LINUX hacker들을 위한

UNIX KERNEL 완전분석으로 가는 길

리눅스를 사랑하는 모든 이들과 함께...

머 릿 말

리눅스는 자유롭게 복사 및 배포가 가능한 Unix계열의 운영체제이다. 값비싼 상용 Unix는 전산학과 재학중인 학생이 아니면 접하기가 어려운 까닭에 자신의 PC에서 Unix를 사용하기를 원하는 많은 가난한 학생들에게 리눅스는 지식의 욕구를 충족시켜줄수 있는 더없이 좋은 선물이 되었다. 게다가 리눅스는 모든 source가 공개되어 있어 운영체제를 제작하기를 원하거나 Unix의 내부를 들여다보길 원하는 이에게는 군침도는 hacking대상이 되는것이다.
리눅스는 현재 많은 연구소, 기관, 회사에 설치, 운용되고 있으며 그 성능이 입증되었다. 그리고 Internet에 물려있는 다수의 host들이 리눅스 시스템으로 이루어져 있다.
본서의 구성은 대부분의 Unix운영체제서적과 구성이 비슷하다. 큰 특징으로는 대부분의 Unix운영체제서적에서 나타나는 개념적인 설명보다는 리눅스 커널 code를 한줄한줄 분석해 가는 것으로 구성하였다. 따라서 실전이 부족한 타 서적의 내용을 이해하는데 도움을 주리라 믿는다.

본서의 목적은 많은 시스템 프로그래머들이 운영체제는 누구처럼 타고난 재능이 있는 사람들이나 만들수 있는 것이라는 막연한 두려움에서 벗어날수 있도록 자신감을 불어넣어 줌으로써 미래에 나올 새로운 운영체제를 만드는데 보다 많은 이들이 동참할수 있도록 하는 것이다.

본서는 컴퓨터초보자가 보기에는 난해한 내용이다. 따라서 이 책을 보는 이들에게는 다음에 열거한 것들이 반드시 선행되어야 할것이다.

- 기본적인 컴퓨터용어에 대한 이해를 하고 있을것.
- C 언어의 기본문법을 이해하고 있을것.
- assembly 언어의 기본문법을 이해하고 있을것.

이 외에도 본인은 다음과 같은 것을 권한다.

- DOS상에서라도 system programing을 해본적이 있을것.
- Unix의 user수준에서의 이해와 Unix programing에 대한 기본적인 이해가 있을것.
  본인은 이것을 갖추지 못한채 시작하여 많은 시행착오를 거쳤다.
- 32bit 보호모드 관련서적을 한번쯤 읽어보자.
  아마도 본서를 읽는 과정에서 가장 큰 부담은 보호모드의 이해일것이다. 하지만 본서에서 보호모드의 실전을 충분히 경험할수 있을것이다.

여느 책들과는 다르게 참고서적(bibliagraphy)을 한 곳에 모아두지 않고, 관련된 내용이 나올때마다 주석으로 달아두었다. 운영체제를 분석하기 위해서는 hardware및 software에 대한 많은 지식과 자료가 요구된다. 당연한 소리지만, 그것을 이 한권에서 모두 설명할수는 없다.
그리고 본서에서는 영어로 된 컴퓨터용어를 발음대로 우리글로 적거나, 무리하게 우리말로 바꾸려고 애쓰지 않고 영어를 그대로 사용한 경우가 많다. 이것은 본인이 컴퓨터관련 서적을 보면서 자주 느꼈던 혼돈과 어색함때문에 어렵게 결정된 것이니만큼, 본인의 애국심을 의심하지는 말기 바란다.
커널 source중에 다음과 같이 컴파일이 되지않는 곳은 책내용의 간결성을 이유로 분석대상에서 제외시켰다.

```
#ifdef 0
  .....
#endif
```

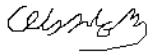
13장에는 kernel전체에서 공통적으로 자주 사용되는 주요함수들을 C library reference형태로 구성해 두었다. 주로 scheduling과 메모리, 그리고 file system에 관한 부분들이다.

본인의 짧은 지식으로 인해 부족한 점이 많을 것으로 안다. 책의 내용중에 문제가 발견되면 언제든지 지적해주기 바란다. 리눅스 커널의 분석은 궁극적으로 본인을 포함한 컴퓨터

와 리눅스를 사랑하는 이들이 함께 이룩해 나가는 것이다.

본인이 처음 리눅스를 설치하면서 당혹스러워 할때, 안면이 전혀 없는 본인의 집을 찾아와 리눅스를 설치해 준 이인아씨에게 가장 먼저 감사드리고 싶다. 이인아씨는 본인에게 PC통신의 막강한 힘과 정보화 사회가 무엇인가를 처음 느끼게 해 준 은인이기도 하다. 이 분외에도 PC통신을 통한 본인의 질문에 답해주신 많은 분들과 타인을 위해 자료실에 자료를 올리는데 지금도 자신의 소중한 시간과 경비를 들이고 있을 이들에게 감사드린다. 이들이 바로 우리나라 정보화사회를 이끌어가는 주역들이라고 믿는다.

1995. 9.



나우우리 ID : techead

living now in Daegu, **Korea**

목 차

저작권에 대한 글

1장

리눅스에 대한 소개

1.1 리눅스란..

리눅스는 **무료로** 사용및 배포가 가능한 UNIX계열(clone)의 운영체제이다. 이것은 현재 상용으로 판매되고 있는 system V 라던지 SCO UNIX와 비교된다. 리눅스외에도 상용이 아닌 운영체제로는 역시 UNIX계열인 386BSD와 MINIX가 있다. MINIX[1]는 UNIX계열로서는 규모가 작은 운영체제로서 리눅스의 모체가 되었다. 그러나 리눅스는 어떠한 다른 운영체제의 코드도 사용하지 않았으며, 완전히 독자적인 코드에 의해 만들어졌다.

리눅스는 UNIX의 모양을 갖추고 있지만 실제로 UNIX는 아니다. 왜냐하면 UNIX는 AT&T사에서 분리되어 나온 USL(Unix System Laborities)의 trademark이기 때문이다. 우리는 리눅스 관련 문서들에서 서두에 다음과 같이 명시되어 있음을 볼수있다.

Unix is a trademark of Unix System Laborities. (*or* X/Open)
Linux is not a trademark, and has no connection to UNIX or to Unix System Laborities.

리눅스 kernel에서는 상업성을 배제하고 있기때문에 대부분의 상용 UNIX에서 보여주는 코드의

---

1) MINIX 개발자인 ANDREW S.TANENBAUM 가 저술한 "OPERATION SYSTEMS DESIGN AND IMPLEMENTATIO -N" 에 MINIX의 source code가 있다.

의도적인 난해성은 보이지 않으며 보다 간결한 알고리즘으로 이루어진다.

리눅스는 핀란드의 헬싱키대학에 재학중인 *라이너스 토발즈*(Linus Torvaldz)라는 학생에 의해 운영체제의 핵심부분인 kernel이 개발되었다. 이 후 세계의 많은 전문 프로그래머(흔히 hacker라고 부른다.)들에 의해 인터넷(Internet)을 통한 kernel개발 및 어플리케이션 개발에 대한 지원이 있어 현재 리눅스는 완전한 운영체제로서 자리매김을 할수있게 되었다. 지금 리눅스는 스스로를 "마지막 해커"라고 부르는 *스톨만*(Richard M. Stallman)이라는 사람에 의해 설립된 FSF(Free Software Foundation)의 GNU project로서 진행되고 있다. 배포되는 리눅스내의 모든 소프트웨어는 programer가 저작권을 가지고 있고, 배포 및 코드의 수정은 GPL(GNU Genral Public Lincense)[2]의 제약을 받는다. FSF는 이름에서 알수 있듯이 **모든 소프트웨어는 자유롭게 공유되어야 한다**는 취지아래 활동하고 있는 비영리단체이다.

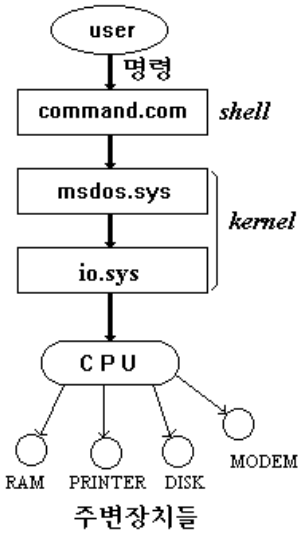
리눅스는 **보호모드**(protected mode)에서 수행되기 때문에 **실 모드**(real mode)에서 수행되는 DOS와는 달리 *multi-tasking*[3], *multi-user*를 지원한다. 리눅스 system의 성능은 많은 사람들에 의해 다양한 방법에 의해 확인되었듯이 486DX2-50에 설치될 경우 workstation급의 강력한 기능을 수행할수 있다.

1.2 DOS kernel

kernel이란 운영체제에서 가장 핵심이 되는 부분으로서 MS-DOS에서는 io.sys, msdos.sys에 해당된다. 그리고 DOS에서의 command.com은 명령번역기(command interpreter) 또는 *shell*이라 불리우는데, DOS의 경우 그림<1.1>과 같은 중간과정을 거쳐 user로부터 주변장치등의 hardware에게 명령이 전달된다.
공장에서 주임격인 msdos.sys는 공장장인 command.com으로부터 받은 명령을 분석하여 io.sys내의 기능공들에게 지시를 한다.(user는 당연히 사장!) 기능공들은 각자에게 할당된 역활에 따라 순서대로 작업이 이루어진다. 이들 기능공들은 DOS interrupt 기능들[4] 각각에 해당한다.

