프로그램이 어떠한 파일에 접근하는지 추측하는 것을 가능케 하는 멋진 툴이다! 끝으로 여러분은 write(1, 'hello', 5hello) 라는 라인을 볼수 있는데, 이것은 printf('hello"); 라인의 감춰진 또다른 라인이다. 여러분이 fopen, fputs, fclose와 같은 라이브러리 함수를 사용하면서부터 위와 같은 write 호출에 대해 익숙치 않을수 있을 것이다.
만일 그렇다면, man 2 write 로 write의 매뉴얼을 살펴보자. write의 2번째 섹션은 system call에 관한 (kill ( ) 과 read( )) 내용을 담고 있다. 3번째 섹션은 library call에 관한 내요을 다루고 있는데 cosh( )와 random( )등의 좀더 여러분이 친숙한 주제들을 다르고 있다.

여러분은 앞으로 짧게나마 커널의 system call을 여러분이 작성한 모듈로 바꾸는 작업을 할 것이다. 크래커들은 종종 트로이얀과 같은 백도어 프로그램을 위해 위와 같은 방법을 사용하기도 한다. 하지만 여러분은 "Tee hee, that tickles!" 와 같이 좀 더 온순한 것들을 위해 여러분의 모듈을 작성할 것이다.

3.1.3 User Space vs Kernel Space

커널은 자원에 관련되어 있다. 그 자원이 비디오 카드, 하드 드라이브 혹은 심지어 메모리를 의미 하기도 한다. 프로그램들은 종종 이러한 자원을 위해 경쟁한다. 우리가 어떤 문서를 저장했다면, updateb가 저장한 내용을 데이터베이스에 업데이트 했을 것이다. 이때 여러분의 vim과 updateb는 동일한 하드드라이브를 동시에 사용하게 된다. 커널은 모든 것들이 올바르고 깔끔하게 정렬되길 원하므로 우리와 같은 유저들에게 위와 같은 상황에서 자원에 대한 접근권한을 부여하지 않을 것이다. 이러한 경쟁상황을 없애기 위해 CPU는 각각의 상황에서 다른 모드로 사용이 가능하다. 각각의 모드는 각자가 하고 싶은 일들을 위해 서로 다른 자유의 정도를 가진다.
인텔 80386 구조는 4개의 모드를 가지는데 각각을 ring이라 부른다. 유닉스는 오직 두 개의 ring을 가지는데 (두개의 모드를 가지는데) 가장 높은 ring, 즉 superviser 모드(ring 0) 와 가장 하부의 링, user 모드(ring 3)을 의미한다.

다시 system call과 라이브러리 함수의 이야기로 돌아가보자. 일반적인 경우에 여러분은 user mode에서 라이브러리 함수들을 사용할 것이다. 그리고 사용된 라이브러리 함수는 여러개의 system call을 호출할 것이며 이러한 system call은 라이브러리 함수의 뒤에서 실행될 것인데 커널의 한부분에서 superviser mode로 실행이 될 것이다. 호출된 system call은 각자의 작업이 끝나면 다시 user mode로 돌아 오게 된다.

3.1.4 Name Space

여러분이 작은 c 프로그램을 작성할 때, 여러분은 가독성을 보장하는 변수를 사용할 것이다. 만일 반면에, 여러분이 다른 사람의 전역변수에 해당하는 어떠한 전역변수 명을 다시 사용한다면 서로의 변수 이름 때문에 충돌이 일어나 프로그램에 큰 문제가 될 것이다. 만약 여러분이 쉽게 구별할 수 없는 이상한 이름의 전역변수가 많은 프로그램을 작성중이라면 여러분은 namespace pollution 이라는 문제를 가진 프로그램을 작성하고 있는 것이다. 큰 프로젝트에서 변수에 대한 여분의 이름에 대해 기억하고, 각자의 변수명과 심볼명을 만드는 틀에 대한 고민이 반드시 필요하다.

커널 코드를 작성할 때도, 심지어 그것이 작디 작은 모듈이라 할지라도 작성된 모듈은 전체의 커널로 linking 될 것인데, 위에서 언급한 부분을 고려한다면 이것은 분명히 주목할만한 문제가 된다. 위와 같은 문제점을 피하는 가낭 좋은 방법은 여러부만의 잘 정의된 prefix를 사용하고 모든 변수를 static 으로 선언 하는 것이다. 전통적으로 커널의 prefix는 소문자로 정의되었다. 여러분이 모든 변수들을 static 으로 선언 하기 싫다면 여러분의 symbol table을 정의해 커널에 등록하는 방법이 있다. 우리는 이것에 대해 차후에 다루어 볼 것이다.

/proc/kallsyms 파일은 커널에서 사용하고 있는 모든 symbol 에 관한 데이터를 가지고 있기 때문에 위를 통해서 사용 가능한 symbol 들을 알아낼수 있다.

3.1.5 Code Space

메모리 관리는 매우 복잡한 주제이다. 우리는 메모리 관리에 대해 전문가가 아니지만, 실제 모듈 작성을 위해 몇가지의 사실에 대해 알고 있어야 한다.

만일 여러분이 segfault의 실제 의미에 대한 생각을 해본적이 없다면 포인터가 실제로 메모리의 위치를 가리키지 않는다는 사실에 대해 놀랄 것이다. 프로세스가 만들어 졌을 때, 커널은 실제 메모리를 프로세스에게 할당한다. (우리가 알고 있는 코드의 실행, 변수, 스택, 힙등의 것들을 위해) 메모리는 0x00000000 으로 시작해 필요한 만큼 주소를 확장한다. (여기서는 "내 컴퓨터는 4GB의 메모리를 가지고 있는데, 무한정 확장될수 있다는 말인가?" 라고 말하는 것이 아니다. 어떤 프로세스를 위해 메모리가 0x00000000 이라는 물리적 주소로부터 시작된다는 그 사실이 중요하고 그것을 말하는 것이다.) 프로세스를 위해 할당된 메모리 주소는 서로 겹치지 않는다. (겹치게 되면 심각한 문제가 발생하게 될 것이다.) 다시 풀어서 말하면, 각각의 프로세스가 실제로 0xbffff978 이라는 메모리 주소를 접근하지만, 실제 접근하는 물리적인 메모리의 위치는 다르다는 것이다. 즉, 프로세스는 각각을 위해 할당된 실제 물리적 메모리의 접근을 위해 offset의 개념으로 위와 같은 주소를 사용하는 것이다. 대부분의 경우에서 각각의 프로세스는 다른 프로세스의 메모리 영역에 접근하지 못한다. 하지만 그러한 방법들이 있는데 그것들은 차후에 다루도록 하자.

커널은 위의 논리에 맞게 자신만의 메모리 영억을 가지고 있다. 모듈은 커널에 동적으로 탑재되고 제거될수 있으므로(semi-autonomous라고 할수 있다 :: 사용자의 요구에 의해서 탑재 혹은 제거 되므로) 각각의 모듈들은 자신만의 메모리보다 커널의 code space를 공유한다. 그러므로, 여러분의 모듈이 segfault 상태를 발생시킨다면 우리들의 커널 역시 segfault의 상태에 빠지게 된다. 그리고 만약 여러분이 이러한 상태에서 데이터를 저장한다면 커널의 데이터는 손상될 것이다. 위의 상태는 실제로 엄청나게 나쁜 상황이므로, 여러분은 위와 같은 상황이 발생하지 않도록 매우 주의해야 한다.

그런데, 단일 커널을 사용하는 커널이라면 위와 같은 상황이 분명 발생할수 있다. 그리고 microkernel 이라 불리는 것이 있는데 위의 커널은 각자만의 작은 codespace를 가지고 있다. GNU hurd 와 QNX Neutrino가 microkernel의 한 예이다.

3.1.6 Device Drivers

드라이버는 모듈의 종류중 하나인데, 시리얼 포트나 TV 수신가트와 같이 하드웨어를 위한 기능을 제공한다. 유닉스에서 각각의 하드웨어는 /dev 폴더에 위치해있는 파일 이름으로 나타내지고, 그 파일들은 하드웨어와의 의사소통 수단을 제공한다. 그리하여, 드라이버는 유저 프로그램의 안 보이는 곳에서 하드웨어와의 통신이 가능하도록 하는 역할을 한다.

3.1.6.1 Major and Minor Numbers

몇 개의 장치 파일들을 살펴보자. 아래에 3개의 IDE hard drive에 대한 정보가 나타나 있다.

```
# ls -l /dev/hda[1-3]
brw-rw---- 1 root disk 3, 1 Jul 5 2000 /dev/hda1
brw-rw---- 1 root disk 3, 2 Jul 5 2000 /dev/hda2
brw-rw---- 1 root disk 3, 3 Jul 5 2000 /dev/hda3
```

컴마 이후의 숫자가 보이는가? 첫 번째 번호는 장치의 major number 을 나타낸다. 두 번째 번호는 minor number을 나타낸다. major number는 여러분에게 어떤 드라이버가 하드웨어에 접근하기위해 사용되는가를 나타낸다. 각각의 드라이버는 고유의 major number가 부여되어 있다. 만일 모든 major number가 같다면 해당 major number를 가진 장비들은 모두 같은 드라이버에 의해 컨트롤 됨을 의미한다. 위의 예에서, 모든 major number 가 3이므로, 하드 드라이브는 모두 같은 드라이버에 의해 컨트롤 됨을 알 수 있다.

minor number 는 같은 드라이버에 의해 컨트롤 되는 장치들을 구별하기 위한 용도로 사용된다. 위의 예로 돌아가서, 비록 모든 하드들이 같은 드라이버에 의해 컨트롤 되지만 각자가 고유의 하드웨어로 구분됨을 알수 있다.

장치들은 두 가지의 타입 : character device 와 block device로 나뉜다. block device는 요청에 대한 버퍼가 존재해서 각각의 요청에 대해 가장 최적의 요청을 고를수가 있다. 이 부분은 저장장치에서 매우 중요한 부분인데 읽거나 쓰는 요청이 들어왔을 때 현제의 섹터 위치보다 먼쪽이 아닌 가까운 쪽을 먼저 처리하는 것이 더욱 빠르게 일을 처리해 낼수 있기 때문이다. 또 다른 차이점은 block device는 input에 대해 block 단위의 output (각각의 장비에 따라 block 의 크기는 다르다.)을 출력하지만, character device는 많거나 혹은 단순히 몇 바이트 정도의 출력물을 낼수 있다는 것이다. (block device보다 결과물의 단위가 유동적이다.) 현제 존재하는 대부분의 장치들은 character device 인데 고정된 block size 단위의 출력물이나 버퍼링 같은 동작이 필요하지 않기 때문이다. 여러분은 어떠한 디바이스 파일이 character device를 위한것인지, 아니면 block device를 위한것인지 에 대해 ls -l 명령어의 출력물의 첫 번째 문자를 보고 알수 있는데, 'b' 는 block device를, 'c' 는 character device를 의미한다. 위의 예는, 모든 디바이스가 block device인데, 아래의 예선 모든 디바이스가 character device이다. (시리얼 포트를 의미함)

```
# ls -l /dev/hda[1-3]
brw-rw---- 1 root disk 3, 1 Jul 5 2000 /dev/hda1
brw-rw---- 1 root disk 3, 2 Jul 5 2000 /dev/hda2
brw-rw---- 1 root disk 3, 3 Jul 5 2000 /dev/hda3
```

디바이스에 할당된 major number 들을 확인하고 싶다면, **/usr/src/linux/Documentation/devices.txt** 를 확인하면 된다.

시스템에 설치된 모든 디바이스 파일들은 **mknod** 명령어를 통해 생성되었다. 우리가 major number로 12, minor number로 2를 가지는 coffee 라는 이름의 character device를 생성하기 위해선 간단하게 **mknod /dev/coffee c 12 2** 라는 명령어를 입력하면 된다. 여러분이 직접 디바이스 파일을 **/dev** 폴더에 넣을 필요는 없다. 보통의 경우에 /dev 폴더에 디바이스 파일을 넣지만, 여러분이 테스트 목적으로 디바이스 파일을 만들었을 경우엔 여러분이 커널 모듈의 컴파일 작업을 하는 폴더에 집어넣어도 관계없다. (즉, 어떠한 폴더에서 작업을 하더라도 관계 없다.) 다만, 작성이 완료된 디바이스 파일만 올바른 폴더에서 관리하면 된다.

위의 주제에서 명확하게 다뤄지지 않았던 몇가지의 사항들에 대해 조금 더 구체적으로 언급을 해보자. 디바이스 파일에 어떠한 장치가 접근하려 할때, 커널은 디바이스 파일의 major number를 사용해 어떤 드라이버가 위의 디바이스 파일에 대한 접근을 제어하기 위해 사용되는지 판단한다. 이 말은, 커널이 minor number을 사용하지 않거나, 혹은 이 숫자가 커널에 아무런 의미가 없는 숫자라고 말할 수 있다. 유일하게 드라이버 자신만이 minor number를 사용해 여러 가지 다른 하드웨어들을 구별한다.

그리고 한편, 여기서 저자가 말한 하드웨어 란 것은 PCI 카드와 같은 것이 아니라 조금은 추상적인 상태의 하드웨어 자체를 의미한다. 아래애 두가지 디바이스 파일을 살펴보자.

```
% ls -l /dev/fd0 /dev/fd0u1680
brwxrwxrwx 1 root floppy 2, 0 Jul 5 2000 /dev/fd0
brw-rw---- 1 root floppy 2, 44 Jul 5 2000 /dev/fd0u1680
```

여러분은 위의 두 개의 디바이스가 block device 이며, 같은 드라이버를 통해 컨트롤 됨을 알 수 있다. 또한 위의 장치는 플로피 디스크를 의미함을 알 수 있다. 여러분이 하나의 플로피 디스크를 가지고 있다고 하더라도 위와 같은 결과가 나올 것이다. 왜일까?

하나는 플로피 드라이브가 1.44MB의 저장소 임을 의미한다. 나머지의 파일은 1.68MB의 'superformatted' 플로피 디스크를 의미하고, 1.44MB의 'formatted' 플로피 드라이브보다 조금 더 많은 양의 데이터를 저장할 수 있다. 위의 예로부터 우리는 두 개의 다른 minor number를 가지고 있는 하나의 플로피 디스크 드라이브 라는 사실을 알 수 있다. 그래서 저자가 조금은 추상적인 의미의 하드웨어 라는 말을 위에서 언급한 것이다.

Chapter4. Character Device Files

4.1 Character Device Drivers

4.1.1 The file_operations Structure

file_operations 구조체 는 /linux/fs.h 에 정의되어 있고, 함수를 가리키는 포인터들을 가지고 있는데 위의 함수들은 디바이스에서 수행되는 다양한 기능을 구현한 것이다. 구조체의 각각의 부분은 드라이버가 정의한 어떤 함수들의 주소에 대응되는데 이는 디바이스에 요청된 여러 가지의 작업들을 다루기 위해서 이다.

예를 들어, 모든 character driver 는 디바이스로부터 읽는 역할을 하는 함수를 필요로 한다. file_operations 구조체는 이러한 기능을 수행하는 함수들의 주소를 가지고 있다. 아래에 커널 2.6.5 버전의 file_operations 구조체를 살펴보자.

```
struct file_operations {
        struct module *owner;
        loff_t(*llseek) (struct file *, loff_t, int);
        ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t(*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t(*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
        ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area) (struct file *, unsigned long,unsigned long, unsigned long, unsigned long);
};
```

드라이버가 구현하지 않는 몇몇 기능들도 존재한다. 예를 들어, 비디오 카드를 다루는 드라이버는 directory 구조체를 읽어올 필요가 없다. file_operations 구조체에서도 이에 대응하는 기능의 진입점은 NULL로 셋팅되어야 할 것이다. 아래애 구조체를 조금 더 편하게 할당하는 gcc extension을 살펴보자. 여러분은 최근의 드라이버들에서 이러한 형태들을 발견할수 있을 것이다.

```
struct file_operations fops = {
read: device_read,
write: device_write,
open: device_open,
release: device_release
};
```

그러나 C99 라는 다른 방법도 있는데, GNU extension 이전에 선호되던 방식이다. 저자가 사용한 gcc의 버전은 2.95인데, 새로운 C99 문장들을 허용한다. 여러분은 다른 어떤이가 여러분의 드라이버를 복사하길 원한다면 아래와 같은 문장을 사용해야 한다.

```
struct file_operations fops = {
.read = device_read,
.write = device_write,
.open = device_open,
.release = device_release
};
```

좀더 의미가 분명해졌다. 그리고 구조체의 멤버중 명확히 하지 않은 모든 멤버들은 gcc에 의해 NULL로 초기화 될 것이다.