---

2) GPL은 FSF에서 배포되는 모든 소프트웨어에 적용되며 그 내용은 GNU document들에 수록되어 있다.
3) 일반적으로 multi-processing과 같은 의미로 사용된다.



그림<1.1> DOS에서의 명령전달

1.3 리눅스 kernel

리눅스의 kernel명은 초기에는 Image, vlminuz이었다가 지금은 주로 **zImage**를 사용하고 있다. 앞에붙은 z는 disk를 절약하기 위해 압축되었다는 의미를 가지고 있다. 리눅스 kernel은 어느 directory에든 위치할수 있다. 하지만 보통은 root directory에 위치한다. 본서에서는 kernel 버젼 1.0을 대상으로 설명한다. 버젼 1.0은 매우 안정된 것으로 라이너스에 의해 신중하게 *버젼 번호*가 결정되었다고 한다. 리눅스의 kernel에 대한 보다 일반적인 사항은 kernel.faq등의 GNU document를 참고하기 바란다.

1.4 kernel 설치

리눅스 kernel은 거의 2~3달 간격으로[5] upgrade판이 발표된다. upgrade된 kernel은 Internet등 여러경로를 통해 source의 형태로 구할수 있는데, 구하여진 kernel source는 사용자에 의한 컴파

4) DOS interrupt table이 나와있는 자료를 참고하라.
5) 이것은 순전히 Linus의 의지에 달려있다.

일을 거쳐 기존의 리눅스 system에 심겨진다.(컴파일된 image를 root directory에 복사한다.) kernel을 upgrade시키는 과정은 다음과 같다.

```
# /usr/src/gzip -dc linux1_0.tgz ¦ tar xvf -      /*  kernel source를 푼다. */
# /usr/src/cd linux
# /usr/src/linux/make config                      /*  kernel 환경설정  */
# /usr/src/linux/make dep                         /*  depend file 생성  */
# /usr/src/linux/make zImage                      /*  compile, link  */
# /usr/src/linux/arch/i386/boot/cp zImage /       /*  image를 root directory에 copy  */
```

이 후 LILO를 다시 설치함으로써 kernel의 upgrade가 끝난다.
컴파일이 제대로 되지 않을때는 현재 system에 설치된 C library의 버젼이 낮지 않은지 확인해 보아야 한다.

1.5 Makefile

◎ main Makefile

```
1) VERSION = 1
   PATCHLEVEL = 0
   ALPHA =

2) all:    Version zImage

3) .EXPORT_ALL_VARIABLES:

4) CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
          else if [ -x /bin/bash ]; then echo /bin/bash; \
          else echo sh; fi ; fi)

   #
   # Make "config" the default target if there is no configuration file or
   # "depend" the target if there is no top-level dependency information.
   #
5) ifeq (.config,$(wildcard .config))
   include .config
```

```
ifeq (.depend,$(wildcard .depend))
include .depend
else
CONFIGURATION = depend
endif
else
CONFIGURATION = config
endif

ifdef CONFIGURATION
CONFIGURE = dummy
endif

#
# ROOT_DEV specifies the default root-device when making the image.
# This can be either FLOPPY,CURRENT,/dev/xxxx or empty, in which case
# the default of FLOPPY is used by 'build'.
#
```

```
6) ROOT_DEV = CURRENT
```

```
#
# If you want to preset the SVGA mode, uncomment the next line and
# set SVGA_MODE to whatever number you want.
# Set it to -DSVGA_MODE=NORMAL_VGA if you just want the EGA/VGA mode.
# The number is the same as you would ordinarily press at bootup.
#

SVGA_MODE=      -DSVGA_MODE=NORMAL_VGA

#
# standard CFLAGS
#
```

```
7) CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -pipe
```

```
ifdef CONFIG_CPP
CFLAGS := $(CFLAGS) -x c++
endif
```

```
   ifdef CONFIG_M486
   CFLAGS := $(CFLAGS) -m486
   else
   CFLAGS := $(CFLAGS) -m386
   endif

   #
   # if you want the ram-disk device, define this to be the
   # size in blocks.
   #

   #RAMDISK = -DRAMDISK=512

8) AS86    =as86 -0 -a
   LD86    =ld86 -0

   AS      =as
   LD      =ld
   HOSTCC  =gcc
   CC      =gcc -D__KERNEL__
   MAKE    =make
   CPP     =$(CC) -E
   AR      =ar
   STRIP   =strip

   ARCHIVES        =kernel/kernel.o mm/mm.o fs/fs.o net/net.o ipc/ipc.o
   FILESYSTEMS     =fs/filesystems.a
   DRIVERS         =drivers/block/block.a \
                drivers/char/char.a \
                drivers/net/net.a \
                ibcs/ibcs.o
   LIBS            =lib/lib.a
   SUBDIRS         =kernel drivers mm fs net ipc ibcs lib

   KERNELHDRS      =/usr/src/linux/include

   ifdef CONFIG_SCSI
   DRIVERS := $(DRIVERS) drivers/scsi/scsi.a
```

```
   endif

   ifdef CONFIG_SOUND
   DRIVERS := $(DRIVERS) drivers/sound/sound.a
   endif

   ifdef CONFIG_MATH_EMULATION
   DRIVERS := $(DRIVERS) drivers/FPU-emu/math.a
   endif

9) .c.s:
        $(CC) $(CFLAGS) -S -o $*.s $<
   .s.o:
        $(AS) -c -o $*.o $<
   .c.o:
        $(CC) $(CFLAGS) -c -o $*.o $<

   Version: dummy
        rm -f tools/version.h

10) config:
        $(CONFIG_SHELL) Configure $(OPTS) < config.in
        @if grep -s '^CONFIG_SOUND' .tmpconfig ; then \
                $(MAKE) -C drivers/sound config; \
                else : ; fi
        mv .tmpconfig .config

   linuxsubdirs: dummy
        set -e; for i in $(SUBDIRS); do $(MAKE) -C $$i; done

   tools/./version.h: tools/version.h

   tools/version.h: $(CONFIGURE) Makefile
        @./makever.sh
        @echo \#define UTS_RELEASE \"$(VERSION).$(PATCHLEVEL)$(ALPHA)\" > tools/version.h
        @echo \#define UTS_VERSION \"\#`cat .version` `date`\" >> tools/version.h
        @echo \#define LINUX_COMPILE_TIME \"`date +%T`\" >> tools/version.h
        @echo \#define LINUX_COMPILE_BY \"`whoami`\" >> tools/version.h
        @echo \#define LINUX_COMPILE_HOST \"`hostname`\" >> tools/version.h
```

```
        @echo \#define LINUX_COMPILE_DOMAIN \"`domainname`\" >> tools/version.h

11)tools/build: tools/build.c $(CONFIGURE)
        $(HOSTCC) $(CFLAGS) -o $@ $<

   boot/head.o: $(CONFIGURE) boot/head.s

   boot/head.s: boot/head.S $(CONFIGURE) include/linux/tasks.h
        $(CPP) -traditional $< -o $@

   tools/version.o: tools/version.c tools/version.h

12)init/main.o: $(CONFIGURE) init/main.c
        $(CC) $(CFLAGS) $(PROFILING) -c -o $*.o $<

   tools/system:   boot/head.o init/main.o tools/version.o linuxsubdirs
        $(LD) $(LDFLAGS) -Ttext 1000 boot/head.o init/main.o tools/version.o \
                $(ARCHIVES) \
                $(FILESYSTEMS) \
                $(DRIVERS) \
                $(LIBS) \
                -o tools/system
        nm tools/zSystem | grep -v '\(compiled\)\|\(\.o$$\)\|\( a \)' | \
                sort > System.map

   boot/setup: boot/setup.o
        $(LD86) -s -o $@ $<

   boot/setup.o: boot/setup.s
        $(AS86) -o $@ $<

   boot/setup.s: boot/setup.S $(CONFIGURE) include/linux/config.h Makefile
        $(CPP) -traditional $(SVGA_MODE) $(RAMDISK) $< -o $@

   boot/bootsect: boot/bootsect.o
        $(LD86) -s -o $@ $<

   boot/bootsect.o: boot/bootsect.s
        $(AS86) -o $@ $<
```

```
   boot/bootsect.s: boot/bootsect.S $(CONFIGURE) include/linux/config.h Makefile
        $(CPP) -traditional $(SVGA_MODE) $(RAMDISK) $< -o $@

13) zBoot/zSystem: zBoot/*.c zBoot/*.S tools/zSystem
        $(MAKE) -C zBoot

14) zImage: $(CONFIGURE) boot/bootsect boot/setup zBoot/zSystem tools/build
        tools/build boot/bootsect boot/setup zBoot/zSystem $(ROOT_DEV) > zImage
        sync

   zdisk: zImage
        dd bs=8192 if=zImage of=/dev/fd0

   zlilo: $(CONFIGURE) zImage
        if [ -f /vmlinuz ]; then mv /vmlinuz /vmlinuz.old; fi
        if [ -f /zSystem.map ]; then mv /zSystem.map /zSystem.old; fi
        cat zImage > /vmlinuz
        cp zSystem.map /
        if [ -x /sbin/lilo ]; then /sbin/lilo; else /etc/lilo/install; fi

15) tools/zSystem:  boot/head.o init/main.o tools/version.o linuxsubdirs
        $(LD) $(LDFLAGS) -Ttext 100000 boot/head.o init/main.o
             tools/version.o \
                $(ARCHIVES) \
                $(FILESYSTEMS) \
                $(DRIVERS) \
                $(LIBS) \
                -o tools/zSystem
        nm tools/zSystem | grep -v '\(compiled\)\|\(\.o$$\)\|\( a \)' | \
16)             sort > zSystem.map

   fs: dummy
        $(MAKE) linuxsubdirs SUBDIRS=fs

   lib: dummy
        $(MAKE) linuxsubdirs SUBDIRS=lib

   mm: dummy
```

```
        $(MAKE) linuxsubdirs SUBDIRS=mm

    ipc: dummy
        $(MAKE) linuxsubdirs SUBDIRS=ipc

    kernel: dummy
        $(MAKE) linuxsubdirs SUBDIRS=kernel

    drivers: dummy
        $(MAKE) linuxsubdirs SUBDIRS=drivers

    net: dummy
        $(MAKE) linuxsubdirs SUBDIRS=net

    clean:
        rm -f kernel/ksyms.lst
        rm -f core `find . -name '*.[oas]' -print`
        rm -f core `find . -name 'core' -print`
        rm -f zImage zSystem.map tools/zSystem tools/system
        rm -f Image System.map boot/bootsect boot/setup
        rm -f zBoot/zSystem zBoot/xtract zBoot/piggyback
        rm -f .tmp* drivers/sound/configure
        rm -f init/*.o tools/build boot/*.o tools/*.o

17) mrproper: clean
        rm -f include/linux/autoconf.h tools/version.h
        rm -f drivers/sound/local.h
        rm -f .version .config* config.old
        rm -f .depend `find . -name .depend -print`

    distclean: mrproper

    backup: mrproper
        cd .. && tar cf - linux | gzip -9 > backup.gz
        sync

    depend dep:
        touch tools/version.h
        for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done > .tmpdepend
```

```
        for i in tools/*.c;do echo -n "tools/";$(CPP) -M $$i;done >> .tmpdepend
        set -e; for i in $(SUBDIRS); do $(MAKE) -C $$i dep; done
        rm -f tools/version.h
        mv .tmpdepend .depend

   ifdef CONFIGURATION
   ..$(CONFIGURATION):
        @echo
        @echo "You have a bad or nonexistent" .$(CONFIGURATION) ": running 'make" $(CONFI
             GURATION)
        @echo"'"
        $(MAKE) $(CONFIGURATION)
        @echo
        @echo "Successful. Try re-making (ignore the error that follows)"
        @echo
        exit 1

18)dummy: ..$(CONFIGURATION)

   else

19)dummy:

   endif

   #
   # Leave these dummy entries for now to tell people that they are going away..
   #
   lilo:
        @echo
        @echo Uncompressed kernel images no longer supported. Use
        @echo \"make zlilo\" instead.
        @echo
        @exit 1

   Image:
        @echo
        @echo Uncompressed kernel images no longer supported. Use
        @echo \"make zImage\" instead.
```

```
        @echo
        @exit 1

    disk:
        @echo
        @echo Uncompressed kernel images no longer supported. Use
        @echo \"make zdisk\" instead.
        @echo
        @exit 1
```

1) kernel version
2) make 실행시 최종적으로 생성될 두 파일. Verison, zImage
3) 각 디렉토리의 sub-makefile로 모든 변수값들이 전달된다.
4) CONFIG_SHELL변수에 우리가 사용중인 shell을 넣는다.
   (GNU MAKE MANUAL 에서 shell FUNCTION부분 참조)
5) 커널 처음 컴파일시 현디렉토리에 .depend .config가 없음을 확인하고 make depend, make config를 실행할수 있도록 한다.

   ```
   # make depend
   ```

   는 생성시킬 object file의 target과 dependancy 관계에 대한 정보 file인 **.depend**을 생성하고 다음과 같은 메세지를 출력한다.

   ```
   Successful. Try re-making (ignore the error that follows)
   ```

   다시

   ```
   # make
   ```

   할때는 .depend file이 include된다. 다음은 main directory에 있는 .depend file의 내용이다. .depend file은 sub-Makefile이 있는 각 sub-directory들에도 생긴다. 이렇게 함으로써 9번 line과 같이 암시적 규칙(implicit rule)을 사용할때 사람이 taget과 dependancy 관계를 일일이 작성하지 않아도 되는 것이다.

```
init/main.o : init/main.c /usr/lib/gcc-lib/i486-linux/2.5.8/include/stdarg.h
/usr/include/asm/system.h \
/usr/include/linux/segment.h /usr/include/asm/io.h /usr/include/linux/types.h \
/usr/include/linux/fcntl.h /usr/include/linux/config.h /usr/include/linux/autoconf.h \
/usr/include/linux/sched.h /usr/include/linux/tasks.h /usr/include/linux/head.h \
/usr/include/linux/fs.h /usr/include/linux/linkage.h /usr/include/linux/limits.h \
```

```
/usr/include/linux/wait.h /usr/include/linux/dirent.h /usr/include/linux/vfs.h \
/usr/include/linux/net.h /usr/include/linux/socket.h /usr/include/linux/sockios.h \
/usr/include/linux/pipe_fs_i.h /usr/include/linux/minix_fs_i.h
/usr/include/linux/ext_fs_i.h \
/usr/include/linux/ext2_fs_i.h /usr/include/linux/hpfs_fs_i.h
/usr/include/linux/msdos_fs_i.h \
/usr/include/linux/iso_fs_i.h /usr/include/linux/nfs_fs_i.h /usr/include/linux/nfs.h \
/usr/include/linux/xia_fs_i.h /usr/include/linux/sysv_fs_i.h
/usr/include/linux/minix_fs_sb.h \
/usr/include/linux/ext_fs_sb.h /usr/include/linux/ext2_fs_sb.h
/usr/include/linux/hpfs_fs_sb.h \
/usr/include/linux/msdos_fs_sb.h /usr/include/linux/iso_fs_sb.h
/usr/include/linux/nfs_fs_sb.h \
/usr/include/linux/xia_fs_sb.h /usr/include/linux/sysv_fs_sb.h /usr/include/linux/mm.h \
/usr/include/linux/page.h /usr/include/linux/errno.h /usr/include/linux/kernel.h \
/usr/include/linux/signal.h /usr/include/linux/time.h /usr/include/linux/param.h \
/usr/include/linux/resource.h /usr/include/linux/vm86.h /usr/include/linux/math_emu.h \
/usr/include/linux/tty.h /usr/include/linux/termios.h /usr/include/linux/unistd.h \
/usr/include/linux/string.h /usr/include/linux/timer.h /usr/include/linux/ctype.h \
/usr/include/linux/delay.h /usr/include/linux/utsname.h /usr/include/linux/ioport.h
tools/build.o : tools/build.c /usr/include/stdio.h /usr/include/features.h
/usr/include/sys/cdefs.h \
/usr/include/libio.h /usr/include/_G_config.h /usr/include/string.h
/usr/lib/gcc-lib/i486-linux/2.5.8/include/stddef.h \
/usr/include/stdlib.h /usr/include/errno.h /usr/include/linux/errno.h
/usr/lib/gcc-lib/i486-linux/2.5.8/include/float.h \
/usr/include/alloca.h /usr/include/sys/types.h /usr/include/linux/types.h
/usr/include/sys/stat.h \
/usr/include/linux/stat.h /usr/include/sys/sysmacros.h /usr/include/unistd.h \
/usr/include/posix_opt.h /usr/include/gnu/types.h /usr/include/fcntl.h
/usr/include/linux/fcntl.h \
/usr/include/linux/config.h /usr/include/linux/autoconf.h /usr/include/linux/a.out.h \
/usr/include/linux/page.h
tools/version.o : tools/version.c /usr/include/linux/config.h
/usr/include/linux/autoconf.h \
/usr/include/linux/utsname.h tools/./version.h
```

그런데 11번 line과 12번 line의 경우를 보자. 이미 규칙(rule)이 작성되어 있다. 이럴때는 명시적 규칙이 적용된다. 명시적 규칙이 암시적 규칙보다 우선하기 때문이다.

6) 11번 line의 build실행시 사용.

7) 옵션 -W : warinig 메세지 출력.
-02 : 최적화. 반면에 컴파일 시간이 길어진다.
-fomit-frame-ponter : frame pointer를 요구하지 않는 함수에 대해서 는 레지스터에 frame pointer를 save하지 마라.
-pipe : 컴파일 상태사이의 대화를 위해 임시파일이 아닌 파이프를 사용하라.
(GNU CC MANUAL 참조)

8) as86과 as는 16 bit 어셈블러와 32 bit 어셈블러에 각각 해당한다. bootsect.S와 setup.S는 as86에 의해, head.S는 as에 의해 어셈블링(assembling)이 이루어진다. ld86과 ld도 또한 같은 내용이다.
-E 옵션은 프리프로세서(#include,#define)등을 먼저 처리한후 컴파일을 멈추라는 의미이다.
컴파일은 순서대로 표<1.1>와 같은 과정을 거친다.