4.1.2 The file structure

각각의 디바이스들은 linux/fs.h 에 정의된 커널의 file 구조체로 표현될 수 있다. 하지만 위의 구조체는 커널 수준에서 사용되므로 유저레벨의 사용 환경 에서는 확인 할 수 없음에 유의하자. glibc에 정의되고 커널공간의 함수에서 사용되지 않는 FILE과는 다른 것이다. 그리고 위에서도 볼수 있듯이 file 이라는 이름은 잘못 지어졌다. 위의 file은 디스크 상의 파일이 아니라 추상적으로 파일을 '여는' 것을 나타낸다.

file 구조체의 객체는 보통 flip 이라 불린다. 그리고 여러분은 struct file file 이라는 구조체 역시 확인할 수 있을 것이다. 햇갈리지 않도록 주의하자.

이제 file의 정의를 살펴보자. struct dentry 와 같이 디바이스 드라이버가 사용하지 않는 많은 진입점들은 무시해도 좋다. 왜나면 드라이버는 file에 포함된 구조체들만 사용할 뿐, 직접적으로 file에 접근하지 않기 때문이다.

4.1.3 Registering A Device

이미 언급하였지만, character device들은 주로 /dev 파일에 있는 디바이스 파일들을 통해 접근할 수 있다. 디바이스 파일의 major number 는 어떤 드라이버가 어떤 디바이스 파일을 다루는지 나타내 준다고 했고, minor number 는 사용중인 장치에서 드라이버가 하나 이상의 디바이스를 관리 할 때 의미가 있다고 했다.

여러분의 시스템에 드라이버를 설치 한다는 것은 커널에 등록을 해야 한다는 것을 의미 한다. 이 말은 모듈의 초기화 동안 major number를 드라이버에 할당 한다는 것과 같은 의미이다. 여러분은 linuxx/fs.h 에 정의되어 있는 register_chrdev 함수를 통해 위와 같은 동작을 수행 할 수 있다.

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

인자중에서 unsigned int major 는 여러분이 요청한 major number를 의미하고, const char *name 은 /proc/devices 에 나타날 디바이스의 이름을 의미하며 struct file_operations *fops 는 여러분의 드라이버를 위한 file_operations 테이블을 가리키는 포인터를 의미한다. 이 함수의 리턴값중 음수의 의미는 드라이버의 등록이 정상적으로 수행되지 않았음을 의미한다. 우리가 register_chrdev 으로 minor number를 넘기지 않았음을 떠올려 보자. 그 이유는 커널은 minor number를 사용하지 않고 단지 드라이버 자신의 구별을 위해서만 사용되기 때문이였다.

이제 질문은 "어떻게하면 중복되지 않는 major number를 알아 낼 수 있을까?" 이다. 가장 쉬운 방법중 하나는 Documentation/devices.txt 파일을 살펴서 사용되지 않는 major number를 알아 내는 것인데, 이 방법은 "여러분이 선택한 번호가 다른 장치에 의해서 조금 나중에 등록되지 않는다" 라는 점을 확실히 보장할수 없기 때문에 좋은 방법이 아니라고 말할수 있다. 가장 좋은 방법은 커널의 dynamic major number를 확인 하는 것이다.

여러분이 register_chrdev 함수를 통해 커널에 0을 전달 하면, 커널은 동적으로 할당한 major number를 리턴할 것이다. 아직 남아있는 문제점은 여러분이 major number를 알지 못한다면 디바이스 파일을 만들 수 없다는 점이다. 이문제에 대해 여러 가지 해결책들이 있는데, 그 중 하나는 드라이버가 새로 할당한 번호를 출력할수 있다는 점을 이용해 여러분이 디바이스 파일을 직접 만드는 것이다. 두 번째는 새로 등록된 디바이스는 /proc/devices 에 엔트리를 가지고 있을 것이다. 그 정보를 활용해 직접 디바이스 파일을 제작 하거나 혹은 쉘 스크립트를 작성해 파일을 읽고 디바이스 파일을 만드는 것이다. 저자가 소개할 마지막 세 번째 방법은 디바이스 등록 후 mknod system call을 이용해 디바이스 파일을 만들고 cleanup_module를 호출할 동안 지우는 것이다.

4.1.4 Unregistering A Device

커널 모듈이 메모리에 올라가 있는 중에 다시 그 커널모듈을 언로드 하는 것을 허용하는건 매우 위험한 생각이다. 만약 디바이스 파일을 프로세스가 사용중일 때 여러분이 적절한 함수를 특정메모리 위치에 호출하는 파일을 통해 여러분이 커널 모듈을 제거 한다면 어떤 결과가 일어날까? 운이 좋다면, 다른 어떤 코드가 그 위치에 로딩되어 있지 않아서 다행히 에러 메시지를 하나쯤 보고 끝날 수도 있지만, 운이 없는 경우라면, 어떤 다른 커널이 현 커널 내부의 어떤 함수의 한 중간의 자리로 로딩됨을 의미하고, 이것에 대한 결과는 상상할 수 없을 만큼 시스템에 치명 적일 것이다.

일반적으로, 여러분이 뭔가를 허락하고 싶지 않을 경우엔 함수를 통해 에러코드를 리턴 할 것이다. 물론 cleanup_module 함수는 void 함수이기 때문에 에러코드를 리턴할 수는 없다. 하지만 시스템엔 어떤 프로세스가 여러분의 모듈을 사용하는가에 대해 기록하는 부분이 있는데, 여러분은 그 정보를 /proc/modules 의 세 번째 필드를 통해 확인 할수 있다. 만약 그 필드가 0 이라면, rmmod 는 실행되지 않을 것이다. 이렇게 확인 하는 경우에 여러분이 cleanup_module 에 대한 필드를 체크할 필요가 없음을 기억하자. 그 이유는 위의 필드 값의 확인은 linux/module.c 에 정의되어 있는 sys_delete_module 함수의 호출을 통해 수행 될 것이기 때문이다. 여러분은 또한 위의 값의 변경을 위해 매번 위와 같은 파일에 직접적으로 접근 할 필요가 없다. 위의 기능을 위해 linux/module.h 에 몇가지 함수가 정의 되어 있기 때문이다.

- try_module_get(THIS_MODULE): Increment the use count.
- module_put(THIS_MODULE): Decrement the use count.

위의 필드 값을 정확하게 유지하는 것은 매우 중요한 일이다. 만약 여러분이 위의 필드값을 정확하게 유지 하거나 사용하지 못한다면, 여러분은 모듈을 절대로 언드로 하지 못할 것이다. 이 문제는 곧 여러분이 실제 모듈들을 개발하는 과정에서 실질적인 문제로 다가올 것이다.

4.1.5 chardev.c

아래에 있는 예제 코드는 chardev 라고 불리는 character device를 만드는 코드이다. 여러분은 이 장치의 디바이스 파일을 cat을 사용하거나 어떤 프로그램을 통해 열수가 있고, 드라이버는 위의 디바이스 파일이 몇 번이나 사용됬는지(읽기를 목적으로) 그에 대한 정보를 기록할 것이다. 기본적으로 디바이스 파일을 “echo "hi" >/dev/hello" 와 같은 형식으로 직접적으로 수정 혹은 쓸 수 있는 기능은 지원되지 않지만 (루트권한이 없는이상 쓰기가 불가능하다) 이러한 시도를 할 경우에 위와 같은 기능을 지원하지 않는다고 알려주는 메시지가 출력될 것이다. 우리가 버터를 통해 읽는 데이터가 보이지 않는다고 걱정할 필요는 없다. 이 가이드 북에선 그렇게 깊은 내용을 다루지는 않을 것이고 단지 우리가 받을수 있는 형태로 간단한 데이터와 메시지를 읽고 출력 하는 정도의 실습을 할것이기 때문이다.

Example 4-1. chardev.c

```
/*
* chardev.c: Creates a read-only char device that says how many times
* you've read from the dev file
*/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h> /* for put_user */
/*
* Prototypes - this would normally go in a .h file
*/
```

```
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices */
#define BUF_LEN 80 /* Max length of the message from the device */

/*
* Global variables are declared as static, so are global within the file.
*/
static int Major; /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open?

/* Used to prevent multiple access to device */
static char msg[BUF_LEN]; /* The msg the device will give when asked */
static char *msg_Ptr;
static struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};

/*
* This function is called when the module is loaded
*/
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);
        if (Major < 0) {
                printk(KERN_ALERT "Registering char device failed with %d₩n", Major);
        return Major;
        }
        printk(KERN_INFO "I was assigned major number %d. To talk to₩n", Major);
        printk(KERN_INFO "the driver, create a dev file with₩n");
        printk(KERN_INFO "'mknod /dev/%s c %d 0'.₩n", DEVICE_NAME, Major);
        printk(KERN_INFO "Try various minor numbers. Try to cat and echo to₩n");
        printk(KERN_INFO "the device file.₩n");
        printk(KERN_INFO "Remove the device file and module when done.₩n");
        return SUCCESS;
}

/*
* This function is called when the module is unloaded
*/
void cleanup_module(void)
{
        /*
        * Unregister the device
        */
        int ret = unregister_chrdev(Major, DEVICE_NAME);
        if (ret < 0)
                printk(KERN_ALERT "Error in unregister_chrdev: %d₩n", ret);
        }
        /*
        * Methods
        */

        /*
        * Called when a process tries to open the device file, like
        * "cat /dev/mycharfile"
        */
```

```
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;
        if (Device_Open)
                return -EBUSY;
        Device_Open++;
        printf(msg, "I already told you %d times Hello world!₩n", counter++);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);
        return SUCCESS;
}

/*
* Called when a process closes the device file.
*/
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--; /* We're now ready for our next caller */
        /*
        * Decrement the usage count, or else once you opened the file, you'll
        * never get get rid of the module.
        */
        module_put(THIS_MODULE);
        return 0;
}

/*
* Called when a process, which already opened the dev file, attempts to
* read from it.
*/
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h */
char *buffer, /* buffer to fill with data */
size_t length, /* length of the buffer */
loff_t * offset)
{
        /*
        * Number of bytes actually written to the buffer
        */
        int bytes_read = 0;

         /*
        * If we're at the end of the message,
        * return 0 signifying end of file
        */
        if (*msg_Ptr == 0)
                return 0;

         /*
        * Actually put the data into the buffer
        */
        while (length && *msg_Ptr) {
                /*
                * The buffer is in the user data segment, not the kernel
                * segment so "*" assignment won't work. We have to use
                * put_user which copies data from the kernel data segment to
                * the user data segment.
                */
                put_user(*(msg_Ptr++), buffer++);
                length--;
                bytes_read++;
        }
        /*
        * Most read functions return the number of bytes put into the buffer
        */
        return bytes_read;
}

/*
* Called when a process writes to dev file: echo "hi" > /dev/hello
*/
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        printk(KERN_ALERT "Sorry, this operation isn't supported.₩n");
        return -EINVAL;
}
```

4.1.6 Writing Modules for Multiple Kernel Version

커널이 프로세스에 대한 정보를 출력하는 가장 주요한 방식인 system call 은 일반적으로 여러 커널 버전에서 똑같게 유지되어 왔다. 어떠한 새로운 시스템이 추가되더라도, 이전의 시스템은 본래 하던방식과 동일하게 동작할 것이다. 새로운 커널버젼이 일반적으로 실행되던 프로세스의 실행을 방해하지 않는 이러한 방식은 시스템간 호환을 위해 필수적이다. 대부분의 경우에서 디바이스 파일은 커널의 버전이 바뀌더라도 동일하게 유지된다. 반면에 커널 내부의 인터페이스들은 버전에 따라 달라지게 된다.

리눅스 커널 버전은 stable version (배포판, 안정된 버전 :: n.$짝수번호#.m) 과 development version (개발버전, 테스트 중인 버전 :: n.$홀수번호$.m) 으로 나뉜다. 개발버전은 새로운 개념을 포함하고 있는데, 당연히 새로 적용되는 기능들에 대해 여러 가지 테스트 목적으로 배포되어 다음 버전을 위해 지속적으로 재구현된다. 결과적으로 여러분들은 지속적으로 바뀌는 개발버전들이 동일한 내부 인터페이스를 가질 것이라는 것을 보장할 수가 없게 된다. 물론 이러한 가이드북 역시 매번 버전이 바뀔때마다 업데이트 될수 없는 이유도 여기에 있다. 반면에 배포판에서는, 내부적으로 동일한 커널 인터페이스를 가질 것 이라고 생각해도 된다.

커널 버전마다 여러 가지 차이점이 있고, 여러분이 여러 가지 커널 버전을 지원하는 모듈을 작성하고 싶다면, 여러분은 우리가 흔히 말하는 조건부 컴파일 지시자 (c에서 #if, #ifdef 와 같은)를 사용해서 코드를 작성해야 한다. 이렇게 하는 이유는 LINUX_VERSION_CODE와 KERNEL_VERSION 매크로의 비교를 위해서 이다. 커널 버전 a / b / c 에서는 위의 매크로 버전이 $ 2^(16)a + 2^(8)b + c $ 일 것이다.
반면에 이 가이드의 이전 버전에서는 이러한 하위 호환성을 위한 코드 작성방법에 대해 세밀하게 다루었지만, 이번엔 그러한 내용을 다루지 않았다. 모듈 작성에 관심이 있는 사람은 자신의 커널 버전에 맞는 LKMPG 가이드 북을 알아서 찾아 볼 것이다. 그리하여 우리도 커널과 비슷하게 LKMPG의 업데이트를 하기로 결정했다. (물론 배포판 커널의 세부 버전차이까지는 고려하지 않을 것이다.) 우리는 LKMPG의 versioning을 위해 patchlevel을 사용할 것이고, 그렇다면 LKMPG version 2.4.x 는 커널 2.4.x를, LKMPG version 2.6.x 는 커널 2.6.x 의 내용을 다루게 될 것이다. 또한 각각 LKMPG의 버전에 맞추어 여러분의 커널을 업데이트 하는등의 준비를 할 것을 당부하겠다.

Chapter5. The /proc File System

5.1 The /proc File System

리눅스에는 커널과 커널모듈을 위한 새로운 매커니즘이 추가되었는데, 이는 프로세스들에게 정보를 전달하기 위한 것이다. 이를 /proc 파일 시스템 이라고 하는데, 원래 프로세스 정보에 대해 쉽게 접근하기 위해 제작되었고 최근엔 커널에서 /proc/meminfo 의 메모리 사용 통계 내용이나 /proc/modules 와 같이 모듈의 리스트들에 대한 정보와 같이 사용가자 관심을 가질만한 거의 대다수의 정보를 다루는데 사용된다.

proc 파일시스템의 사용방법은 디바이스 드라이버들과 상당부분 유사하다. 핸들러 함수들을 (우리가 배울 proc 파일시스템의 구조체는 오직 한 개의 포인터만을 제공하는데, 이것은 사용자가 /proc 파일을 읽으려고 할 때 호출된다.) 조작하기 위한 포인터를 포함해 /proc 파일이 필요로 하는 모든 정보를 가진 구조체가 만들어진다. 그리고 init_module 이 커널에 위의 구조체를 등록하고 cleanup_module을 이용해 제거한다.

우리가 proc_register_dynamic을 사용하는 이유는 우리가 실제로 사용되는 inode 번호를 정하기 싫어서가 아니라 커널 스스로 inode 번호로 인한 충돌을 방지하기 위해서 이다. 일반적인 파일 시스템들은 일반적인 메모리(/proc 이 위치한) 보다 디스크에 위치하는 경우가 많고 그래서 inode 번호는 위의 파일들의 index-node가 위치한 디스크의 어떤 위치를 가리키는 포인터의 역할을 한다. inode 는 파일의 허가에 관려된 정보와 같은 파일 정보를 디스크 위치정보 혹은 파일의 데이터가 위치한 정보와 함께 가지고 있다.

사용자가 매번 파일이 열리고 닫힐 때 마다 신경을 쓰는 경우가 없도록 try_module_put 이나 try_module_get 과 같은 매크로는 존재하지 않고, 또한 파일이 열린 후 모듈이 제거되더라도 그에 따르는 결과를 막을 방법 역시 존재하지 않는다.

아래에 /proc 파일을 어떻게 사용하는 것인가에 대한 간단한 예를 실었다. 아래의 예는 /proc 파일 시스템을 위한 HelloWorld와 같은 간단한 예인데, (본문에서는 Hello World를 보통명사처럼 사용함..) 세 부분으로 나누어져 있다 :: /proc 폴더에 init_module 함수를 포함한 helloworld 파일을 만들고, procfs_read 함수를 통해 /proc/helloworld 파일이 사용중일 때 값을 리턴하는 부분, 그리고 cleanup_module을 사용해 /proc/helloworld 파일을 삭제 하는 부분이다.