컴파일 순서	과 정	작 업	gcc 옵션
1	preprocessing	preprocessor을 처리한후 정지	-E
2	compile proper	C code를 어셈블리 code로 만든 후 정지	-S
3	assembling	object 모듈생성	
4	link	object module들을 묶어 실행 file 생성	

<표1.1> 컴파일 과정

10) make config시 수행.

13) 15번 line의 tools/zSystem을 source로하여 zBoot디렉토리내의 Makefile을 대상으로 컴파일. 이곳에서 압축이 이루어져 compressed kernel이 되도록 한다.

14) kernel image.
build에 의해 컴파일된 bootsect, setup, zSystem이 합쳐진다.

15) 16) 각 sub-directory의 컴파일된 오브젝트들을 link한다. -Ttext 옵션은 코드가 메모리상에 놓이는 위치를 의미한다. 최종적으로 kernel은 0x100000 byte의 위치에 놓이게 된다.
zSystem.map file에는 object file에서의 symbol들이 적재되며 초기필드의 hexa값들에 의해 각 symbol들의 메모리내에서의 위치(주소값)를 알수 있다. 이 file은 커널을 분석하는데 있어 매우 유용하다.

17) make clean은 depend file과 config file은 없애지 않는데 반해, 이것은 완전히 kernel source 만 남도록 한다.

18) 19) dummy와 앞에서 사용된 CONFIGURE는 depend file과 config file의 존재를 확인하기 위해 이용된다.

◎ /zBoot내의 sub-Makefile

```
   HEAD = head.o
   SYSTEM = ../tools/zSystem
   #LD = gcc
   #TEST = -DTEST_DRIVER

1)  zOBJECTS = $(HEAD) inflate.o unzip.o misc.o

   CFLAGS = -O2 -DSTDC_HEADERS $(TEST)

2) .c.s:
         $(CC) $(CFLAGS) -S -o $*.s $<
   .s.o:
         $(AS) -c -o $*.o $<
   .c.o:
         $(CC) $(CFLAGS) -c -o $*.o $<

3)  all:        zSystem

4)  zSystem:    piggy.o $(zOBJECTS)
                $(LD) $(LDFLAGS) -o zSystem -Ttext 1000 $(zOBJECTS) piggy.o

    head.o:     head.s

5)  head.s: head.S ../include/linux/tasks.h
         $(CPP) -traditional head.S -o head.s

6)  piggy.o:    $(SYSTEM) xtract piggyback
         ./xtract $(SYSTEM) | gzip -9 | ./piggyback > piggy.o

7)  $(SYSTEM):
```

```
$(MAKE) -C .. tools/zSystem
```

1) zSystem에 들어가는 내용의 순서를 알수 있다. 압축헤드용파일인 head.S,압축을 풀기위한 압축프로그램 코드,tools/zSystem압축된 내용의 순서로 들어가게 된다. zOBJECTS는 그림<1.2>에서 압축풀기 코드에 해당한다.
부팅시 bootsect.S,setup.S가 수행되고 나서 이 head.S가 수행되면서 압축코드를 이용해 여기서 압축했던 tools/zSystem을 풀어 원하는 위치(물리메모리 0x100000<1M>)에 갖다 놓고 boot/he-ad.S가 수행된다.

◆ kernel이 시작되는 메모리번지에 대해

zSystem.map내의 startup_32의 번지를 보자.

```
00100000 t startup_32
001000c0 t isnew
001000e7 t is486
.....
```

그리고, main-Makefile의 13번 line에 다음과 같은 우리의 규칙을 추가시켜 **zSystem2.map**을 생성시켜 보았다.

```
zBoot/zSystem: zBoot/*.c zBoot/*.S tools/zSystem
     $(MAKE) -C zBoot
     nm zBoot/zSystem | grep -v '\(compiled\)\|\(\.o$$\)\|\( a \)' | \
         sort > zSystem2.map
```

생성된 zSystem2.map내의 startup_32의 번지를 보자

```
00001000 t startup_32
00001058 T _huft_build
00001658 T _huft_free
.....
```

zSystem.map내의 startup_32는 boot/head.S에 있는 symbol로서 실제 kernel이 시작되는 번지를 가리키며, 우리가 만든 **zSystem2.map**내의 startup_32는 zBoot/head.S내의 symbol로서 압축 kernel이 놓여질 번지를 나타내고 있다.

2) -S옵션은 어셈블리어 source를 생성후 Stop
gcc의 -c 옵션은 compiling, 또는 assembling후에 정지, 즉 link는 하지 말라는 의미.

3) zSystem이 최종생성파일.

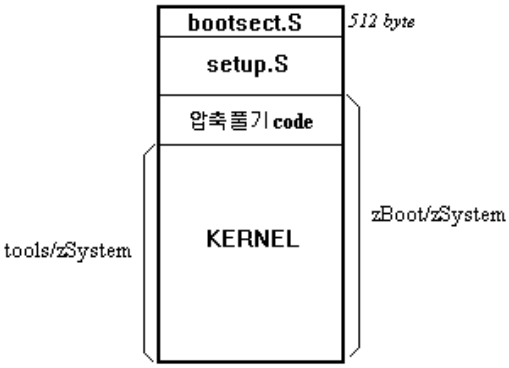
4) 0x1000번지를 text segment로 link함으로써 이 번지가 초기에 압축된 kernel이 놓이는 곳이 된다.

5) -traditinal옵션은 전형적인 C compiler의 특징을 지원한다.
(자세한 것은 GNU CC MANUAL참조)

6) 여기서 zSystem을 압축한다.
tools/zSystem에는 boot/bootsect.S와 boot/setup.S를 제외한 boot/head.S와 나머지 모든 ker-nel source들이 컴파일되어 들어간다. (**main Makefile** 12번 line)

7) -C옵션의 디렉토리(여기서는 ..)로 directory change후 make. tools/zSystem이 만들어지도록 한다.

결과로서 생기는 zImage file의 구조는 그림<1.2>과 같다.



<그림 1.2> zImage file구조

1.6 Kernel Image

만들어진 kernel image인 **zImage** file은 일반적으로 root directory에 놓이게 된다.

부팅 디스켓을 만들때 이 file이 디스켓의 맨 처음부터 들어가게되므로 bootsect.S가 디스켓의 head 0, cylinder 0, sector 1에 놓이게 된다.

```
# rdev zImage /dev/hda3
# cp zImge /dev/fd0
```

에 의해 부팅디스켓을 만들수 있다.

1.7 보호모드

Intel Microprocessor의 386DX급이상에서는 두가지 모드를 지원한다. 즉, 16 bit체계인 **실 모드**와 32 bit체계인 **보호모드**를 지원하는데 DOS가 실 모드를 이용하는 운영체제이며, OS/2, UNIX, WINDOWS-NT등이 보호모드를 이용하는 운영체제이다.
매킨토시에 장착된 M68계열 processor는 애초에 8 bit환경에서 32 bit환경으로 발전되었지만[6], INTEL의 8086계열은 같은시기에 16 bit체계를 선택했기 때문에 다시 32 bit체계를 덧붙이는 번거로움을 겪고 있는 것이다. DOS의 대중성때문에 Intel이 16 bit체계를 버릴수도 없는 것이 현실이다. DOS 응용소프트웨어를 완전하게 수용할수 있는 32 bit운영체제의 출현과 더불어 DOS 응용프로그램의 생산이 점진적으로 중단된다면, 어쩌면 그리 멀지않은 시기에 8086계열에서 실모드와 보호모드가 공존하는 지금과 같은 절름발이 형태의 CPU는 사라질지도 모른다. 이것은 리눅스가 지향하는 바이기도 하다.

부팅시 ROM의 POST(자기진단 프로그램)가 수행될때는 실 모드이나 32 bit운영체제의 경우 kernel에 의해 보호모드로 진입하게 된다. 보호모드에서는 실모드에 비해 다음과 같은 대표적인 장점을 가지고 있다. 이것은 DOS와 리눅스의 기능상의 차이이기도 하다.

1) 실 모드에서는 program의 코드부분[7]은 반드시 1M byte memory내에 있어야 한다. 그것은 16

---

6) 컴퓨터구조 이야기-영진출판사 211 page
7) program은 code부분과 data부분으로 나누어진다.

bit로 번지지정을 할수 있는 한계가 1M로 제한되어 있기 때문이다. 따라서 16 Mbyte의 RAM을 장착하였더라도 코드부분으로는 상용메모리인 640kbyte밖에 사용하지 못한다. 다음은 DOS상에서 mem 유틸리티를 사용하여 memory상태를 보았다.

```
D:\> MEM
Memory Type        Total =  Used  +  Free
----------------  ------   ------   ------
Conventional        640K      55K     585K
Upper                 0K       0K       0K
Adapter RAM/ROM     384K     384K       0K
Extended (XMS)     7168K    6384K     784K
----------------  ------   ------   ------
Total memory       8192K    6823K    1369K

Total under 1 MB    640K      55K     585K

Total Expanded (EMS)               6528K (6684672 bytes)
Free Expanded (EMS)                6144K (6291456 bytes)

Largest executable program size          585K (599024 bytes) ◀
Largest free upper memory block            0K (0 bytes)
MS-DOS is resident in the high memory area.
```

아래에서 3번째 이탤릭체로 된부분에서 보듯이 640 kbyte내에만 실행파일이 들어갈수있다.
반면, 보호모드에서는 memory 전부를 코드를 위해 사용할수 있다. 따라서 RAM이 많이 깔려있다면 상당히 큰 실행file도 수행시킬수 있는 것이다. 게다가 paging이란 기능을 사용하면 실메모리보다도 훨씬 큰 실행file을 수행할수 있는데 paging에 대해서는 7장 메모리 관리에서 다룬다.

2) 실 모드에서는 여러개의 program이 메모리에 올라와서 수행될수는 없다. 단지 interrupt를 이용하는 RAM상주 프로그램만이 메모리를 같이 차지할수 있는 것이다. 만약에 실 모드에서 여러개의 프로그램이 메모리에서 수행된다면 프로그램들이 다른 프로그램 영역을 침범해서 엉망이 될것이다. 물론 소프트웨어적으로 이것을 처리해 주는 방법이 있지만 그렇게 하면 운영체제의 부담이 커지게 된다. 보호모드에서는 이것을 processor가 처리해 줌으로써 메모리만 충분하다면 아주 많은 프로그램들이 한꺼번에 수행될수 있게 된다. 즉 보호모드에서 메모리에 있는 프로그램들은 각자의 영역에서 수행되며 특별한 권한이 없이는 다른 프로그램영역을 침범하지 못한다.

위의 두가지 이유에 의해 보호모드에서는 여러명의 user가 수많은 program을 수행한다.
앞으로는 수행중인 program을 *task* 또는 *process*라고 부르겠다. 굳이 이들 둘의 용어들에 의미 차이를 둘수도 있겠으나 내용의 간결성을 위해 본서에서는 같은 의미로 사용할것이다.

보호모드에서의 프로그래밍은 실모드에 비해 복잡한 편이다. kernel을 분석하기 위해서는 적어도 386 구조(architecture)와 관련된 책 한권쯤은 옆에 두고 보아야 할것으로 생각한다.

1.8 kernel hacking을 위한 유틸리티

다음에 kernel을 분석하기 위해 유용하게 사용될수 있는 유틸리티 몇가지를 소개한다.
자세한 용법에 대해서는 manual page를 참고하기 바란다.

1) nm

object file에서 symbol이 메모리상에 위치하는 지점을 알수 있다. 예를 들어 코드에 다음과 같은 부분이 있다고 치자.

```
unsigned int * handler
.....................
   handler();      /*   hd_interrupt함수 수행   */
```

실제로 리눅스 kernel에 handler()라고 정의된 함수는 없다. 따라서 이것은 kernel내의 어딘가에서 handler pointer변수에 함수 pointer를 다음과 같이 넣어두었을 것이다.

```
handler = hd_interrupt
```

그런데 이 부분을 찾지 못한다면 어떻게 할것인가? 즉 handler변수가 hd_interrupt함수 pointer를 가지고 있는지 모른다면 말이다.
이 때 printk함수[8]를 사용하여 handler값을 부팅중에 출력하게 만든다.

```
handler();
printK("handler : %x\n", handler);
```

8) 일단은 kernel에서 사용되는 printf함수라고 이해하자.

이제 kernel을 컴파일하고, 다시 kernel을 설치한후 부팅시킨다. 부팅중에 handler값을 읽어 zSystem.map안의 symbol의 번지와 비교해본다.

2) grep

file에서 원하는 문자열이 있는지 찾는다.

3) defrag 패키지

defrag패키지는 disk상의 file system을 관리하는 유틸리티이다. 본서에서는 이들 중에서 disk의 내용을 dump해 주는 **e2dump**를 주로 사용할것이다.

2장

부 팅 (초기화)

2.1 LILO

2.1.1 LILO란..

LILO(Linux loader)란 hard disk로 리눅스를 부팅시킬수 있도록 하기위해 주로 사용되는 프로그램이다. LILO가 설치되면 부팅초기에 DOS와 리눅스 그리고 다른 운영체제를 선택적으로 부팅시키는 것이 가능하다.

LILO source를 컴파일한 후 make install을 하면 **boot.b, os2_d.b, chain.b, any_d.b** 등의 파일이 /boot 또는 /etc/lilo에 설치된다. boot.b 내부는 2부분으로 나뉘어지는데 MBR(master boot sector)이 될 부분과 주변 device를 check해주는 프로그램이 들어있다. 전자는 LILO설치시 mbr에 들어가게 되며 후자는 MBR가 수행된후 곧 제어권을 넘겨받아 수행된다.

LILO 설치시 원래의 mbr은 /etc/LILO/boot.0300에 보관된다.[9] 0300은 device number로서 첫번째 hard disk에 LILO가 설치됨을 의미한다. os2_d.b는 0/S2를 선택적으로 부팅시키기 위해 사용된다. chain.b는 리눅스와 함께 다른 운영체제(예를들어 DOS)와 함께 설치하기 위해 사용된다. any_d.b는 두번째 디스크에 설치된 운영체제를 선택적으로 부팅시키기 위해 사용되는데, 예를 들

9) 이것은 LILO version 1.1의 경우이고, 1.3에서는 /boot/your_original_boot_sector에 보관되는 것을 확인했다.

어 DOS가 두번째 디스크에 설치되어 있다면, chain.b를 any_d.b로 바꾼다음 LILO를 설치하면 된다.

```
# boot/cp any_d.b chain.b
```

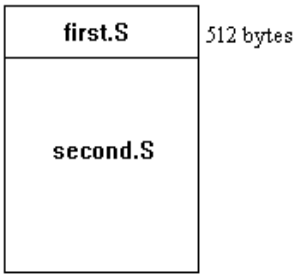
이 후 선택적으로 DOS를 부팅시 any_d.b의 코드가 수행되어 459K번지에 디스크를 절환하는 램상주 프로그램을 적재하게 되고, 부팅후에도 DOS가 첫번째 디스크에 설치된 것처럼 동작하게 된다. 본서에서는 DOS와 리눅스만을 선택적으로 부팅시키는 경우에 대해서만 논의한다.

LILO 설치후 '**map**'이라는 file이 생성된다. 이 file안에는 선택적으로 부팅시킬려는 각 운영체제의 bootsect 위치sector와 리눅스 kernel image가 들어있다.

2.1.2 LILO 설치

이 절은 **boot.b**와 **map** file에 대해 설명한다. 여기서는 단지 두 파일이 어떠한 구조로 되어 있는지만 아는 것으로 충분할것이다. 그 내용에 대해서는 부팅과정에서 자세하게 확인된다.

1) LILO source의 Makefile을 보면 알수 있듯이, boot.b에 first.S와 second.S가 그림<2.1>과 같이 들어간다.



<그림2.1>

2) LILO설치후 first.S는 **MBR**이 되며 뒷부분(0x1be위치)에 여전히 partition테이블을 가진다. 이 테이블은 change.S의 0x1be위치에도 들어가서 DOS를 부팅시킬때 사용된다.

3) LILO는 DOS와 리눅스를 프롬프트상에서 선택할수 있으며 DOS가 default인 다음과 같은 con-fig에서 그림<2.2>와 같은 map을 생성한다.

lilo를 위한 config

```
boot = /dev/hda
prompt
install = /boot/boot.b
#compact        # faster, but won't work on all systems.
#delay = 5      # optional, for systems that boot very quickly
timeout = 100
vga = normal    # force sane state
ramdisk = 0     # paranoia setting
  other = /dev/hda1
        label = dos
  image = /zImage
        label = linux
        root = /dev/hda3
```

다음은 그림<2.2>에 나와 있는 image descriptor table의 descriptor구조체와 내용이다.