매번 /proc/helloworld 파일이 사용될 때 마다, procfs_read 함수가 호출될 것이다. 이 함수의 인자중 첫 번째와 세 번째 인자값은 매우 중요한데, 첫 번째 인자는 버퍼를, 세 번째 인자는 오프셋을 의미한다. 버퍼의 내용은 위 파일을 사용할 어플리케이션(cat과 같은)으로 들어가게될 버퍼값이고, 오프셋은 사용중인 파일에서의 위치를 의미한다. 만약 위의 함수(procfs_read) 의 리턴값이 null 이 아니라면, 또다시 호출될 것이다. 즉, 리턴값에 주의해서 위의 함수를 사용해야 한다는 것이다.

```
% cat /proc/helloworld
HelloWorld!
```

Example 5-1. procfs1.c

```
/*
* procfs1.c - create a "file" in /proc
*
*/
#include <linux/module.h> /* Specifically, a module */
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/proc_fs.h> /* Necessary because we use the proc fs */
```

```
#define procfs_name "helloworld"

/**
* This structure hold information about the /proc file
*
*/
struct proc_dir_entry *Our_Proc_File;

/* Put data into the proc fs file.
*
* Arguments
* =========
* 1. The buffer where the data is to be inserted, if
* you decide to use it.
* 2. A pointer to a pointer to characters. This is
* useful if you don't want to use the buffer
* allocated by the kernel.
* 3. The current position in the file
* 4. The size of the buffer in the first argument.
* 5. Write a "1" here to indicate EOF.
* 6. A pointer to data (useful in case one common
* read for multiple /proc/... entries)
*
* Usage and Return Value
* ======================
* A return value of zero means you have no further
* information at this time (end of file). A negative
* return value is an error condition.
*
* For More Information
* ====================
* The way I discovered what to do with this function
* wasn't by reading documentation, but by reading the
* code which used it. I just looked to see what uses
* the get_info field of proc_dir_entry struct (I used a
* combination of find and grep, if you're interested),
* and I saw that it is used in <kernel source
* directory>/fs/proc/array.c.
*
* If something is unknown about the kernel, this is
* usually the way to go. In Linux we have the great
* advantage of having the kernel source code for
* free - use it.
*/
int procfile_read(char *buffer, char **buffer_location, off_t offset, int buffer_length, int *eof, void *data)
{
        int ret;
        printk(KERN_INFO "procfile_read (/proc/%s) called₩n", procfs_name);
        /*
        * We give all of our information in one go, so if the
        * user asks us if we have more information the
        * answer should always be no.
        *
        * This is important because the standard read
        * function from the library would continue to issue
        * the read system call until the kernel replies
        * that it has no more information, or until its
        * buffer is filled.
        */
        if (offset > 0) {
                /* we have finished to read, return 0 */
                ret = 0;
        } else {
                /* fill the buffer, return the buffer size */
        ret = sprintf(buffer, "HelloWorld!₩n");
        }
        return ret;
}
```

```
int init_module()
{
        Our_Proc_File = create_proc_entry(procfs_name, 0644, NULL);
        if (Our_Proc_File == NULL) {
                remove_proc_entry(procfs_name, &proc_root);
                printk(KERN_ALERT "Error: Could not initialize /proc/%s₩n", procfs_name);
                return -ENOMEM;
        }
        Our_Proc_File->read_proc = procfile_read;
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->mode = S_IFREG | S_IRUGO;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 37;
        printk(KERN_INFO "/proc/%s created₩n", procfs_name);
        return 0; /* everything is ok */
}

void cleanup_module()
{
        remove_proc_entry(procfs_name, &proc_root);
        printk(KERN_INFO "/proc/%s removed₩n", procfs_name);
}
```

5.2 Read and Write a /proc File

우리는 /proc/helloworld 파일을 읽는 아주 간단한 /proc 파일의 예를 살펴보았다. (Example 5.1) 또한 /proc 내부에 파일을 쓰는 것 역시 가능하다. 이땐 위의 예와 같이 파일을 읽는 경우와 비슷하게 동작하는데, /proc 내부의 파일이 읽어질 때 함수가 호출된다는 것이다. 그러나 위의 경우와 파일의 읽기 혹은 유저로부터 넘겨진 데이터간엔 약간의 차이점이 존재하는데, 이러한 차이점 때문에 여러분은 copy_from_user 혹은 get_user 함수를 통해 유저 공간의 데이터들을 커널공간으로 import 해야만 한다.

copy_from_user 혹은 get_user 함수가 존재하는 이유는 리눅스 메모리가 단편화 되어있기 때문이다. (다른 프로세서는 다를수도 있지만 일단 인텔 아키텍쳐를 기준으로 하자.) 이것의 의미는 포인터 자체만으로는 메모리의 특정한 부분을 참조할수 없고 단지 메모리 세그먼트에 위치한 부분만 참조가 가능하며 이 때문에 여러분이 어떤 부분의 메모리 세그먼트가 사용가능한지 알아야만 하는 이유가 된다. 커널을 위한 메모리 세그먼트는 단 하나만 존재하며, 나머지는 프로세스들을 위해 할당된다.

하나의 프로세스엔 단 하나의 메모리 세그먼트만 할당되고 사용가능하기 때문에, 여러분이 실행을 위한 일반적인 프로그램을 작성중이라면, 메모리 세그먼트에 대해 걱정할 필요가 없다. 여러분이 커널 모듈을 작성할 때, 일반적으로 여러분은 시스템에 의해 자동적으로 조정되는 커널메모리 세그먼트에 접근하고 싶을 것이다. 그러나 메모리버퍼의 내용이 현제 작동중인 프로세스에서 커널로 전달될 때, 커널함수는 프로세스 세그먼트 안에 존재하는 메모리 버퍼 포인터를 전달받는다. put_user 와 get_user 매크로(함수) 는 여러분이 메모리에 접근할수 있도록 하는 역할을 하는데, 이러한 함수들은 하나의 문자만 다루며, 여러분은 copy_to_user 와 copy_from_user 함수를 통해 여러개의 문자들을 다룰수 있다. 여러분이 유저공간의 데이터를 커널 공간으로 import 하기위한 쓰기함수를 위해 커널공간에 버퍼가 제공되는데 읽기 함수를 위해 제공되는 것이 아니라는걸 알자. (이미 데이터가 커널에 존재하므로 버퍼를 읽기용으로 제공하는건 무의미하다.)

Example 5-2. procfs2.c

```
/**
* procfs2.c - create a "file" in /proc
*
*/
```

```
#include <linux/module.h> /* Specifically, a module */
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/proc_fs.h> /* Necessary because we use the proc fs */
#include <asm/uaccess.h> /* for copy_from_user */

#define PROCFS_MAX_SIZE 1024
#define PROCFS_NAME "buffer1k"
/**
* This structure hold information about the /proc file
*
*/
static struct proc_dir_entry *Our_Proc_File;

/**
* The buffer used to store character for this module
*
*/
static char procfs_buffer[PROCFS_MAX_SIZE];

/**
* The size of the buffer
*
*/
static unsigned long procfs_buffer_size = 0;

/**
* This function is called then the /proc file is read
*
*/
int procfile_read(char *buffer, char **buffer_location, off_t offset, int buffer_length, int *eof, void *data)
{
        int ret;
        printk(KERN_INFO "procfile_read (/proc/%s) called₩n", PROCFS_NAME);
        if (offset > 0) {
                /* we have finished to read, return 0 */
                ret = 0;
        } else {
                /* fill the buffer, return the buffer size */
                memcpy(buffer, procfs_buffer, procfs_buffer_size);
                ret = procfs_buffer_size;
        }
        return ret;
}

/**
* This function is called with the /proc file is written
*
*/
int procfile_write(struct file *file, const char *buffer, unsigned long count, void *data)
{
        /* get buffer size */
        procfs_buffer_size = count;
        if (procfs_buffer_size > PROCFS_MAX_SIZE ) {
                procfs_buffer_size = PROCFS_MAX_SIZE;
        }

        if ( copy_from_user(procfs_buffer, buffer, procfs_buffer_size) ) {
                return -EFAULT;
        }
        return procfs_buffer_size;
}

/**
*This function is called when the module is loaded
*
*/
```

```
int init_module()
{
        /* create the /proc file */
        Our_Proc_File = create_proc_entry(PROCFS_NAME, 0644, NULL);
        if (Our_Proc_File == NULL) {
                remove_proc_entry(PROCFS_NAME, &proc_root);
                printk(KERN_ALERT "Error: Could not initialize /proc/%s₩n",
                PROCFS_NAME);
                return -ENOMEM;
        }

        Our_Proc_File->read_proc = procfile_read;
        Our_Proc_File->write_proc = procfile_write;
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->mode = S_IFREG | S_IRUGO;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 37;
        printk(KERN_INFO "/proc/%s created₩n", PROCFS_NAME);
        return 0; /* everything is ok */
}

/**
*This function is called when the module is unloaded
*
*/
void cleanup_module()
{
        remove_proc_entry(PROCFS_NAME, &proc_root);
        printk(KERN_INFO "/proc/%s removed₩n", PROCFS_NAME);
}
```

5.3 Manage /proc file with standard filesystem

여러분은 /proc 인터페이스를 통한 /proc 파일을 어떻게 쓰고 읽는지에 대해 살펴보았다. 그러나 inodes를 통한 /proc 파일 관리가 가능하다. 가장 흥미로운 부분은 허가와 같은 고급기능의 사용에 있다.

리눅스는 파일시스템 등록을 위한 표준화된 메커니즘이 존재한다. 모든 파일시스템이 inode와 파일 동작들을 다루기 위한 자신만의 함수를 가지게 된 이후로, 위의 모든 함수, file_operations 구조체의 포인터를 포함하는 inode_operations 구조체의 모든 포인터들을 관리하고 다루기위한 특별한 구조체가 존재한다. /proc 내부에서 우리가 새로운 파일을 등록할 때, 어떤 inode_operations 구조체가 사용될 것인지 구체화 할 수 있다. inode_operations 는 procfs_read와 procfs_write 함수의 포인터를 가지고 이쓴 file_operations 구조체의 포인터를 포함하는 구조체이고, 위의 메커니즘을 우리가 사용하는 것이다.

또하나 흥미로운점은 module_permission 함수인데, 이 함수는 /proc 파일과 함께 프로세스가 뭔가를 하려할 때 호출되며, /proc 파일에 접근을 허용할 것인가, 아닌가를 결정한다. 위의 모듈은 기본적으로 현 사용 유저의 uid와 실행을 기반을 두지만, 우리가 원하는 어떠한 것이라도 (어떤 프로세스가 지금 접근하려는 파일을 사용했는지, 오늘의 시스템 시간 혹은 우리가 입력한 마지막 input value등) 판단의 기반으로 삼을수 있다.

커널에서 읽기와 쓰기의 기본적인 역할이 반대라는걸 아는 것은 매우 중요하다. 읽기함수는 output을 위해 사용되는 반면 쓰기 함수들은 input을 위해 사용된다. 이러한 이유는 읽기와 쓰기함수는 유저의 관점에서 작업을 하기 때문이다. - 만약 프로세스가 무언가를 읽으려고 한다면, 그 의미는 커널에게 출력을 요구하는 것이고, 그리고 만약 프로세스가 무엇인가를 쓰는 행동을 한다면, 커널은 이것을 input으로 받기 때문이다.

Example 5-3. procfs3.c

```
/*
* procfs3.c - create a "file" in /proc, use the file_operation way
* to manage the file.
*/
```

```
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <asm/uaccess.h> /* for copy_*_user */

#define PROC_ENTRY_FILENAME "buffer2k"
#define PROCFS_MAX_SIZE 2048

/**
* The buffer (2k) for this module
*
*/

static char procfs_buffer[PROCFS_MAX_SIZE];
/**
* The size of the data hold in the buffer
*
*/

static unsigned long procfs_buffer_size = 0;
/**
* The structure keeping information about the /proc file
*
*/

static struct proc_dir_entry *Our_Proc_File;
/**
* This funtion is called when the /proc file is read
*
*/

static ssize_t procfs_read(struct file *filp, /* see include/linux/fs.h */
                           char *buffer, /* buffer to fill with data */
                           size_t length, /* length of the buffer */
                           loff_t * offset)
{
        static int finished = 0;
        /*
        * We return 0 to indicate end of file, that we have
        * no more information. Otherwise, processes will
        * continue to read from us in an endless loop.
        */
        if ( finished ) {
                printk(KERN_INFO "procfs_read: END₩n");
                finished = 0;
                return 0;
        }
        finished = 1;

        /*
        * We use put_to_user to copy the string from the kernel's
        * memory segment to the memory segment of the process
        * that called us. get_from_user, BTW, is
        * used for the reverse.
        */
        if ( copy_to_user(buffer, procfs_buffer, procfs_buffer_size) ) {
                return -EFAULT;
                }
        printk(KERN_INFO "procfs_read: read %lu bytes₩n", procfs_buffer_size);
        return procfs_buffer_size; /* Return the number of bytes "read" */
}
```

```
/*
* This function is called when /proc is written
*/
static ssize_t
procfs_write(struct file *file, const char *buffer, size_t len, loff_t * off)
{
        if ( len > PROCFS_MAX_SIZE ) {
                procfs_buffer_size = PROCFS_MAX_SIZE;
        }
        else {
                procfs_buffer_size = len;
        }
        if ( copy_from_user(procfs_buffer, buffer, procfs_buffer_size) ) {
        return -EFAULT;
}
printk(KERN_INFO "procfs_write: write %lu bytes₩n", procfs_buffer_size);
return procfs_buffer_size;
}
/*
* This function decides whether to allow an operation
* (return zero) or not allow it (return a non-zero
* which indicates why it is not allowed).
*
* The operation can be one of the following values:
* 0 - Execute (run the "file" - meaningless in our case)
* 2 - Write (input to the kernel module)
* 4 - Read (output from the kernel module)
*
* This is the real function that checks file
* permissions. The permissions returned by ls -l are
* for referece only, and can be overridden here.
*/
static int module_permission(struct inode *inode, int op, struct nameidata *foo)
{
        /*
        * We allow everybody to read from our module, but
        * only root (uid 0) may write to it
        */
        if (op == 4 || (op == 2 && current->euid == 0))
                return 0;
        /*
        * If it's anything else, access is denied
        */
        return -EACCES;
}

/*
* The file is opened - we don't really care about
* that, but it does mean we need to increment the
* module's reference count.
*/
int procfs_open(struct inode *inode, struct file *file)
{
        try_module_get(THIS_MODULE);
        return 0;
}
/*
* The file is closed - again, interesting only because
* of the reference count.
*/
```

```
int procfs_close(struct inode *inode, struct file *file)
{
        module_put(THIS_MODULE);
        return 0; /* success */
}
static struct file_operations File_Ops_4_Our_Proc_File = {
                .read = procfs_read,
                .write = procfs_write,
                .open = procfs_open,
                .release = procfs_close,
};

/*
* Inode operations for our proc file. We need it so
* we'll have some place to specify the file operations
* structure we want to use, and the function we use for
* permissions. It's also possible to specify functions
* to be called for anything else which could be done to
* an inode (although we don't bother, we just put
* NULL).
*/
static struct inode_operations Inode_Ops_4_Our_Proc_File = {
        .permission = module_permission, /* check for permissions */
};

/*
* Module initialization and cleanup
*/
int init_module()
{
        /* create the /proc file */
        Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
        /* check if the /proc file was created successfuly */
        if (Our_Proc_File == NULL){
                printk(KERN_ALERT "Error: Could not initialize /proc/%s₩n",
                PROC_ENTRY_FILENAME);
                return -ENOMEM;
        }
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
        Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
        Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 80;
        printk(KERN_INFO "/proc/%s created₩n", PROC_ENTRY_FILENAME);
        return 0; /* success */
}
void cleanup_module()
{
        remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
        printk(KERN_INFO "/proc/%s removed₩n", PROC_ENTRY_FILENAME);
}
```

procfs에 관한 예제를 조금 더 살펴보고 싶다면, 일단 다음의 내용을 잘 상기하도록 하자. 첫 번째로, 여러 가지 루머가 나오고 있는데 procfs 는 sysfs에 비교해볼 때 너무 오래되고 구식의 방법이라는 말이 있다. 두 번째로는 만약 여러분이 그래도 조금 procfs를 더 연구하고 싶다면, linux/Documentation/DocBook/ 폴더를 통해 살펴볼수 있다는 것이다. make help 명령어를 커널 최상위 디렉토리에서 사용하여 여러분이 좋아하는 포맷으로 변경하는 방법을 알아보도록 하자. 예를들어 make htmldocs 와 같이 말이다. 이러한 메커니즘의 사용을 고려하여 여러분이 필요한 커널 관련정보를 문서화 할수 있을 것이다.