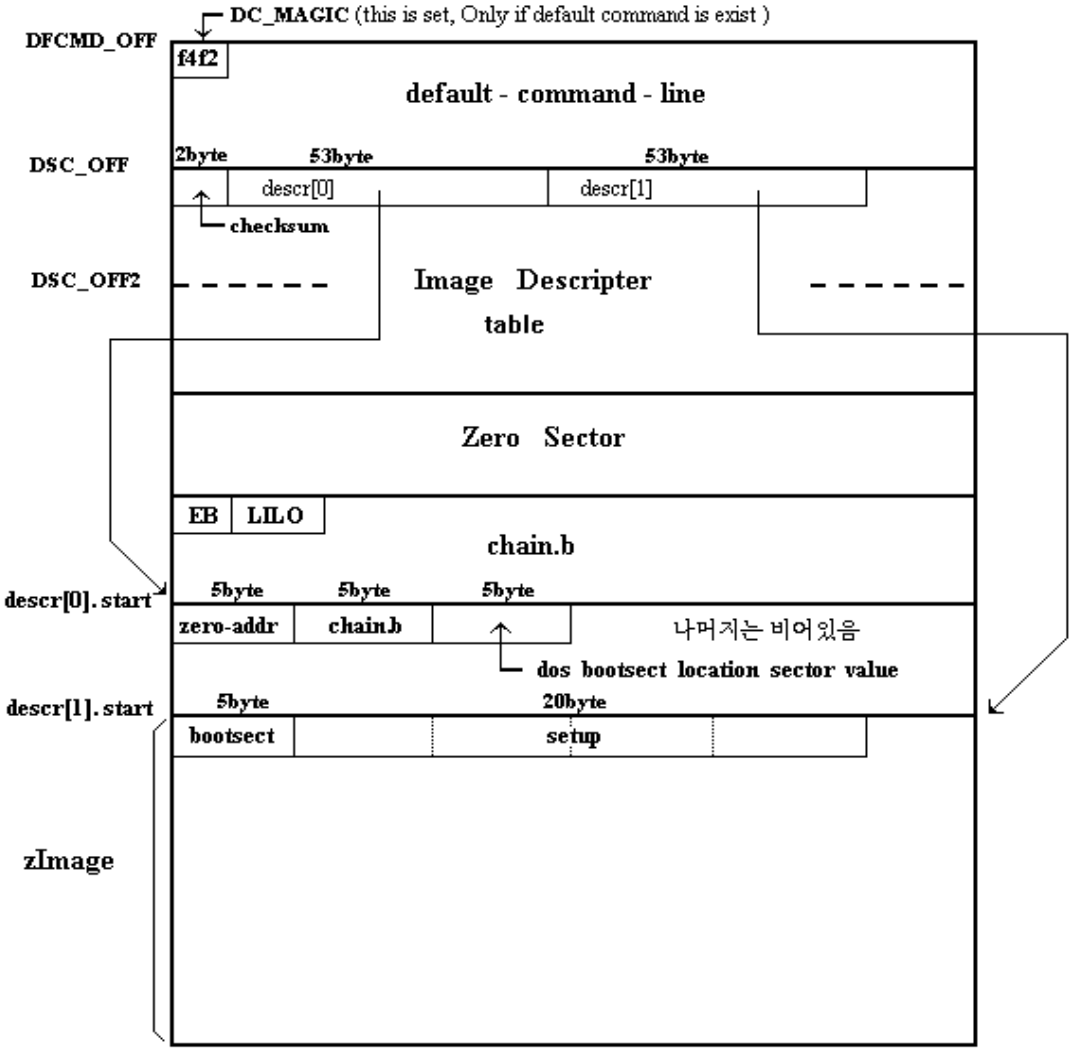
```
typedef struct {
    char name[MAX_IMAGE_NAME+1];
    char password[MAX_PW+1];
    SECTOR_ADDR start;
    unsigned short present; /* parameter bit map */
    unsigned short root_ro,ram_disk,vga_mode,root_dev;
        /* override image settings */
    unsigned short bss_seg,bss_segs,bss_words;
        /* BSS := bss_seg:0 to (bss_seg+bss_segs-1):(bss_words*2)
           bss_seg == 0: no initialization of an unstripped kernel
           bss_words == 0: don't initialize remaining bytes in last sector */
} IMAGE_DESCR;
```

◆ descriptor 53byte 내용

1) image 이름(config에서 label=) **16byte**
2) password **16byte**
3) map file에서 name에 해당하는 section의 위치.
(sect,trk,dev,head,sect-num) **5byte**
4) present bits **2byte**
5)번에 해당하는 내용의 값이 config나 command line에 의해 재지정되면 이 bit들의 아래bit부터 bit 0(ram_disk), bit 1(vga_mode), bit 2(root_dev), bit 3(root_ro)가 각각 set 된다.
나머지 bit는 사용되지 않는다.
5) root_ro,ram_disk,vga_mode,root_dev **8byte(2byte씩)**

```
root_ro     : root filesystem read only
ram_disk    : ramdisk size re-set
root_dev    : root partition re-set
vga_mode    : VGA mode re-set
```

6) bss_seg,bss_segs,bss_word **6byte(2byte씩)**
mkck script를 사용하여 리눅스 kernel을 compound unstripped kernel로 만들어 사용할 경우 값들이 지정된다. default값은 각각 0이며, 본서에서는 mkck를 사용하지 않는 것을 기준으로 한다.
mkck script와 LILO TECHNICAL MANUAL 참조.



<그림2.2> LILO MAP FILE

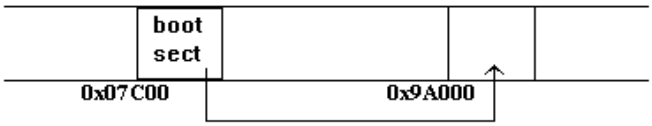
2.2 리눅스 부팅과정

2.2.1 부팅시작

LILO 설치할때 mbr에 들어간 first.S의 컴파일된 코드가 0x07c00에 올라오며 first.S에 제어가 넘어간다.

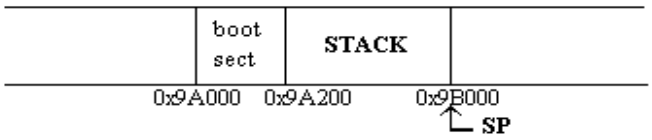
2.2.2 first.S 수행과정

1) 0x07c00에 있는 first.S를 0x9A000로 자기복제한다.



<그림2.3>

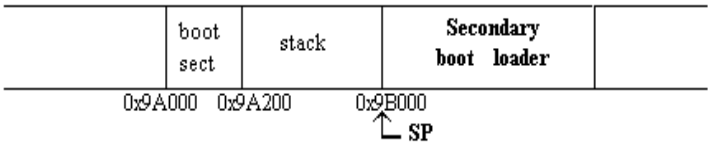
2) set **Stack**.



<그림2.4>

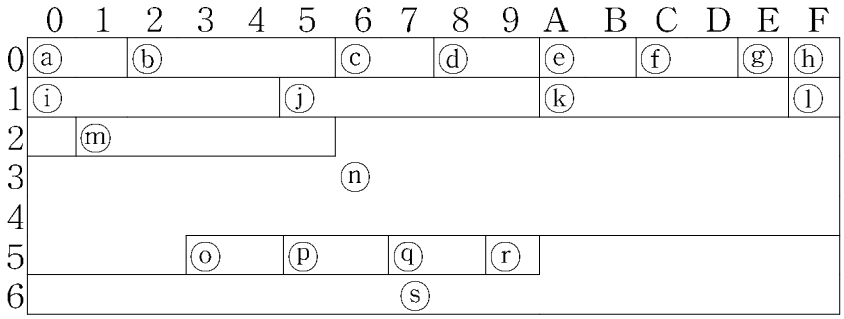
3) 'L' display.

4) load된 mbr내의 second.S의 위치섹터값(<표2.1>의 ⓝ)을 찾아서 0x9B000 ~ 0x9D000에 loa-d.



<그림2.5>

5) 'I' display.



<표2.1> **lilo 설치후 마스터 부트섹터**

ⓐ EB(jump의 기계어)

ⓑ "**LILO**" 문자열

ⓒ stage

STAGE_FIRST인 1이 들어간다. sencond.S의 stage는 STAGE_SECOND가 된다.)

ⓓ LILO package version.

ⓔ timeout

프롬프트에서 이 시간동안 키보드입력이 없으면 자동 boot. default 값은 무한대.

ⓕ delay

첫 image를 부팅하기 전에 기다리는 시간. serial이 set되면 이 값은 20이 된다.

LILO user manual 참조.

ⓖ port

remote login시 사용되는 port. serial 옵션에 의해 set되며 이 값이 1이면 COM 1을 의미한다.

LILO user manual 참조.

ⓗ serial number

RS-232C와 관련된 값. bps,parity,charactor bits.

LILO user manual 참조.

ⓘ DSC_OFF 위치값.

5byte는 각각 MAP file의 image descriptor부분의 DSC_OFF(표<2.2>의 LILO map file참조)의 secter,track,device,head,secter수 등을 나타낸다.

ⓙ DSC_OFF2 위치값.

5byte는 각각 MAP file의 image descriptor부분의 DSC_OFF2(표<2.2>의 LILO map file 참조)의 secter,track,device,head,secter수 등을 나타낸다.

ⓚ DFCMD_OFF 위치값.

5byte는 각각 MAP file의 image descriptor부분의 DFCMD_OFF(표<2.2>의 LILO map fi -le참조)의 secter,track,device,head,secter수 등을 나타낸다.

ⓛ message 길이.

16bit이므로 message의 최대길이는 64k이다.

ⓜ message 파일 위치.

5byte는 각각 message 파일위치의 secter,track,device,head,secter수 등을 나타낸다.

ⓝ secondary loader 위치값.

5byte는 각각 boot.b file의 second.S코드가 있는 부분의 secter,track,device,head,secter수 등을 나타낸다.

9개의 sector를 지정할수 있는 공간이 있으나, 실제로 앞쪽의 7 sector부분만을 사용한다.

ⓞ~ⓡ first.S 코드에서 부팅시 ROM에서 넘어온 si,es,bx,dl값이 들어가는 공간이나 bootsect에서 이곳은 비어있다.

ⓢ 이하에는 first.S 코드가 들어간다.

ⓣ bootsect의 마지막 64byte에는 기존의 mbr에 있던 *partition table*이 들어간다.

2.2.3 second.S 수행과정

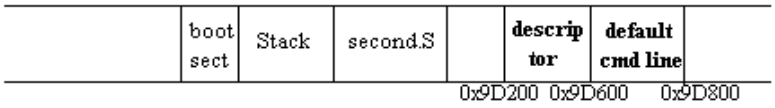
1) bootsect의 serial영역에서 port값 있는지 확인.
2) "**L**" display.
3) break flag를 clear

   LILO user manual 의 serial=param 옵션 참조.
4) prompt에서 입력대기시간 초과를 다루는 timeout handler호출.
5) second.S가 제대로 load되었나 확인.
6) bootsect의 Image descriptor위치 값으로 descriptor load.
7) descriptor **checksum.**

   descriptor의 내용을 2byte단위로 모두 더해서 0xabcd(INIT_CKS)와
   checksum을 더하면 0이 되어야 한다.
8) **default command line**을 load.

   LILO설치시 LILO명령에 따르는 옵션의 내용이다. **LILO user manual**의 command line

options참조하라.



<그림2.9>

9) 0x9D600에 있는 2byte가 DC_MAGIC값과 같은지 확인.
default command line이 없으면 공란.

dookay :

10) "O" display.
11) delay=0로 만든다.
12) 기존 delay = 0xffff이면 interactive prompt를 위한 외부 parameter check.
13) first.S의 ext_dl이 EX_DL_MAG인지 check.

noex :

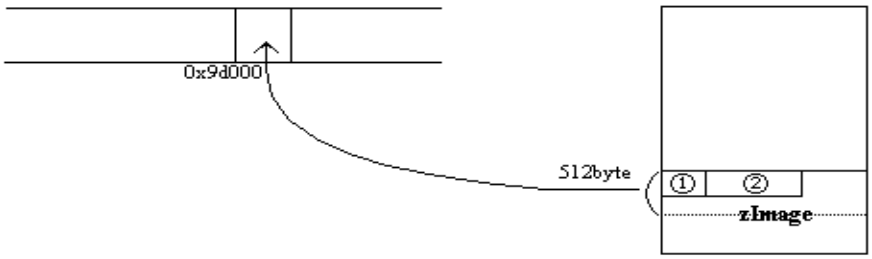
14) default command line check후 default command가 없으면 "**boot :**" prompt를 출력하고 command를 받아들이는 곳으로 jump.

iloop :

15) message file이 있는지 check.
16) "**boot :**" display.
17) interactive mode인지 check후 key입력 받음. key내용은 "**cmdline :**" 다음에 놓인다.
18) key를 받으면 key값에서의 image이름(DOS 또는 리눅스)과 map file의 Image descriptor section내에서 같은 이름을 가진 descriptor를 찾아 start값을 취한다.

doboot :

19) Image descriptor section내에서 같은 이름을 가진 descriptor의 start pointer(push bx)
20) image name display.
21) map file에서 bx값에 해당하는 descriper section의 첫 sector load.

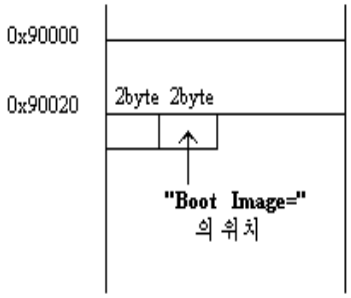


<그림2.7>

◆ 이것으로서 우리는 map file에 zImage내용이 있으므로 LILO설치후 root directory 에 있는 zImage 파일을 지워도 부팅이 된다.(향후 LILO 다시 설치할때 문제는 되겠지만..)

22) 그림<2.7>에서 ① 값을 이용, 0x90000에 zImage의 bootsect를 load한다. 이 bootsect는 /boot /first.S로서 floppy 디스켓으로 부팅할때 사용된다.
이 과정은 2.3절에서 설명된다.

23) CL_MAGIC(a33f)와 "**Boot Image=**" 의 위치를 0x90020에 넣는다.

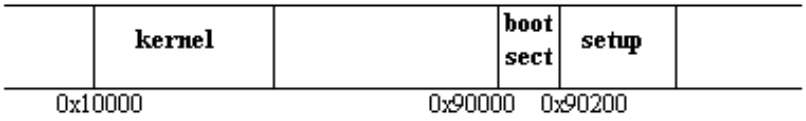


<그림2.8>

23) 해당 Image의 Image descriptor의 내용 check.

24) ② 값을 이용, map file 에서 setup load.

25) 0x10000에 system load.

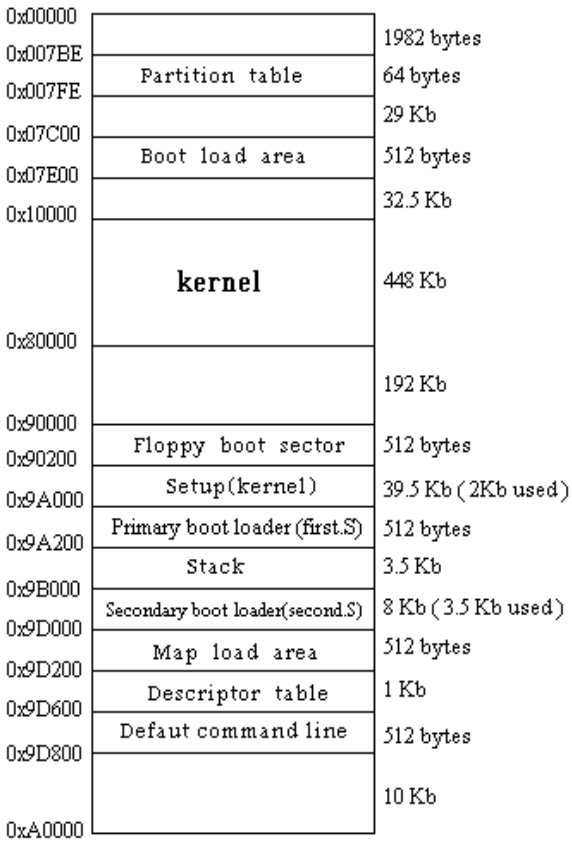


<그림2.9>

26) **setup**에 control 넘김.

2.2.4 LILO 작업이 끝난 후의 메모리 상태.

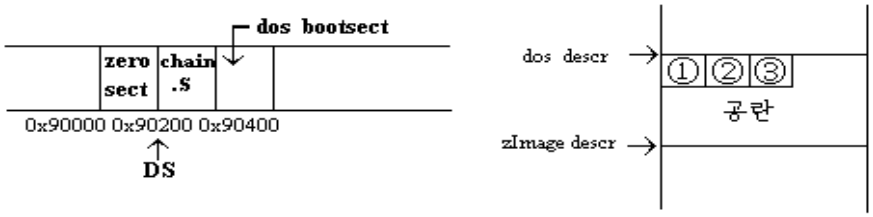
독자의 편의를 위해 LILO pakage에 포함된 **LILO technical manual**의 표를 그대로 옮긴다.



<그림2.10>

2.3 DOS 부팅의 경우

21) map file의 ①,②,③ 에 의해 다음과 같이 도스 부팅을 위한 내용이 load.



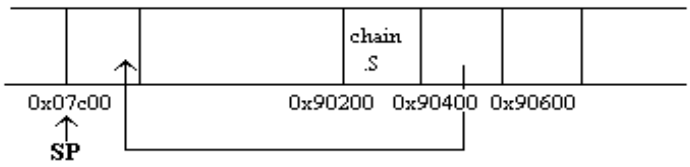
<그림2.11>

① zero_sect 위치
② chain.S 위치
③ DOS bootsect 위치
(이것이 기존의 mbr은 아니다.)

22) chain.S에 제어 넘김.

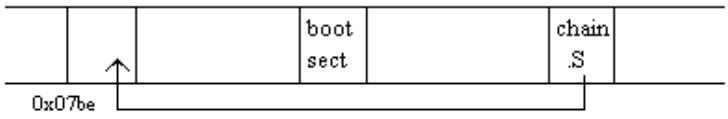
◎ chain.S 코드 수행과정

1) **bootsect** move.



<그림2.15>

2) chain.S 내의 **partition table** move.



<그림2.16>

3) DS : SI, ES : SI <= partition table pointer

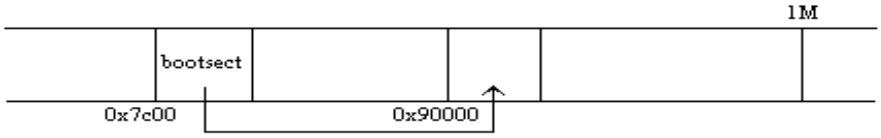
4) dx <= drive and head
5) DS, ES <= 0
6) ax <= 0xaa55 (**LILO TECHNICAL MANUAL** 참조)
7) **bootsect**에 control 넘김.

2.4 커널 코드 시작

2.4.1 bootsect.S (floppy로 부팅시 사용) 수행과정

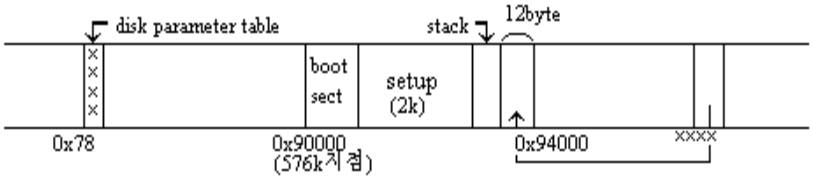
hard disk로 부팅시는 사용되지 않는다.