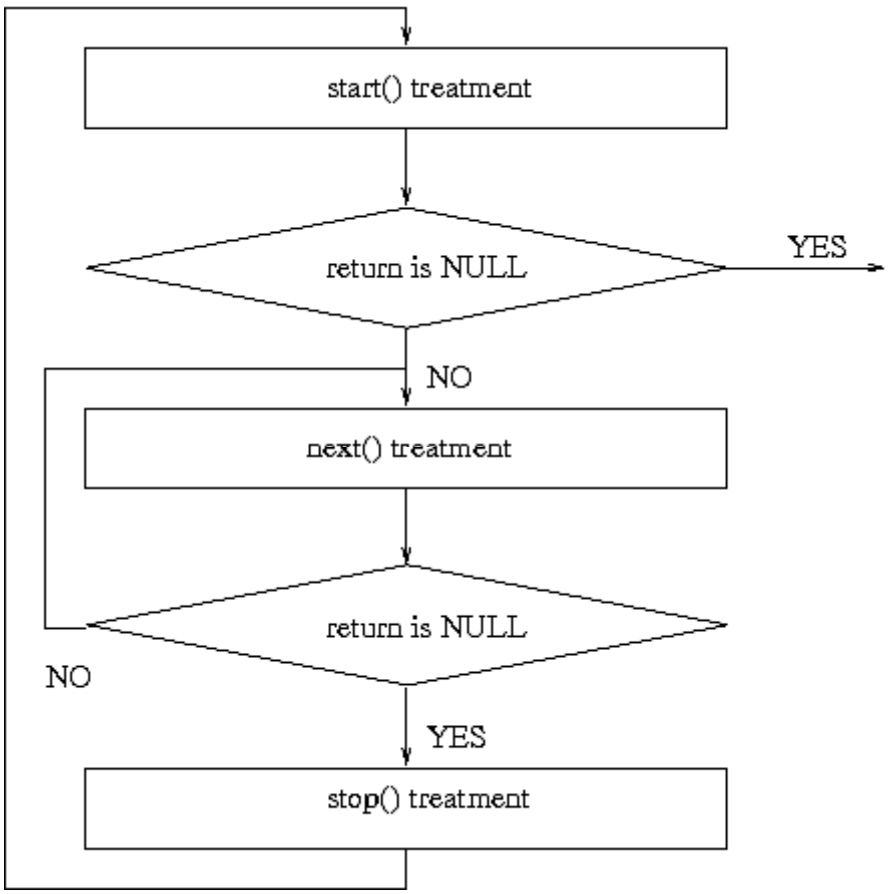
5.4 Manage /proc file with seq_file

이미 우리가 살펴봤듯이, /proc 파일의 작성은 조금 복잡하다. 그래서 seq_file 이라는 API가 존재해서 출력을 위한 /proc 파일의 formatting을 쉽게 해 준다. 위의 API는 아래 3가지의 함수들의 순서를 기반으로 하는데 (start( ), next( ), 그리고 stop( ) 함수이다.) 사용자가 /proc 파일을 읽으려고 할 때, seq_file API가 정해진 순서대로 작동을 하게 된다.

첫 번째로 시작되는 이벤트는 start( ) 함수인데, 만일 이 함수가 NULL을 리턴한다면 다음 이벤트함수인 next( ) 함수가 호출된다. next( ) 함수는 iterator 의 역할을 하며, 모든 데이터를 iterate 하는 목적으로 작동된다. next( ) 함수가 매번 호출될때마다, show( ) 함수역시 호출되는데 유저로부터 읽어진 데이터 값을 버퍼에 쓰는 역할을 한다. next( ) 함수가 NULL을 리턴하면 마지막 이벤트순서로 stop( ) 함수가 호출된다.

주의하자. 만약 위의 순서가 종료된다면 다른 동작을위한 seq_file의 일련의 이벤트가 시작될 것이다. 이것의 의미는, stop( ) 함수가 종료되면 start( ) 함수가 다시 호출된다는 것이다. 위의 루프는 stop( ) 함수가 NULL값을 리턴할 경우 종료된다. 아래의 "How seq_file works" 라는 제목의 표를 살펴보도록 하자.

Figure 5-1. How seq_file works



Example 5-4. procfs4.c

```
/**
* procfs4.c - create a "file" in /proc
* This program uses the seq_file library to manage the /proc file.
*
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <linux/seq_file.h> /* for seq_file */

#define PROC_NAME "iter"
MODULE_AUTHOR("Philippe Reynes");
MODULE_LICENSE("GPL");

/**
* This function is called at the beginning of a sequence.
* ie, when:
* - the /proc file is read (first time)
* - after the function stop (end of sequence)
*
*/
static void *my_seq_start(struct seq_file *s, loff_t *pos)
{
        static unsigned long counter = 0;
        /* beginning a new sequence ? */
        if ( *pos == 0 )
        {
                /* yes => return a non null value to begin the sequence */
                return &counter;
        }
        else
        {
        /* no => it's the end of the sequence, return end to stop reading */
        *pos = 0;
        return NULL;
        }
}

/**
* This function is called after the beginning of a sequence.
* It's called untill the return is NULL (this ends the sequence).
*
*/
static void *my_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
        unsigned long *tmp_v = (unsigned long *)v;
        (*tmp_v)++;
        (*pos)++;
        return NULL;
}
/**
* This function is called at the end of a sequence
*
*/
static void my_seq_stop(struct seq_file *s, void *v)
{
        /* nothing to do, we use a static value in start() */
}
```

```
/**
* This function is called for each "step" of a sequence
*
*/
static int my_seq_show(struct seq_file *s, void *v)
{
        loff_t *spos = (loff_t *) v;
        seq_printf(s, "%Ld₩n", *spos);
        return 0;
}

/**
* This structure gather "function" to manage the sequence
*
*/
static struct seq_operations my_seq_ops = {
                .start = my_seq_start,
                .next = my_seq_next,
                .stop = my_seq_stop,
                .show = my_seq_show
};

/**
* This function is called when the /proc file is open.
*
*/
static int my_open(struct inode *inode, struct file *file)
{
        return seq_open(file, &my_seq_ops);
};

/**
* This structure gather "function" that manage the /proc file
*
*/
static struct file_operations my_file_ops = {
                .owner = THIS_MODULE,
                .open = my_open,
                .read = seq_read,
                .llseek = seq_lseek,
                .release = seq_release
};

/**
* This function is called when the module is loaded
*
*/
int init_module(void)
{
        struct proc_dir_entry *entry;
        entry = create_proc_entry(PROC_NAME, 0, NULL);
        if (entry) {
                entry->proc_fops = &my_file_ops;
        }
        return 0;
}

/**
* This function is called when the module is unloaded.
*
*/
void cleanup_module(void)
{
        remove_proc_entry(PROC_NAME, NULL);
}
```

만약 여러분이 관련 정보를 조금 더 얻고싶다면 아래의 웹페이지를 방문하여 정보를 얻을수 있다.

- http://lwn.net/Articles/22355/
- http://www.kernelnewbies.org/documents/seq_file_howto.txt

그리고 /linux/seq_file 에 관한 정보 또한 찾을수 있을 것이다.

Chapter6. Using /proc For Input

6.1 TODO : Write a chapter about sysfs

이 챕터는 sysfs에 관련 내용을 위한 공간인데, 현제는 비어있다. 여러분들중 sysfs 에 관해 지식을 가지고 있고, 우리와 함께 이 챕터에 관한 내용을 쓰길 원한다면 언제든지 연락을 해주길 바란다.(LKMPG maintainers)

Chapter7. Talking To Device Files

7.1 Talking to Device Files (writes and IOCTLs)

디바이스 파일은 물리적 장치들을 표현하는데 이용된다. 대부분의 물리적 장치들은 입력 뿐만 아니라 출력에도 사용되는데, 프로세스로부터 장치로 출력을 전송하기 위한 커널의 디바이스 드라이버의 메커니즘이 존재한다. 그 메커니즘은 디바이스 파일을 열고 마치 파일에 내용을 기록하듯이 쓰면 된다. 아래의 예에서 위의 메커니즘은 device_write로 구현되어 있다.

하지만 위의 방법은 항상 만족스러운 결과를 가져오지 않는다. 만약 여러분이 시리얼 포트에 연결된 모뎀을 가지고 있다고 가정해보자. (만약에 여러분이 내부모뎀을 가지고 있다고 하더라도, CPU의 관점에선 모뎀이 시리얼 포트에 연결되어 있다고 볼 것이다. 너무 복잡하게 생각하지 말자.) 보통의 일반적인 경우라면 디바이스 파일을 이용해 어떤 내용을 모뎀에게 쓰고(입력의 개념), 모뎀으로부터 나온 출력을 읽을 것이다. (들어온 데이터를 받거나, 혹은 전화선을 이용해 내용을 전송 할 것이다.) 그러나, 이러한 동작 방식은 시리얼 포트 자체와의 데이터 수,송신에 대한 의문점을 낳는다.

유닉스에서는 이러한 의문점을 ioctl (short for Input Output ConTroL)이라는 특별한 함수를 사용하여 해결한다. 모든 장치들은 자신들만의 고유한 ioctl 명령어를 가지고 있고, read ioctl's (프로세스로부터 커널로 정보를 전송) 혹은 write ioctl's (프로세스로 정보 전달) 와 같은 표현을 사용한다. ioctl 함수는 3개의 인자와 함께 사용하는데 적절한 디바이스 파일의 file descripto(파일 기술자), ioctl 번호 그리고 하나의 인자인데, 이 인자를 캐스팅하여 사용하는 것이 가능하다. (구조체 같은 것은 전달할수 없으나, 구조체 포인터와 같이 일반적인 범위의 캐스팅이 가능하다.)

ioctl 번호는 주 장치 번호, ioctl의 형태, 명령, 그리고 인자의 형태 등을 변환한다. ioctl 번호는 헤더 파일에 정의된 보통 하나의 매크로 호출(_IO, _IOR, _IOW, _IOWR --- 형태에 의존적인)에 의해 만들어진다. 이 헤더 파일은 이용되는 ioctl(이렇게 해야 접근할 수 있는 ioctl 함수를 만들 수 있다)과 커널 모듈(이것의 이해를 가능하게 한다) 모두에 의해 #inlcude되어 포함되어져야 한다. 아래의 예에서, 헤더 파일은 chardev.h와 ioctl.c를 이용하는 프로그램에 포함된다.

자신의 커널 모듈에 ioctl의 이용을 원한다면, 이것은 가장 좋은 공식적인 ioctl 지정 방법이다, 이렇게 함으로서 허가되지 않은 누군가가 우연히 ioctl의 제어를 얻으면, 잘못된 것을 알게 될 것이다. 상세한 정보를 원하면, 커널 소스의 `Documentation/ioctl-number.txt'을 참조하라.

Example 7-1. chardev.c

```
/*
* chardev.c - Create an input/output character device
*/

#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/fs.h>
#include <asm/uaccess.h> /* for get_user and put_user */
#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80
```

```
/*
* Is the device open right now? Used to prevent
* concurent access into the same device
*/
static int Device_Open = 0;

/*
* The message the device will give when asked
*/
static char Message[BUF_LEN];

/*
* How far did the process reading the message get?
* Useful if the message is larger than the size of the
* buffer we get to fill in device_read.
*/
static char *Message_Ptr;

/*
* This is called whenever a process attempts to open the device file
*/
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
        printk(KERN_INFO "device_open(%p)₩n", file);
#endif
        /*
        * We don't want to talk to two processes at the same time
        */
        if (Device_Open)
                return -EBUSY;
        Device_Open++;
        /*
        * Initialize the message
        */
        Message_Ptr = Message;
        try_module_get(THIS_MODULE);
        return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
        printk(KERN_INFO "device_release(%p,%p)₩n", inode, file);
#endif

        /*
        * We're now ready for our next caller
        */
        Device_Open--;
        module_put(THIS_MODULE);
        return SUCCESS;
}

/*
* This function is called whenever a process which has already opened the
* device file attempts to read from it.
*/
static ssize_t device_read(struct file *file, /* see include/linux/fs.h */
```

```
        char __user * buffer, /* buffer to be
                                        * filled with data */
        size_t length, /* length of the buffer */
        loff_t * offset)
{
        /*
        * Number of bytes actually written to the buffer
        */
        int bytes_read = 0;

#ifdef DEBUG
        printk(KERN_INFO "device_read(%p,%p,%d)₩n", file, buffer, length);
#endif

        /*
        * If we're at the end of the message, return 0
        * (which signifies end of file)
        */
        if (*Message_Ptr == 0)
                return 0;

        /*
        * Actually put the data into the buffer
        */
        while (length && *Message_Ptr) {

                /*
                * Because the buffer is in the user data segment,
                * not the kernel data segment, assignment wouldn't
                * work. Instead, we have to use put_user which
                * copies data from the kernel data segment to the
                * user data segment.
                */
                put_user(*(Message_Ptr++), buffer++);
                length--;
                bytes_read++;
}

#ifdef DEBUG
                printk(KERN_INFO "Read %d bytes, %d left₩n", bytes_read, length);
#endif

                /*
                * Read functions are supposed to return the number
                * of bytes actually inserted into the buffer
                */
                return bytes_read;
}

/*
* This function is called when somebody tries to
* write into our device file.
*/
static ssize_t
device_write(struct file *file, const char __user * buffer, size_t length, loff_t * offset)
{
        int i;
#ifdef DEBUG
        printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, length);
#endif

        for (i = 0; i < length && i < BUF_LEN; i++)
                get_user(Message[i], buffer + i);
        Message_Ptr = Message;
        /*
        * Again, return the number of input characters used
        */
        return i;
}
```

```
/*
* This function is called whenever a process tries to do an ioctl on our
* device file. We get two extra parameters (additional to the inode and file
* structures, which all device functions get): the number of the ioctl called
* and the parameter given to the ioctl function.
*
* If the ioctl is write or read/write (meaning output is returned to the
* calling process), the ioctl call returns the output of this function.
*
*/
int device_ioctl(struct inode *inode, /* see include/linux/fs.h */
                 struct file *file, /* ditto */
                 unsigned int ioctl_num, /* number and param for ioctl */
                 unsigned long ioctl_param)
{
        int i;
        char *temp;
        char ch;
        /*
        * Switch according to the ioctl called
        */
        switch (ioctl_num) {
        case IOCTL_SET_MSG:
                /*
                * Receive a pointer to a message (in user space) and set that
                * to be the device's message. Get the parameter given to
                * ioctl by the process.
                */
                temp = (char *)ioctl_param;
                /*
                * Find the length of the message
                */
                get_user(ch, temp);
                for (i = 0; ch && i < BUF_LEN; i++, temp++)
                        get_user(ch, temp);
                device_write(file, (char *)ioctl_param, i, 0);
                break;
        case IOCTL_GET_MSG:
                /*
                * Give the current message to the calling process -
                * the parameter we got is a pointer, fill it.
                */
                i = device_read(file, (char *)ioctl_param, 99, 0);
                /*
                * Put a zero at the end of the buffer, so it will be
                * properly
                */
                put_user('₩0', (char *)ioctl_param + i);
                break;
        case IOCTL_GET_NTH_BYTE:
                /*
                * This ioctl is both input (ioctl_param) and
                * output (the return value of this function)
                */
                return Message[ioctl_param];
                break;
        }
        return SUCCESS;
}
```

```
/* Module Declarations */
/*
* This structure will hold the functions to be called
* when a process does something to the device we
* created. Since a pointer to this structure is kept in
* the devices table, it can't be local to
* init_module. NULL is for unimplemented functions.
*/
struct file_operations Fops = {
        .read = device_read,
        .write = device_write,
        .ioctl = device_ioctl,
        .open = device_open,
        .release = device_release, /* a.k.a. close */
};

/*
* Initialize the module - Register the character device
*/
int init_module()
{
        int ret_val;
        /*
        * Register the character device (atleast try)
        */
        ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

        /*
        * Negative values signify an error
        */
        if (ret_val < 0) {
                printk(KERN_ALERT "%s failed with %d₩n",
                "Sorry, registering the character device ", ret_val);
                return ret_val;
        }
        printk(KERN_INFO "%s The major device number is %d.₩n",
                "Registeration is a success", MAJOR_NUM);
        printk(KERN_INFO "If you want to talk to the device driver,₩n");
        printk(KERN_INFO "you'll have to create a device file. ₩n");
        printk(KERN_INFO "We suggest you use:₩n");
        printk(KERN_INFO "mknod %s c %d 0₩n", DEVICE_FILE_NAME, MAJOR_NUM);
        printk(KERN_INFO "The device file name is important, because₩n");
        printk(KERN_INFO "the ioctl program assumes that's the₩n");
        printk(KERN_INFO "file you'll use.₩n");
        return 0;
}

/*
* Cleanup - unregister the appropriate file from /proc
*/
void cleanup_module()
{
        int ret;
        /*
        * Unregister the device
        */
        ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

        /*
        * If there's an error, report it
        */
        if (ret < 0)
                printk(KERN_ALERT "Error: unregister_chrdev: %d₩n", ret);
}
```

Example 7-2. chardev.h

```
/*
* chardev.h - the header file with the ioctl definitions.
*
* The declarations here have to be in a header file, because
* they need to be known both to the kernel module
* (in chardev.c) and the process calling ioctl (ioctl.c)
*/
#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/*
* The major device number. We can't rely on dynamic
* registration any more, because ioctls need to know it.
*/
#define MAJOR_NUM 100

/*
* Set the message of the device driver
*/
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)

/*
* _IOR means that we're creating an ioctl command
* number for passing information from a user process
* to the kernel module.
*
* The first arguments, MAJOR_NUM, is the major device
* number we're using.
*
* The second argument is the number of the command
* (there could be several with different meanings).
*
* The third argument is the type we want to get from
* the process to the kernel.
*/
/*
* Get the message of the device driver
*/
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/*
* This IOCTL is used for output, to get the message
* of the device driver. However, we still need the
* buffer to place the message in to be input,
* as it is allocated by the process.
*/

/*
* Get the n'th byte of the message
*/
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/*
* The IOCTL is used for both input and output. It
* receives from the user a number, n, and returns
* Message[n].
*/

/*
* The name of the device file
*/
#define DEVICE_FILE_NAME "char_dev"

#endif
```

Example 7-3. ioctl.c

```
/*
* ioctl.c - the process to use ioctl's to control the kernel module
*
* Until now we could have used cat for input and output. But now
* we need to do ioctl's, which require writing our own process.
*/

/*
* device specifics, such as ioctl numbers and the
* major device file.
*/
#include "chardev.h"
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h> /* open */
#include <unistd.h> /* exit */
#include <sys/ioctl.h> /* ioctl */

/*
* Functions for the ioctl calls
*/
ioctl_set_msg(int file_desc, char *message)
{
        int ret_val;
        ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
        if (ret_val < 0) {
                printf("ioctl_set_msg failed:%d₩n", ret_val);
        exit(-1);
        }
}

ioctl_get_msg(int file_desc)
{
        int ret_val;
        char message[100];
        /*
        * Warning - this is dangerous because we don't tell
        * the kernel how far it's allowed to write, so it
        * might overflow the buffer. In a real production
        * program, we would have used two ioctls - one to tell
        * the kernel the buffer length and another to give
        * it the buffer to fill
        */
        ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
        if (ret_val < 0) {
                printf("ioctl_get_msg failed:%d₩n", ret_val);
                exit(-1);
        }
        printf("get_msg message:%s₩n", message);
}
```

```
ioctl_get_nth_byte(int file_desc)
{
        int i;
        char c;
        printf("get_nth_byte message:");
        i = 0;
        do {
                c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);
                if (c < 0) {
                printf("ioctl_get_nth_byte failed at the %d'th byte:₩n",i);
                exit(-1);
                }
        putchar(c);
        } while (c != 0);
        putchar('₩n');
}

/*
* Main - Call the ioctl functions
*/
main()
{
        int file_desc, ret_val;
        char *msg = "Message passed by ioctl₩n";
        file_desc = open(DEVICE_FILE_NAME, 0);
        if (file_desc < 0) {
                printf("Can't open device file: %s₩n", DEVICE_FILE_NAME);
                exit(-1);
        }
        ioctl_get_nth_byte(file_desc);
        ioctl_get_msg(file_desc);
        ioctl_set_msg(file_desc, msg);
        close(file_desc);
}
```

Chapter8. System Calls

8.1 System Calls

무엇보다도, 우리가 해왔던 것은 /proc 파일들과 디바이스 핸들러들을 등록하기 위해 잘 정의된 커널 메커니즘을 사용했다는 것이다. 만약 여러분들이 커널 프로그래머적인 생각으로 디바이스 드라이버를 작성 하는것과 같은 , 무언가 여러분들이 원하는 것들을 한 것이였다면 다행이다. 그러나 여러분들이 뭔가 일반적이지 못한 것을 하고싶다면, 즉 시스템의 행동을 조금 다른방향으로 풀어가야 한다면? 그것은 여러분에게 달렸다.

이러한 부분들이 커널 프로그래머가 위험해 빠질수 있는 방향이다. 예를들어, 아래에 있는 예제를 작성하는 동안, 저자는 open( ) 이라는 시스템 호출함수를 죽였다. 이 의미는 저자는 그 어떠한 파일도 열지 못하고, 어떠한 프로그램도 실행하지 못한다는 것이고 그래서 저자는 컴퓨터 조차 정상적인 방법으로 종료하지 못한다는 것이다. 그래서 강제적으로 시스템 전원을 내렸다. 다행이 어떠한 파일도 손상되지 않았다. 이러한 상황에서 여러분이 어떠한 파일도 손상되는 것을 방지하기 위해 insmod 와 rmmod 명령을 실행하기 전에 sync 명령어를 실행해자.

디바이스 파일과 /proc 파일들은 잊자. 그것들은 단지 시스템의 작은, 조금 세부적인 부분들일 뿐이다. 커널의 통신 메커니즘을 위한 진짜 프로세스이자 모든 프로세스들에 의해 사용되는 것은 바로 시스템 호출이다. (System call) 어떤 프로세스가 커널로부터 서비스를 요청한다면 (파일을 열거나 새로운 프로세스를 올리고 메모리를 요청하는등의) 시스템 호출 메커니즘이 사용된다. 여러분들이 커널의 행동들을 조금 흥미로운 방향으로 바꾸길 원한다면 시스템 호출에 관한 내용을 수정하는 것이 적합하다. 여러분들이 프로그램들이 사용하는 시스템 호출이 뭔가를 확인하고 싶다면 strace < 인자 > 명령어를 실행해라.

일반적으로, 프로세스는 커널에 접근하도록 허락되지 않는다. 커널 메모리에도 접근할수 없고 커널 함수를 호출할수도 없다. CPU는 이러한 기능을 요구하고 강요한다. (이러한 환경을 "보호 모드" 라고 부른다.)

시스템 호출은 이러한 일반적인 "보호 모드"의 예외의 경우들이다. 시스템 호출이 일어나면 프로세스는 CPU의 레지스터에 적절한 값을 채우고 특별한 명령을 호출해 미리 정의된 커널의 특정 위치로 점프하여 명령을 수행한다. (물론, 그 위치는 프로세스들이 쓸수는 없고 단지 내용을 읽을수만 있는 장소이다.) 인텔 CPU의 경우엔 위의 행동은 인터럽트 0X80 으로 수행할수 있다. 하드웨어는 여러분들이 일단 그러한 장소로 점프를 해서 뭔가를 수행한다면 더이항 제한된 유저모드로 작업을 하지 않고 시스템 커널을 작동할것을 알고 있다. 그리고 여러분들은 여러분들이 하고픈 어떠한 것도 할수 있는 권한이 부여된다.

커널 프로세스들이 점프할수 있는 장소는 system _call 이라 불린다. 그 위치에 있는 프로시져는 커널에게 어떠한 프로세스가 요구한 서비스가 뭔지 알려주는 시스템 콜 번호를 확인한다. 그리고 시스템 콜 테이블을 확인하여 (sys_call_table) 호출할 커널 함수의 주소를 확인한다. 그리고 해당 커널 함수를 호출하고 리턴 한 후에 몇가지의 시스템 체크를 수행 한 후 다시 프로세스로 돌아간다. (아니면 다른 프로세스들로 돌아가는데, 이는 프로세스들의 실행타임이 run out 된 경우이다.) 만약 여러분이 이러한 내용의 코드를 읽고 싶다면, arch/$<$architecture$>$/kernel/entry.S 파일을 열어 ENTRY(system_call) 이후 부분을 을 확인하면 된다.
만약 여러분이 일반적으로 수행되는 시스템 콜의 작업들을 수정하고 싶다면 여러분들이 해야 할 일은 여러분들의 고유의 함수를 작성하여 구현하는 것이다. (보통 여러분이 작성한 작은 분량의 코드를 더해서 원래의 함수를 호출하는 방식으로 수행한다.) 그리고 sys_call_table의 포인터를 수정해 여러분들이 작성한 함수를 가리키도록 한다. 왜나면 여러분들이 작성한 함수들은 후에 제거될 것이고 시스템을 불안정한 상태로 두지 않기 위해서인데, 이를 위해 cleanup_modules 명령어를 수행해 원래의 상태로 테이블을 돌리는 작업을 수행하자.

아래에 있는 소스코드는 커널모듈의 한 예이다. 어떤 특정 유저를 감시해 유저가 파일을 열었을 때 printk( ) 함수를 사용해 메시지를 출력하는 내용이다. 이것을 위해 시스템 콜을 재배치해 여러분이 작성한 함수를 통해 파일을 열자. 이 함수는 현제 프로세스의 uid를 확인하고 우리가 감시한 유저의 uid 와 동일한지 검사해 만약 그렇다면 printk( ) 함수를 사용해 유저의 이름과 열린 파일의 이름을 출력한다. 그리고 어떠한 방법으로든 원래의 open( ) 함수를 같은

인자값을 사용하여 실제로 파일을 연다.

init_module 함수는 sys_call_table 의 내용을 적절한 위치로 재배치하고 원래의 변수에 저장되 있는 포인터값을 유지한다. cleanup_module 함수는 위의 변수들을 사용해 원래의 상태로 복구한다. 이러한 접근은 위험한데 왜나면 두 개의 커널 모듈이 같은 시스템 콜을 변경할 가능성이 있기 때문이다. 여러분들이 두 개의 커널 모듈을 가지고 있다고 생각해 보자. (A,B) A 커널 모듈은 A_open을, B 커널은 B_open 이라는 시스템 콜을 사용할 것이다. 이제 A 가 커널로 탑재되었고 시스템콜이 A_open 명령에 의해 재배치 되었다. 다음으로 B 모듈이 커널에 탑재되고 B_open을 통해 동일한 커널의 시스템 콜을 재배치 했다고 하자. B 모듈은 A가 정리해 놓은 상황이 원래의 상황이라고 생각할 것이다.

그리고 두가지 상황을 고려해 볼수 있는데, 첫 번째로 B 모듈이 먼저 제거됐다고 하면 모든 것들이 정상적으로 작동될 것이다. B 모듈은 A가 재배치 해 놓은 상태로, A 모듈은 그것을 다시 원래의 상태로 돌려 놓을 것이기 때문이다. 하지만 A 모듈이 먼저 제거 되고 그 후에 B 모듈이 제거 된 상황을 고려 해 보자. A의 모듈 제거로 인해 최초의 상태로 시스템 콜이 복구될 것인데 이 상태는 B가 알고 있는 시스템의 상황이 아니다. 그리고 B가 제거된다면 자신이 알고있는 기존의 A_open 상태로 복구할 것인데, 더 이상 메모리에 존재 하지 않는 상태다. 그리고 시스템이 충돌해 복구할수 없는 오류가 발생하게 된다. 이 문제를 처음 보더라도 우리는 시스템 콜이 열려있는 함수와 동일한지 체크하고 동일하지 않다면 시스템 상태를 변경하지 않도록 하면 모든 문제가 해결될 것처럼 보이지만 (결국 B는 커널에서 제거될 때 시스템 콜을 변경 하지 않게 된다.) 이런 수정은 문제를 더욱 심각하게 만든다. A 모듈이 제거될 때, 시스템 콜이 B_open 으로 변경되고 더 이상 A_open을 가리키지 않게 되고 그래서 메모리로부터 제거될 때 sys_open을 되돌리지 못하게 된다. 불행하게도 B_open은 더 이상 존재하지 않는 A_open을 호출하려고 할 것이며 B 모듈을 제거하지 않아도 시스템은 충돌을 일으킬 것이다.

위와 같은 부분으로 연관되는 모든 문제는 생산을 위해 실행할수 없는 syscall 강탈을 야기한다는 점에 주목하자. 사용자가 sys_call_table 사용으로 생기는 잠제적인 문제를 제거하기 위해서는 더 이상 추출되게 해서는 안된다. 이 말은, 만약 여러분이 이 예제 그 이상으로 뭔가를 하길 원한다면, 여러분은 sys_call_table의 추출을 위해 여러분의 현제 커널을 patch 해야 한다. 예제 디렉토리에서 README와 patch를 찾을수 있을 것이다. 여러분이 상상하는 바와 같이 이러한 수정은 얼토 당토 안하는 이야기가 될 것이다.

Example 8-1. syscall.c

```
/*
* syscall.c
*
* System call "stealing" sample.
*/
/*
* Copyright (C) 2001 by Peter Jay Salzman
*/
/*
* The necessary header files
*/
/*
* Standard in kernel modules
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module, */
#include <linux/moduleparam.h> /* which will have params */
#include <linux/unistd.h> /* The list of system calls */

/*
* For the current (process) structure, we need
* this to know who the current user is.
*/
#include <linux/sched.h>
#include <asm/uaccess.h>
```

```
/*
* The system call table (a table of functions). We
* just define this as external, and the kernel will
* fill it up for us when we are insmod'ed
*
* sys_call_table is no longer exported in 2.6.x kernels.
* If you really want to try this DANGEROUS module you will
* have to apply the supplied patch against your current kernel
* and recompile it.
*/
extern void *sys_call_table[];

/*
* UID we want to spy on - will be filled from the
* command line
*/
static int uid;
module_param(uid, int, 0644);

/*
* A pointer to the original system call. The reason
* we keep this, rather than call the original function
* (sys_open), is because somebody else might have
* replaced the system call before us. Note that this
* is not 100% safe, because if another module
* replaced sys_open before us, then when we're inserted
* we'll call the function in that module - and it
* might be removed before we are.
*
* Another reason for this is that we can't get sys_open.
* It's a static variable, so it is not exported.
*/
asmlinkage int (*original_call) (const char *, int, int);

/*
* The function we'll replace sys_open (the function
* called when you call the open system call) with. To
* find the exact prototype, with the number and type
* of arguments, we find the original function first
* (it's at fs/open.c).
*
* In theory, this means that we're tied to the
* current version of the kernel. In practice, the
* system calls almost never change (it would wreck havoc
* and require programs to be recompiled, since the system
* calls are the interface between the kernel and the
* processes).
*/
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
        int i = 0;
        char ch;
        /*
        * Check if this is the user we're spying on
        */
        if (uid == current->uid) {
                /*
                * Report the file, if relevant
                */
                printk("Opened file by %d: ", uid);
                do {
                        get_user(ch, filename + i);
                        i++;
                        printk("%c", ch);
                } while (ch != 0);
                printk("₩n");
        }
```

```
        /*
        * Call the original sys_open - otherwise, we lose
        * the ability to open files
        */
        return original_call(filename, flags, mode);
}

/*
* Initialize the module - replace the system call
*/
int init_module()
{
        /*
        * Warning - too late for it now, but maybe for
        * next time...
        */
        printk(KERN_ALERT "I'm dangerous. I hope you did a ");
        printk(KERN_ALERT "sync before you insmod'ed me.₩n");
        printk(KERN_ALERT "My counterpart, cleanup_module(), is even");
        printk(KERN_ALERT "more dangerous. If₩n");
        printk(KERN_ALERT "you value your file system, it will ");
        printk(KERN_ALERT "be ₩"sync; rmmod₩" ₩n");
        printk(KERN_ALERT "when you remove this module.₩n");

        /*
        * Keep a pointer to the original function in
        * original_call, and then replace the system call
        * in the system call table with our_sys_open
        */
        original_call = sys_call_table[__NR_open];
        sys_call_table[__NR_open] = our_sys_open;

        /*
        * To get the address of the function for system
        * call foo, go to sys_call_table[__NR_foo].
        */
        printk(KERN_INFO "Spying on UID:%d₩n", uid);
        return 0;
}

/*
* Cleanup - unregister the appropriate file from /proc
*/
void cleanup_module()
{
        /*
        * Return the system call back to normal
        */
        if (sys_call_table[__NR_open] != our_sys_open) {
                printk(KERN_ALERT "Somebody else also played with the ");
                printk(KERN_ALERT "open system call₩n");
                printk(KERN_ALERT "The system may be left in ");
                printk(KERN_ALERT "an unstable state.₩n");
        }
        sys_call_table[__NR_open] = original_call;
}
```

Chapter9. Blocking Processes

9.1 Blocking Processes

누군가가 여러분이 당장 처리해주지 못하는 부탁을 하였을 때 어떻게 할 것인가? 여러분이 인간이고 여러분은 다른 인간 때문에 귀찮음을 느끼며 살아가고 있고, 여러분이 할 수 있는 말은 "지금 바로 말고, 난 바빠! 저리가!" (이런 매정한!) 라는 말 뿐일 것이다. 만약 여러분이 커널 모듈이고, 여러분은 프로세스들로부터 굉장히 괴롭힘을 당하고 귀찮음을 느끼고 있다고 생각해보자. 여러분은 이러한 상황을 탈출하기 위한 여러 가지 방법들을 가지고 있을 것이다. 그중 하나로, 여러분이 프로세스들이 요청한 작업을 수행할 수 있을 때 까지 해당 프로세스를 제워 두는 것이다.
(깨워두면 계속 " 내꺼부터 처리해줘! 라고 말할것이기 때문에..) 그 후, 프로세스들은 커널에 의해 (마치 수면제를 먹은마냥) 조용해 지게 될 것이고, 언제든지 깨어 날수 있는 상태가 된다. (이런 방법으로 하나의 CPU에서 많은 수의 프로세스가 동시간에 수행될수 있게 된다.)

커널 모듈은 이러한 모습의 한 예다. /proc/sleep 이라 불리는 파일은 하나의 프로세스를 열수 있다. 만약 파일이 이미 열려져 있다면 커널 모듈은 wait_event_inturruptible 함수를 호출한다. 이 함수는 작업의 상태를 TASK_INTERRUPTIBLE 로 바꾸는데 (작업은 커널 데이터 구조를 의미하는데 이것은 프로세스와 system_call에 관한 정보를 가지고 있을수도 있고, 그 뿐만 아니라 어떠한 것도 가지고 있을 수 있다.) 이 말은 작업이 프로세스가 어떠한 방식에서든 깨어나 Wait Queue에 들어가기 전까지 실행되지 않음을 의미한다. (역주 : CPU 스케쥴링에 관한 내용은 운영체제 관련 서적에서 자세히 설명하고 있다. 리눅스의 Wait Queue에 관한 설명이 http://blog.naver.com/sysapi/20011482139 이곳에 잘 나와있으니 관심있는 독자들은 참고하길 바란다.) 그리고 함수는 스케쥴러를 호출해 컨텍스트 스위칭을 수행한다. (역주 : 컨텍스트 스위칭 역시 운영체제와 관련된 내용이고, 이글을 읽는 독자들은 자세히 알고 있으리라 생각되지만, 설명을 덧붙히자면, 현제 프로세스 혹은 쓰레드의 내용을 저장하고 다른 프로세스에게 제어권을 넘겨주는 것을 말한다. 이러한 작업은 사용자가 인지할수 없는 속도로 작동되기 때문에 여러 가지 프로세스들이 사용자가 보기에 동시간대에 수행될수 있게 된다.)

프로세스가 다 수행되면, module_close 함수가 호출된다. 이 함수는 Wait Queue에 있는 모든 프로세스를 깨우고 (큐에서 오직 하나의 프로세스를 깨우는 메커니즘은 없다.) 리턴 된 후에 기다리고 있던 프로세스가 실행 된다. 동시에 스케쥴러는 CPU가 어떤 프로세스를 수행하게 할 것인가를 결정한다. 이런 행동은 module_inturruptible_sleep_on 함수가 수행된 후 바로 시작된다. 그 후에 다른 프로세스에게 파일이 계속 열려 있고 위의 생명주기와 함께 계속 수행될 것이라고 알릴 수 있는 전역 변수를 생성하는 과정을 수행할 수 있다.

그래서 우리는 다른 프로세스들이 파일에 접근하려고 시도하는 동안 tail -f 명령어를 사용해 백그라운드 상태에서 파일이 열려있게 할 것이다. kill %1 명령어에 의해 첫 번째 백그라운드 프로세스가 죽게되면 두 번째 프로세스가 일어나 파일에 접근할수 있게 되고 결국에 종료 될 것이다.

조금 더 흥미롭게 하기 위해 module_close에 파일에 접근할 수 있기를 기다리는 프로세스들을 깨울 수 있는 기능에 대해 독점권을 부여하지 않을 것이다. Ctrl+c(SIGINT)와 같은 신호 또한 프로세스를 깨울 수 있다. 이러한 경우에 -EINTR 명령어를 통해 즉시 복귀하고자 원할 것이다. (Ctrl+c 와 같은 시그널로 인해 가로체기를 당한 상황이라 가정했을 때, 다시 복귀해야 한다고 생각하자.) 이러한 행동은 유저들에게 중요한데, 예를들어 프로세스가 파일을 받기 전에 프로세스를 죽인 상황이라면 말이다.

우리가 기억해야 할 점이 하나 더 있다. 가끔 프로세스는 잠들길 원치 않는 경우가 있다. (사람도 그렇다.) 그리곤 잠들기 싫은 프로세스는 자신이 원하는 것을 즉시 얻길 원하거나 아니면 그러한 요구는 수행될수 없다는 답만 얻을 뿐이다. 이러한 프로세스들은 파일을 열 때 O_NONBLOCK 플레그를 사용한다. 커널은 아래와 같은 예제에서 파일을 여는 것과 같이 에러코드 -EAGAIN이 없었다면 단순히 block 될지도 몰랐던 잠들기 싫어하는 프로세스들의 명령을 -EAGAIN 명령을 통해 응답하도록 되어 있다. cat_noblock은 이번챕터의 소스 디렉토리에 있고 O_NONBLOCK 플레그와 함께 파일을 연다.

```
hostname:~/lkmpg-examples/09-BlockingProcesses# insmod sleep.ko
hostname:~/lkmpg-examples/09-BlockingProcesses# cat_noblock /proc/sleep
Last input:
hostname:~/lkmpg-examples/09-BlockingProcesses# tail -f /proc/sleep &
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
tail: /proc/sleep: file truncated
[1] 6540
hostname:~/lkmpg-examples/09-BlockingProcesses# cat_noblock /proc/sleep
Open would block
hostname:~/lkmpg-examples/09-BlockingProcesses# kill %1
[1]+ Terminated tail -f /proc/sleep
hostname:~/lkmpg-examples/09-BlockingProcesses# cat_noblock /proc/sleep
Last input:
hostname:~/lkmpg-examples/09-BlockingProcesses#
```

Example 9-1. sleep.c

```
/*
* sleep.c - create a /proc file, and if several processes try to open it at
* the same time, put all but one to sleep
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <linux/sched.h> /* For putting processes to sleep and waking them up */
#include <asm/uaccess.h> /* for get_user and put_user */

/*
* The module's file functions
*/

/*
* Here we keep the last message received, to prove that we can process our
* input
*/
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];

static struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sleep"

/*
* Since we use the file operations struct, we can't use the special proc
* output provisions - we have to use a standard read function, which is this
* function
*/
```

```
static ssize_t module_output(struct file *file, /* see include/linux/fs.h */
                             char *buf, /* The buffer to put data to (in the user segment) */
                             size_t len, /* The length of the buffer */
                             loff_t * offset)
{
        static int finished = 0;
        int i;
        char message[MESSAGE_LENGTH + 30];

        /*
        * Return 0 to signify end of file - that we have nothing
        * more to say at this point.
        */
        if (finished) {
                finished = 0;
                return 0;
        }

        /*
        * If you don't understand this by now, you're hopeless as a kernel
        * programmer.
        */
        sprintf(message, "Last input:%s₩n", Message);
        for (i = 0; i < len && message[i]; i++)
                put_user(message[i], buf + i);
        finished = 1;
        return i; /* Return the number of bytes "read" */
}

/*
* This function receives input from the user when the user writes to the /proc
* file.
*/
static ssize_t module_input(struct file *file, /* The file itself */
                            const char *buf, /* The buffer with input */
                            size_t length, /* The buffer's length */
                            loff_t * offset)
{                           /* offset to file - ignore */
        int i;

        /*
        * Put the input into Message, where module_output will later be
        * able to use it
        */
        for (i = 0; i < MESSAGE_LENGTH - 1 && i < length; i++)
                get_user(Message[i], buf + i);

        /*
        * we want a standard, zero terminated string
        */
        Message[i] = '₩0';

        /*
        * We need to return the number of input characters used
        */
        return i;
}

/*
* 1 if the file is currently open by somebody
*/
int Already_Open = 0;

/*
* Queue of processes who want our file
*/
DECLARE_WAIT_QUEUE_HEAD(WaitQ);
/*
```

```
* Called when the /proc file is opened
*/
static int module_open(struct inode *inode, struct file *file)
{
        /*
        * If the file's flags include O_NONBLOCK, it means the process doesn't
        * want to wait for the file. In this case, if the file is already
        * open, we should fail with -EAGAIN, meaning "you'll have to try
        * again", instead of blocking a process which would rather stay awake.
        */
        if ((file->f_flags & O_NONBLOCK) && Already_Open)
        return -EAGAIN;

        /*
        * This is the correct place for try_module_get(THIS_MODULE) because
        * if a process is in the loop, which is within the kernel module,
        * the kernel module must not be removed.
        */
        try_module_get(THIS_MODULE);

        /*
        * If the file is already open, wait until it isn't
        */
        while (Already_Open) {
                int i, is_sig = 0;

                /*
                * This function puts the current process, including any system
                * calls, such as us, to sleep. Execution will be resumed right
                * after the function call, either because somebody called
                * wake_up(&WaitQ) (only module_close does that, when the file
                * is closed) or when a signal, such as Ctrl-C, is sent
                * to the process
                */
                wait_event_interruptible(WaitQ, !Already_Open);

                /*
                * If we woke up because we got a signal we're not blocking,
                * return -EINTR (fail the system call). This allows processes
                * to be killed or stopped.
                */

/*
* Emmanuel Papirakis:
*
* This is a little update to work with 2.2.*. Signals now are contained in
* two words (64 bits) and are stored in a structure that contains an array of
* two unsigned longs. We now have to make 2 checks in our if.
*
* Ori Pomerantz:
*
* Nobody promised me they'll never use more than 64 bits, or that this book
* won't be used for a version of Linux with a word size of 16 bits. This code
* would work in any case.
*/
                for (i = 0; i < _NSIG_WORDS && !is_sig; i++)
                        is_sig =
                        current->pending.signal.sig[i] & ~current->blocked.sig[i];
                if (is_sig) {
                        /*
                        * It's important to put module_put(THIS_MODULE) here,
                        * because for processes where the open is interrupted
                        * there will never be a corresponding close. If we
                        * don't decrement the usage count here, we will be
```

```
                        * don't decrement the usage count here, we will be
                        * left with a positive usage count which we'll have no
                        * way to bring down to zero, giving us an immortal
                        * module, which can only be killed by rebooting
                        * the machine.
                        */
                       module_put(THIS_MODULE);
                       return -EINTR;
               }
}

        /*
        * If we got here, Already_Open must be zero
        */

        /*
        * Open the file
        */
        Already_Open = 1;
        return 0;               /* Allow the access */
}

/*
* Called when the /proc file is closed
*/
int module_close(struct inode *inode, struct file *file)
{
        /*
        * Set Already_Open to zero, so one of the processes in the WaitQ will
        * be able to set Already_Open back to one and to open the file. All
        * the other processes will be called when Already_Open is back to one,
        * so they'll go back to sleep.
        */
        Already_Open = 0;

        /*
        * Wake up all the processes in WaitQ, so if anybody is waiting for the
        * file, they can have it.
        */
        wake_up(&WaitQ);
        module_put(THIS_MODULE);
        return 0;                       /* success */
}

/*
* This function decides whether to allow an operation (return zero) or not
* allow it (return a non-zero which indicates why it is not allowed).
*
* The operation can be one of the following values:
* 0 - Execute (run the "file" - meaningless in our case)
* 2 - Write (input to the kernel module)
* 4 - Read (output from the kernel module)
*
* This is the real function that checks file permissions. The permissions
* returned by ls -l are for reference only, and can be overridden here.
*/
static int module_permission(struct inode *inode, int op, struct nameidata *nd)
{
        /*
        * We allow everybody to read from our module, but only root (uid 0)
        * may write to it
        */
        if (op == 4 || (op == 2 && current->euid == 0))
                return 0;
        /*
        * If it's anything else, access is denied
        */
        return -EACCES;
}
```

```
/*
* Structures to register as the /proc file, with pointers to all the relevant
* functions.
*/

/*
* File operations for our proc file. This is where we place pointers to all
* the functions called when somebody tries to do something to our file. NULL
* means we don't want to deal with something.
*/
static struct file_operations File_Ops_4_Our_Proc_File = {
                        .read = module_output, /* "read" from the file */
                        .write = module_input, /* "write" to the file */
                        .open = module_open, /* called when the /proc file is opened */
                        .release = module_close, /* called when it's closed */
};

/*
* Inode operations for our proc file. We need it so we'll have somewhere to
* specify the file operations structure we want to use, and the function we
* use for permissions. It's also possible to specify functions to be called
* for anything else which could be done to an inode (although we don't bother,
* we just put NULL).
*/
static struct inode_operations Inode_Ops_4_Our_Proc_File = {
                                    .permission = module_permission, /* check for permissions */
};

/*
* Module initialization and cleanup
*/

/*
* Initialize the module - register the proc file
*/
int init_module()
{
        Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
        if (Our_Proc_File == NULL) {
                remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
                printk(KERN_ALERT "Error: Could not initialize /proc/test₩n");
                return -ENOMEM;
        }

        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
        Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
        Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 80;
        printk(KERN_INFO "/proc/test created₩n");
        return 0;
}

/*
* Cleanup - unregister our file from /proc. This could get dangerous if
* there are still processes waiting in WaitQ, because they are inside our
* open function, which will get unloaded. I'll explain how to avoid removal
* of a kernel module in such a case in chapter 10.
*/
void cleanup_module()
{
        remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
        printk(KERN_INFO "/proc/test removed₩n");
}
```

Example 9-2. cat_noblock.c

```
/* cat_noblock.c - open a file and display its contents, but exit rather than wait for input */
/* Copyright (C) 1998 by Ori Pomerantz */

#include <stdio.h> /* standard I/O */
#include <fcntl.h> /* for open */
#include <unistd.h> /* for read */
#include <stdlib.h> /* for exit */
#include <errno.h> /* for errno */

#define MAX_BYTES 1024*4

main(int argc, char *argv[])
{
        int fd; /* The file descriptor for the file to read */
        size_t bytes; /* The number of bytes read */
        char buffer[MAX_BYTES]; /* The buffer for the bytes */

        /* Usage */
        if (argc != 2) {
                printf("Usage: %s <filename>₩n", argv[0]);
                puts("Reads the content of a file, but doesn't wait for input");
                exit(-1);
        }

        /* Open the file for reading in non blocking mode */
        fd = open(argv[1], O_RDONLY | O_NONBLOCK);

        /* If open failed */
        if (errno = EAGAIN)
                puts("Open would block");
        else
                puts("Open failed");
        exit(-1);
}

/* Read the file and output its contents */
do {
        int i;

        /* Read characters from the file */
        bytes = read(fd, buffer, MAX_BYTES);

        /* If there's an error, report it and die */
        if (bytes == -1) {
                if (errno = EAGAIN)
                        puts("Normally I'd block, but you told me not to");
                else
                        puts("Another read error");
        exit(-1);
        }

        /* Print the characters */
        if (bytes > 0) {
                for(i=0; i<bytes; i++)
                        putchar(buffer[i]);
                }

        /* While there are no errors and the file isn't over */
        } while (bytes > 0);
}
```

Chapter10. Replacing Printks

10.1 Replacing printk

섹션 1.2.1.2를 통해서, X와 커널모듈 프로그래밍은 맞지 않다고 말했었다. 커널 모듈을 개발하는 상황이라면 맞는 말이다. 그러나 실제로 사용할 때 여러분은 tty가 무엇이든간에 모듈을 불러오기 위해 메시지를 전송하고 싶을 것이다.

현제 작업의 tty 구조체를 얻기위해 현제 실행되고 있는 작업을 가리키는 current라는 포인터를 사용하면 위와 같은 상황을 만들 수 있다. 우리는 tty 구조체를 살펴서 tty에 스트링을 쓰기위해 사용하는 스트링 작성 함수를 가리키는 포인터를 찾아 볼 것이다.

Example 10-1. print_string.c

```
/*
* print_string.c - Send output to the tty we're running on, regardless if it's
* through X11, telnet, etc. We do this by printing the string to the tty
* associated with the current task.
*/
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/sched.h> /* For current */
#include <linux/tty.h> /* For the tty declarations */
#include <linux/version.h> /* For LINUX_VERSION_CODE */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

static void print_string(char *str)
{
        struct tty_struct *my_tty;

        /*
        * tty struct went into signal struct in 2.6.6
        */

        #if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,5) )

                /*
                * The tty for the current task
                */
                my_tty = current->tty;
         #else
                /*
                * The tty for the current task, for 2.6.6+ kernels
                */
                my_tty = current->signal->tty;
        #endif

        /*
        * If my_tty is NULL, the current task has no tty you can print to
        * (ie, if it's a daemon). If so, there's nothing we can do.
        if (my_tty != NULL) {
                /*
                * my_tty->driver is a struct which holds the tty's functions,
                * one of which (write) is used to write strings to the tty.
                * It can be used to take a string either from the user's or kernel's memory segment.
                *
                * The function's 1st parameter is the tty to write to,
                * because the same function would normally be used for all
                * tty's of a certain type. The 2nd parameter controls
                * whether the function receives a string from kernel
                * memory (false, 0) or from user memory (true, non zero).
                * BTW: this param has been removed in Kernels > 2.6.9
                * The (2nd) 3rd parameter is a pointer to a string.
                * The (3rd) 4th parameter is the length of the string.
                *
                * As you will see below, sometimes it's necessary to use
                * preprocessor stuff to create code that works for different
                * kernel versions. The (naive) approach we've taken here
                * does not scale well. The right way to deal with this is described in section 2 of
                * linux/Documentation/SubmittingPatches
                */
                ((my_tty->driver)->write) (my_tty, /* The tty itself */
```

```
#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,9) )
                                                0, /* Don't take the string from user space */
#endif
                                                str, /* String */
                                                strlen(str)); /* Length */
                /*
                * ttys were originally hardware devices, which (usually)
                * strictly followed the ASCII standard. In ASCII, to move to
                * a new line you need two characters, a carriage return and a
                * line feed. On Unix, the ASCII line feed is used for both
                * purposes - so we can't just use ₩n, because it wouldn't have
                * a carriage return and the next line will start at the
                * column right after the line feed.
                *
                * This is why text files are different between Unix and
                * MS Windows. In CP/M and derivatives, like MS-DOS and
                * MS Windows, the ASCII standard was strictly adhered to,
                * and therefore a newline requirs both a LF and a CR.
                */
#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,9) )
                ((my_tty->driver)->write) (my_tty, 0, "₩015₩012", 2);
#else
                ((my_tty->driver)->write) (my_tty, "₩015₩012", 2);
#endif
        }
}

static int __init print_string_init(void)
{
        print_string("The module has been inserted. Hello world!");
        return 0;
}

static void __exit print_string_exit(void)
{
        print_string("The module has been removed. Farewell world!");
}

module_init(print_string_init);
module_exit(print_string_exit);
```

10.2 Flashing keyboard LEDs

보통 일반적인 경우라면, 여러분은 좀 더 간단하고 직접적인 방법으로 외부세상과 통신하길 원할 것이다. 키보드 LED에 불빛비추기 는 이에 대한 정답이다. 즉각적인 방법으로 관심을 끌거나 혹은 현제의 상태를 표시하기 때문이다. 키보드의 LED는 모든 하드웨어에 존재한고 항상 볼 수 있으며 특별히 설정이 필요하지 않고 tty 혹은 파일에 작성하는 것과 비교해서 사용이 훨씬 간편하고 거슬리지 않는다.

다음 예제는 간단한 커널 모듈을 구현한 것인데 커널에 탑제되면 종료시까지 키보드의 LED를 깜빡인다.

Example 10-2. kbleds.c

```
/*
* kbleds.c - Blink keyboard leds until the module is unloaded.
*/
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>
#include <linux/tty.h> /* For fg_console, MAX_NR_CONSOLES */
#include <linux/kd.h> /* For KDSETLED */
#include <linux/vt.h>
#include <linux/console_struct.h> /* For vc_cons */
```

```
MODULE_DESCRIPTION("Example module illustrating the use of Keyboard LEDs.");
MODULE_AUTHOR("Daniele Paolo Scarpazza");
MODULE_LICENSE("GPL");

struct timer_list my_timer;
struct tty_driver *my_driver;
char kbledstatus = 0;

#define BLINK_DELAY HZ/5
#define ALL_LEDS_ON 0x07
#define RESTORE_LEDS 0xFF

/*
* Function my_timer_func blinks the keyboard LEDs periodically by invoking
* command KDSETLED of ioctl() on the keyboard driver. To learn more on virtual
* terminal ioctl operations, please see file:
* /usr/src/linux/drivers/char/vt_ioctl.c, function vt_ioctl().
*
* The argument to KDSETLED is alternatively set to 7 (thus causing the led
* mode to be set to LED_SHOW_IOCTL, and all the leds are lit) and to 0xFF
* (any value above 7 switches back the led mode to LED_SHOW_FLAGS, thus
* the LEDs reflect the actual keyboard status). To learn more on this,
* please see file:
* /usr/src/linux/drivers/char/keyboard.c, function setledstate().
*
*/
static void my_timer_func(unsigned long ptr)
{
        int *pstatus = (int *)ptr;
        if (*pstatus == ALL_LEDS_ON)
                *pstatus = RESTORE_LEDS;
        else
                *pstatus = ALL_LEDS_ON;
        (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED, *pstatus);
        my_timer.expires = jiffies + BLINK_DELAY;
        add_timer(&my_timer);
}

static int __init kbleds_init(void)
{
        int i;
        printk(KERN_INFO "kbleds: loading₩n");
        printk(KERN_INFO "kbleds: fgconsole is %x₩n", fg_console);
        for (i = 0; i < MAX_NR_CONSOLES; i++) {
                if (!vc_cons[i].d)
                        break;
                printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx₩n", i,MAX_NR_CONSOLES,
                        vc_cons[i].d->vc_num, (unsigned long)vc_cons[i].d->vc_tty);
                }
        printk(KERN_INFO "kbleds: finished scanning consoles₩n");
        my_driver = vc_cons[fg_console].d->vc_tty->driver;
        printk(KERN_INFO "kbleds: tty driver magic %x₩n", my_driver->magic);

        /*
        * Set up the LED blink timer the first time
        */
        init_timer(&my_timer);
        my_timer.function = my_timer_func;
        my_timer.data = (unsigned long)&kbledstatus;
        my_timer.expires = jiffies + BLINK_DELAY;
        add_timer(&my_timer);
        return 0;
}

static void __exit kbleds_cleanup(void)
{
        printk(KERN_INFO "kbleds: unloading...₩n");
        del_timer(&my_timer);
        (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED, RESTORE_LEDS);
}
module_init(kbleds_init);
module_exit(kbleds_cleanup);
```

이 장의 챕터중에서 어떤것도 여러분들의 디버깅 욕구를 채우지 못한다면, 시도해볼만한 다른 몇가지의 방법들이 있다. 여러분들의 욕구를 채워주기 위해서 make menuconfig 의 CONFIG_LL_DEBUG 옵션이 좋을거란 생각을 해봤는가?

위의 옵션을 활성화 하게되면 여러분은 low level의 시리얼 포트 접근권한을 얻게 된다. 위의 옵션 변경이 별로 강력해 보지이 않지만, printascii를 사용하기 위해 kernel/printk.c 파일을 patch 하거나 다른 필수적인 syscall을 patch 함으로써 가상적으로 시스템 동작과정을 추적할수 있게 된다.

여러분들이 작성한 모든 코드는 시리얼 라인 위에서 작동한다. 새롭거나 전에는 지원하지 않던 아키텍쳐를 위해 커널포팅을 하는 방법은 보통 가장 우선시 되는 작업들중 하나이다.(역주 : 커널포팅은 임베디드 시스템에 리눅스 커널을 올리기 위해서 주어진 환경에 맞도록 리눅스 커널의 아키텍쳐 부분과 하드웨어 제어를 위한 디바이스 드라이버 소스를 수정하는 것이다.) 넷콘솔에 로그인 하는 방법도 시도해볼만한 가치가 있다.
이 방법을 통해 여러분들의 디버깅을 도울만한 여러 가지 것들을 보게 될 것이고, 그 방법들을 사용하기 전에 알아야 할 몇가지의 것들이 있다.

디버깅은 항상 귀찮고 거슬리는 작업이다. 디버그 코드를 더하는 것은 발생한 버그가 사라지는 것처럼 보이게 한다. 이러한 특징 때문에 여러분들은 더해지는 디버그 코드가 항상 최소가 되도록 노력해야 하고 실제 production code 상에서는 보이지 않도록 해야한다.

Chapter11. Scheduling Tasks

11.1 Scheduling Tasks

매우 빈번하게 우리들은 특정한 시각에 해야만 하거나 혹은 매우 자주해야 하는 일들을 가지고 있다. 이러한 일들이 프로세스에 의해 행해져야 한다면 우리들은 crontab 파일에 추가함으로써 일들일 처리할수 있다. 또한 이러한 일들이 커널 모듈에 의해 행해져야 한다면 우리는 두가지의 방법을 생각해 볼수 있다. 첫 번째는 파일을 여는 것처럼 필요할 때 시스템 콜을 통해서 커널 모듈이 일을 처리할수 있도록 위의 방법과 마찬가지로 crontab 파일을 이용할 수가 있을 것이다. 그러나 이 방법은 굉장히 비효율적인데 crontab에 저장된 새로운 프로세스를 실행할 때, 메모리로부터 새로운 NE를 위해 (역주 : NE는 New Excutable 의 줄임말로, DOS MZ excutable 포맷을 변경한 16-bit .exe 파일 포맷이다.) 메모리를 읽고 메모리 어딘가에 위치한 모든 커널모듈을 깨우기 때문이다.

이 방법 대신 타이머 인터럽트를 이용해 인터럽트가 요청됐을 때 호출되는 함수를 만드는 방법이 더 효율적인 방법이다. 함수 포인터를 가지고 있는 workqueue_struct 구조체에 우리들의 task를 만드는 방법인데, queue_delayed_work를 사용해 큐의 순서대로 타이머 인터럽트 요청시 실행되는 my_workqueue 라고 불리는 큐에 우리들의 task를 집어넣는 것이다. 왜나면 우리들은 함수가 실행되길 원하고, 함수가 호출되든 안되는 다시 my_workqueue에 집어 넣어서 다음 인터럽트 요청시 똑같이 반복해서 실행되길 원하기 때문이다.

한가지 기억해야할 부분이 있는데, rmmod로 모듈이 제거될 때, 첫 번째로 참조 카운터가 체크되는데 그 값이 0이라면 module_cleanup 함수가 호출될 것이다. 그 후 메모리에서 모든 함수가 제거될 것이다. 이러한 과정은 적절하게 진행되어야만 하고 그렇지 않다면 안좋은 일들이 발생하게 될 것이다. 아래의 코드를 보면서 어떻게하면 위의 절차를 안전한 방법으로 진행할 수 있는지 확인해 보자.

Example 11-1. sched.c

```
/*
* sched.c - scheduale a function to be called on every timer interrupt.
*
* Copyright (C) 2001 by Peter Jay Salzman
*/

/*
* The necessary header files
*/

/*
* Standard in kernel modules
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use the proc fs */
#include <linux/workqueue.h> /* We scheduale tasks here */
#include <linux/sched.h> /* We need to put ourselves to sleep and wake up later */
#include <linux/init.h> /* For __init and __exit */
#include <linux/interrupt.h> /* For irqreturn_t */

struct proc_dir_entry *Our_Proc_File;

#define PROC_ENTRY_FILENAME "sched"
#define MY_WORK_QUEUE_NAME "WQsched.c"
```

```
/*
* The number of times the timer interrupt has been called so far
*/
static int TimerIntrpt = 0;
static void intrpt_routine(void *);
static int die = 0; /* set this to 1 for shutdown */

/*
* The work queue structure for this task, from workqueue.h
*/
static struct workqueue_struct *my_workqueue;
static struct work_struct Task;
static DECLARE_WORK(Task, intrpt_routine, NULL);

/*
* This function will be called on every timer interrupt. Notice the void*
* pointer - task functions can be used for more than one purpose, each time
* getting a different parameter.
*/
static void intrpt_routine(void *irrelevant) {
        /*
        * Increment the counter
        */
        TimerIntrpt++;

        /*
        * If cleanup wants us to die
        */
        if (die == 0)
                queue_delayed_work(my_workqueue, &Task, 100);
        }

        /*
        * Put data into the proc fs file.
        */
 ssize_t procfile_read(char *buffer, char **buffer_location, off_t offset, int buffer_length, int *eof, void *data)
{
        int len; /* The number of bytes actually used */

        /*
        * It's static so it will still be in memory
        * when we leave this function
        */
        static char my_buffer[80];

        /*
        * We give all of our information in one go, so if anybody asks us
        * if we have more information the answer should always be no.
        */
        if (offset > 0)
                return 0;

        /*
        * Fill the buffer and get its length
        */
        len = sprintf(my_buffer, "Timer called %d times so far₩n", TimerIntrpt);

        /*
        * Tell the function which called us where the buffer is
        */
        *buffer_location = my_buffer;

        /*
        * Return the length
        */
        return len;
}
```

```
/*
* Initialize the module - register the proc file
*/
int __init init_module() {
        /*
        * Create our /proc file
        */
        Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);

        if (Our_Proc_File == NULL) {
                remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
                printk(KERN_ALERT "Error: Could not initialize /proc/%s₩n", PROC_ENTRY_FILENAME);
                return -ENOMEM;
        }

        Our_Proc_File->read_proc = procfile_read;
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->mode = S_IFREG | S_IRUGO;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 80;

        /*
        * Put the task in the work_timer task queue, so it will be executed at
        * next timer interrupt
        */
        my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
        queue_delayed_work(my_workqueue, &Task, 100);
        printk(KERN_INFO "/proc/%s created₩n", PROC_ENTRY_FILENAME);
        return 0;
}

/*
* Cleanup
*/
void __exit cleanup_module() {
        /*
        * Unregister our /proc file
        */
        remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
        printk(KERN_INFO "/proc/%s removed₩n", PROC_ENTRY_FILENAME);
        die = 1; /* keep intrp_routine from queueing itself */
        cancel_delayed_work(&Task); /* no "new ones" */
        flush_workqueue(my_workqueue); /* wait till all "old ones" finished */
        destroy_workqueue(my_workqueue);

        /*
        * Sleep until intrpt_routine is called one last time. This is
        * necessary, because otherwise we'll deallocate the memory holding
        * intrpt_routine and Task while work_timer still references them.
        * Notice that here we don't allow signals to interrupt us.
        *
        * Since WaitQ is now not NULL, this automatically tells the interrupt
        * routine it's time to die.
        */
}

/*
* some work_queue related functions
* are just available to GPL licensed Modules
*/
MODULE_LICENSE("GPL");
```

Chapter12. Inturrupt Handlers

12.1 Inturrupt Handlers

저번 챕터를 제외하고 커널에서 작업하는 모든 것들은 프로세스에게 요청하거나 아니면 특별한 파일을 다루고 ioctl( ) 함수로 보내거나 (역주 : ioctl 은 Input/Output Control의 약자로 OS의 사용자와 커널을 이어주는 인터페이스의 일부분이다.) , 혹은 시스템 콜을 확인하는 정도였다. 그러나 커널의 일들은 단순한 프로세스들의 요청이 대한 응답이 아니다. 중요한 부분은 machine 과 하드웨어를 이어주는 작업이라고 말할수 있다.

CPU와 다른 하드웨어 사이에는 두가지의 상호작용 방법이 존재한다. 첫 번째는 하드웨어에게 CPU가 명령을 하달하는 방식이고, 다른 하나의 방법은 하드웨어가 CPU에게 뭔가를 요청하는 것이다. 두 번째 방법을 인터럽트 방식이라고 부른다. 인터럽트 방식은 첫 번째 방식보다 구현하기가 어려운데 왜나면 CPU가 아닌 하드웨어의 편의를 위해 다루어져야 하기 때문이다. (역주 : CPU가 일방적으로 명령을 하달하는 것이 하드웨어의 상황을 판단하고 체크하면서 이루어져야 하는 인터럽트의 구현보다 쉽다는 의미이다.) 하드웨어 디바이스들은 매우 작은크기의 RAM을 가지고 있고 그들의 정보를 가능할 때 RAM에서 읽지 않는다면 사라진다는 것이다.

리눅스 아래에서 하드웨어 인터럽트는 IRQ's (InturruptRe quest) 라고 블린다. 두가지의 IRQ's 방식이 있는데 짧거나 긴 IRQ 방식이다. Shor IRQ 는 매우 짧은 시간에 인터럽트들이 다루어 지고 그 순간에 나머지 머신들은 작동 중지 (blocked) 가 되는 방식이다. Long IRQ 는 비교적 긴 시간이 걸리고 그 동안 다른 인터럽트 들이 일어날수도 있다. (그러나 같은 장치에서 또다시 인터럽트가 발생하진 않는다.) 모든 방법이 가능한 상태라면 Long IRQ를 통한 작업이 더 낳다.

CPU가 인터럽트 신호를 인지하면 현제 진행중인 작업을 중단하고 스택에 현제 상태와 파라메터들을 저장 후 인터럽트 핸들러를 호출한다. (현제 진행중인 작업이 요청된 인터럽트의 내용보다 우선순위를 가지고 있다면, 즉 더 중요한 작업이라면 현제 작업이 종료 될 때 까지 인터럽트 요청은 처리되기를 기다려야 한다.) 이 말은 특정한 작업들은 인터럽트 핸들러 자체로는 처리되도록 허용되지 않는다는 말인데, 시스템은 요청된 인터럽트의 상태를 모르기 때문이다. 이 문제의 해결책은 인터럽트 핸들러가 보통 하드웨어로부터 뭔가를 읽거나 보내야 하는 상황과 같이 즉각적으로 처리되야 할 일이 무엇인지 알게 하는 것과 새로운 정보를 조금 뒤 늦게 처리될수 있도록 스케쥴링 하는 방법이다. (위와 같은 처리방법을 bottom half 라 부른다.)

Bottom half 란 interrupt의 결과로 불려지는 커널내의 함수(routine)들의 집합이다. 프로세스의 상태에 의존하지 않으며, sleep()과 같은 함수를 불러서 진행을 블록킹(blocking) 할 수 없다. 참고로 top half란 시스템 콜이나 트랩(trap) 1 의 결과로 생기며, 동기적(synchronous)으로 호출되는 커널내의 함수(routine)들이다. 프로세스와 상태에 의존적이며, sleep()함수를 부름으로써 블록킹 할 수 있다.

인터럽트의 발생시 이를 처리하는 모든 함수들이 불려질 필요는 없다. 즉, 바쁘고 중요한 일을 처리한 다음 나중에 덜 바쁜 일을 처리해 주도록 만들어줄 수 있다. 이와 같은 대표적인 경우로 네트워크(network)에서 발생하는 패킷(packet) 의 처리를 나중으로 미루어 둘 수 도 있을 것이다. 되도록 이면 많은 패킷을 놓치지 않고 빨리 받아서 큐에 넣어둔 다음 네트워크 인터페이스 인터럽트를 처리했다고 알리고, 나중에 이 패킷들에 대한 처리를 다시 조금 한가한 시간에 해주게 된다. 이럴 때 사용할 수 있는 것이 바로 bottom half이다. 따라서, 상대적으로 처리시간이 긴 것들은 이것을 이용해서 나중에 시스템에서 처리해 준다.

커널 버전(version) 2.4에서는bottom half에 대한 처리가 많이 변경되었다. 즉, 소프트웨어 IRQ의 일환으로 처리된다. Bottom half의 초기화는 ~/kernel/softirq.c의 softirq_init()에서 bh_base 배열(array)에 init_bh()함수가 bottom half 핸들러 함수를 등록시켜주는 곳에서 일어나며, bh_acton()함수에서 bottom half에 대한 처리를 해준다. 제거는 remove_bh() 함수가 처리한다. 모든 시스템 콜이 복귀하기 전에 softirq와 mask를 가지고 softirq가 활성 (active)인지를 확인하게 되며, 만약 그럴 경우에는 do_softirq() 함수를 불러서 softirq를 처리한다.

커널은 먼저 처리되야 할 일들이 끝나는 대로 bottom half를 호출한다. 그리고 이것들이 처리되면 커널 모듈에서 허용된 모든 일들이 처리 가능해질 것이다.

이러한 방식을 구현하기 위해서는 request_irq( ) 함수를 호출해야 하는데 이 함수를 통해 relevant IRQ가 요청되었을 때 인터럽트 핸들러를 호출할수 있다. 이함수는 IRQ 번호, 함수이름, 플래그, /proc/interrupts 파일을 위한 이름 그리고 인터럽트 핸들러로 넘기려는 인자값을 받는다. 보통 IRQs를 위해 특정한 번호들이 존재한다. 얼마나 많은

존재하는 많은 IRQs 들은 전부 하드웨어에 의존한다. 플레그들은 SA_SHIRQ를 포함할수 있는데 이는 여러분이 IRQ를 다른 인터럽트 핸들러와 공유할 뜻이 있다는 것을 말한다. 그리고 SA_INERRUPT 는 이 인터럽트가 빠른 인터럽트 라는 것을 의미한다. 이 함수는 IRQ에 다른 핸들러가 없거나 두 핸들러를 공유할 때 성공적으로 실행된다.

인터럽트 핸들러 내에서 우리는 queue_work ( ), mark_bh(BH_IMMEDIATE)를 통해서 bottom half 들을 스케쥴 할수 있다.

12.1.2 Keyboards on the Intel Architecture

이제 남은 모든 챕터들은 Intel 아키텍쳐 중심이다. 만약 여러분들이 Intel 플렛폼에서 작업하지 않는다면 아래의 예제들은 작동하지 않을 것이다.

이번 챕터의 예제를 작성하면서 몇가지 문제가 있었는데, 사실 모든사람들의 컴퓨터에서 작동되는 의미있는 결과를 포함하는 예제를 작성하는 것이 유용하다고 생각하나, 다른 한편으로는 커널은 모든 공통적인 디바이스들을 위해 모든 디바이스 드라이버들을 포함하고 있고, 내가 작성하고자 하는 것들과 공존할수 없기 때문이다. 이 문제를 해결하기 위해 내가 찾은 해결책은 키보드 인터럽트에 관련된 내용을 작성하는 것이고 일반적인 키보드 인터럽트 핸들러를 제일 먼저 disable 시키는 것이였다. drivers/char/keyboard.c처럼 키보드 핸들러가 커널 소스파일에 static symbol로 정의된 이후로 이것들을 복구할 길은 없어졌다. 여러분이 여러분의 파일시스템을 소중하게 생각한다면, 코드를 insmod하기 전에 다른 터미널을 통해 sleep 120; reboot 명령어를 사용하자.

아래의 예제는 IRQ 1에 bind 되는데 이 것은 Intel 아키텍쳐 기반으로 작동되는 키보드 컨트롤러를 위한 IRQ 이다. 그리고 키보드 인터럽트를 받게 되면 키보드의 상태를 읽고 키보드로부터 넘겨지는 값을 읽는다. 그리고 커널이 이 요청들을 실행할수 있다고 판단하는 순간 got_char을 실행해 사용된 키의 코드를 받는다.

Example 12-1. intrpt.c

```
/*
* intrpt.c - An interrupt handler.
*
* Copyright (C) 2001 by Peter Jay Salzman
*/

/*
* The necessary header files
*/

/*
* Standard in kernel modules
*/
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/sched.h>
#include <linux/workqueue.h>
#include <linux/interrupt.h> /* We want an interrupt */
#include <asm/io.h>

#define MY_WORK_QUEUE_NAME "WQsched.c"

static struct workqueue_struct *my_workqueue;

/*
* This will get called by the kernel as soon as it's safe
* to do everything normally allowed by kernel modules.
*/
static void got_char(void *scancode) {
        printk(KERN_INFO "Scan Code %x %s.₩n", (int)*((char *)scancode) & 0x7F,
                                        *((char *)scancode) & 0x80 ? "Released" : "Pressed");
}
```

```
/*
* This function services keyboard interrupts. It reads the relevant
* information from the keyboard and then puts the non time critical
* part into the work queue. This will be run when the kernel considers it safe.
*/
irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs) {
        /*
        * This variables are static because they need to be
        * accessible (through pointers) to the bottom half routine.
        */
        static int initialised = 0;
        static unsigned char scancode;
        static struct work_struct task;
        unsigned char status;

        /*
        * Read keyboard status
        */
        status = inb(0x64);
        scancode = inb(0x60);
        if (initialised == 0) {
                INIT_WORK(&task, got_char, &scancode);
                initialised = 1;
        } else {
                PREPARE_WORK(&task, got_char, &scancode);
        }
        queue_work(my_workqueue, &task);
        return IRQ_HANDLED;
}

/*
* Initialize the module - register the IRQ handler
*/
int init_module() {
        my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);

        /*
        * Since the keyboard handler won't co-exist with another handler,
        * such as us, we have to disable it (free its IRQ) before we do
        * anything. Since we don't know where it is, there's no way to
        * reinstate it later - so the computer will have to be rebooted
        * when we're done.
        */
        free_irq(1, NULL);

        /*
        * Request IRQ 1, the keyboard IRQ, to go to our irq_handler.
        * SA_SHIRQ means we're willing to have othe handlers on this IRQ.
        * SA_INTERRUPT can be used to make the handler into a fast interrupt.
        */
        return request_irq(1, /* The number of the keyboard IRQ on PCs */ irq_handler, /* our handler */
        SA_SHIRQ, "test_keyboard_irq_handler", (void *)(irq_handler));
}

/*
* Cleanup
*/
void cleanup_module() {
        /*
        * This is only here for completeness. It's totally irrelevant, since
        * we don't have a way to restore the normal keyboard interrupt so the
        * computer is completely useless and has to be rebooted.
        */
        free_irq(1, NULL);
}

/*
* some work_queue related functions are just available to GPL licensed Modules
*/
MODULE_LICENSE("GPL");
```

Chapter13. Symmetric Multo Processing

13.1 Symeetrical Multi-Processing

하드웨어의 성능을 향상시키기 위해 할 수 있는 가장 쉽고 싼 방법은 더 많은 CPU를 보드 위에 올리는 것이다. 이를 통해 각각의 CPU가 서로 다른 일들을 하게 하거나 (asymmetrical multi-proccesing, AMO) 아니면 병렬로 처리되게 만들어 같은 작업을 수행하도록 하는 것이다. (symmetrical multi-processing, SMP) AMP로 작업을 수행하도록 구현하는 것은 리눅스와 같은 범용 운영체제에서는 불가능한 방식으로 컴퓨터가 해야만 하는 작업들에 대한 특별한 지식을 요구한다. 반면에 SMP 방식은 AMP에 비해 상대적으로 구현하기가 쉽다.

상대적으로 쉽다는말이 진짜 쉽다는 말을 의미하지는 않는다.... SMP 환경에서 CPU는 같은 메모리를 공유하고 그 결과로 하나의 CPU에서 작동되는 코그의 결과가 다른 CPU가 사용하는 메모리에 영향을 끼친다. 여러분은 더 이상 이전에 여러분들이 세팅한 값이 현제도 유지가 되고 있는지에 대해 확신할수 없게 된다. 다른 CPU가 여러분이 관측하지 못한 순간에 동일 영역에서 작업할수 있기 때문이다. 명백히 이러한 방식으로 프로그램을 작성하는 것은 불가능하다.(안된다는 말)

프로세스 프로그래밍의 경우에 이러한 내용은 주목할만한 대상이 아닌데, 왜나면 프로세스는 일반적으로 하나의 CPU 위에서 동작되기 때문이다. 반면에 커널은 각각 다른 CPU에서 다른 프로세스들을 호출할수 있다.

커널 2.0.x에서 위 내용은 문제되지 않았는데 전체의 커널은 하나의 큰 spin lock에 있기 때문이다. 이 말은 하나의 CPU에 커널이 있는데 다른 CPU가 그 영역에 접근하기 원한다면, 예를들어 시스템 콜 때문에 이런 상황이 발생했다면, 다른 커널은 원래 커널이 위치하고 있던 CPU의 작업이 끝날 때 까지 기다려야 했다. 이런 과정과 절차가 리눅스의 SMP를 안전하게 만들었지만, 효율적이진 못했다.

lock_kernel()

lock_kernel()은 각 아키텍쳐 마다 하나씩 따로 정의되어 있다. 보통 $(TOPDIR)/include/asm-*/smplock.h 에 함수로 정의되어 있고 spark64, sh, cris 아키텍쳐는 매크로로 정의되어 있다.

Lock이 왜 필요하지?

리눅스는 CPU를 여러 개 가진 시스템에서도 실행되고 이를 지원하고 있다. 만약 여러 개의 CPU가 동시에 같은 변수의 값을 조정하고 읽는 경우가 생긴다면 어떻게 되겠는가?
특정 경우엔 값을이 중첩되어서 심각한 오류에 빠지게 될 것이다.
이런 일을 막기 위해 보통 락(lock)이란 것을 쓰는데 쉽게 말해 서로 중첩되서 실행되지 않도록 보장 하는 것이라고 이해하면 된다.
시스템을 초기화 하는 동안에 이런 일이 발생하면 시스템 초기화는 엉망이 되고 커널은 제대로 부팅할 수 없게 된다. 그래서 초기화 동안 락을 걸고 나중에 초기화가 다 끝나면 락을 풀어주게 된다.

Lock - 기초적 설명

멀티태스킹 OS에서 한 변수를 여러 개의 프로세스가 공유하고 이를 동시에 사용한다면 여러 프로세스 간의 연계된 시간에 따라 이 변수를 사용하는 것이 중첩되는데 이를 보통 레이스 컨디션(race condition) 이라고 부른다. 그리고 동시 발생 문제를 다루는 코드를 크리티컬 리전(critical region)이라 부른다. 리눅스는 SMP 상에서 동작하므로 이런 문제가 커널 디자인에 있어서 중요한 문제중 하나다.
위에서 말한 동시 접근과 같은 문제의 해결은 락(lock)을 이용하는 것이고 락은 한번에 하나의 접근만이 크리티컬 리전에 들어가도록 해서 해결한다.

락엔 두가지 타입이 있다. 하나는 스핀락(spinlock)으로 include/asm/spinlock.h에 정의되어 있다. 이 타입은 싱글홀더락(single-holder lock)이고 매우 작고 빠르고 아무데서나 사용할 수 있다.

두번째 타입은 세마포어로 include/asm/semaphore.h에 정의되어 있다. 세마포어는 보통 싱글홀더락(mutex)으로 사용되지만 한번에 여러 홀더를 가질 수 있다. 사용자가 세마포어를 얻지 못하면 사용자의 프로세스는 큐에 넣어지고 세마포어가 사용 가능해질 때 깨어나게된다. 이 말은 프로세스가 기다리는 동안 CPU가 다른 뭔가를 한다는 것이다. 그러나 많은 경우에 그냥 기다릴 수 없을 때가 있다. 이 경우엔 스핀락을 대신 사용해야한다.

커널을 설정할 때 SMP 지원을 체크하지 않았다면 스핀락은 존재하지 않게 된다. 이런 결정은 아무도 동시에 실행하지 않고 락을 걸 이유가 없는 경우 아주 좋은 커널 디자인이라 말할 수 있다. 세마포어는 언제나 존재하는데 이는 사용자 프로세스간에 동기를 맞추기위해 필요하기 때문이다.

커널 2.2.x 에서는 여러개의 CPU가 동일 커널에 동시에 있을수 있다. 이 내용은 모듈을 작성하는 사람이라면 알아두어야 할 부분이다.

Chapter14. Common Pitfalls

14.1 Common Pitfalls

여러분이 커널 모듈을 실제로 작성하기 전에, 여러분이 알아야할 몇가지의 것들이 있다.

표준 라이브러리 사용

여러분들은 표준 라이브러리를 사용할수 없다. 커널 모듈엔 오직 커널 함수들만 존재하고 이 함수들은 /proc/kallsyms에서 확인할수 있다.

인터럽트 헤제

아주 짧은 순간을 위해라면 가능하지만, 해제 후 다시 복귀 시켜놓지 않는다면 시스템이 먹통이 될 것이고 재부팅을 해야만 할 것이다.