1) bootsect move.



<그림2.17>

2) 0x78번지에 들어있는 disk parameter table 주소를 이용하여 disk parameter 12byte 90000h + 4000h - 12의 위치부터 복사해 넣는다.[10]



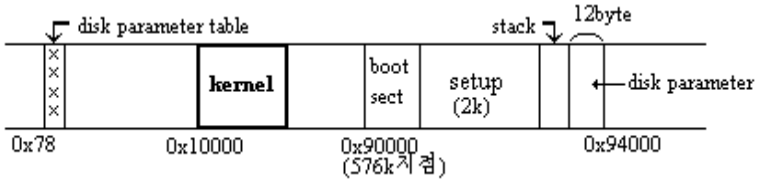
<그림2.15>

3) 0x78번지의 내용을 90000h + 40000h + 12로 고친다.
4) 옮겨놓은 disk param의 5번지의 값을 18(1.44M의 sectors per track)로 고친다.
5) FDC(floppy disk controller) RESET.
6) setup 코드 읽어들임.(그림<2.15> 참조)

---

10) **프로그래머를 위한 pc source book 하 7-22(매크로)**에서 비디오 param, disk param 참조.

7) sector 18에서 data read 가능여부 확인하여 cx에 그 값(읽기 가능 최대 sector수)을 넣는다.
8) 읽을수 없으면 sector 15(1.2M)와 sector 9(360K)에서 확인하여 cx에 그 값을 넣는다.
9) sectors <= cx.
10) "**loading...**" display.
11) kernel을 옮길 곳이 64KB(1 segment)경계인지 확인.(아니면 무한loop에 의한 부팅정지)
12) 0x7F000(508K) 만큼을 0x10000에 읽어들임.

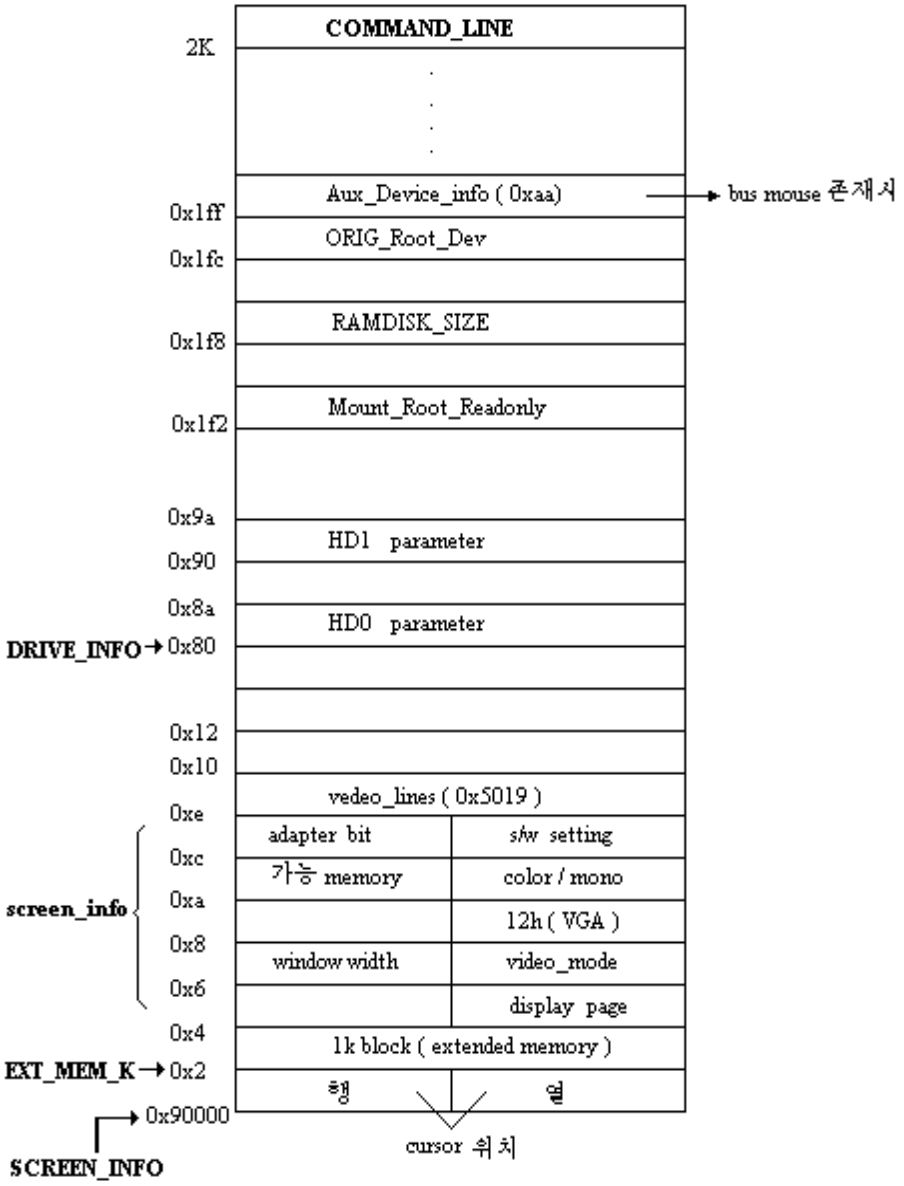


<그림2.19>

13) offset 508에 위치한 root_dev부분을 주목하자. ***rdev유틸리티***가 하는 일은, kernel 이미지 파일에서 이곳의 내용을 해당 root device로 갱신하는 것이다. 이 **ROOT_DEV**변수는 부팅(boot-ing)후반과정에서 mount_root함수에 의해 root device를 mount할때 사용된다.

2.4.2 setup.S 코드 수행과정

1) extended memeory 크기 구해서 0x90000+2에 넣는다.
2) keyboard repeat rate를 최대값(지연시간 **250ms**, 타자율 **30cps**)으로 둔다.
3) EGA / VGA config 값 구함.
(BH = color/mono BL = 가능 memory CH = adapter bit CL = s/w setting)
< VGA 일 경우 AL = 12h, 아닐 경우 AL = 0h >
4) VGA 인지, 다른 graphic card 인지를 check한다.
5) video card data 구함.(4,5,6,7 bytes)
6) 41h와 46h에 있는 hd0, hd1 parameter table address를 이용하여 90000h+80h와 90000h+90h에 hd0, hd1 parameter을 각각 10byte씩 복사한다.
7) hd1이 있는지 확인.
8) 없으면 hd1 parameter 영역 90h에 0로 10byte 채움.

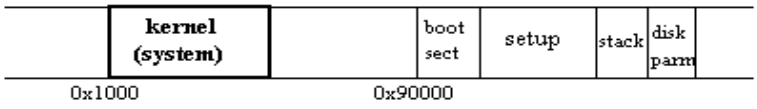


<그림2.20>

9) mouse가 존재하면 90000h+1ffh에 0xaa넣음.

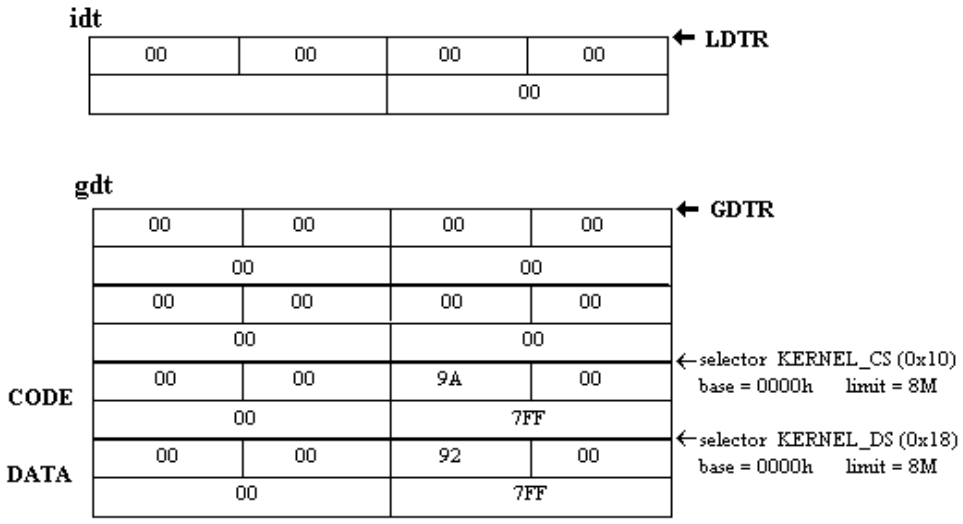
10) NMI를 포함하여 모든 인터럽트 사용을 불가능하게 disable.(out 0x70, 0x80)

11) 0x10000에 있는 커널을 0x1000으로 576k(9000h-1000h)를 옮김.



<그림2.21>

12) ds를 90200h로 setup후 idt, gdt load.



<그림2.22>

13) keyboard buffer가 비었는지 확인.(empty_8042)

14) A20 line on.[11]

15) coprocessor reset.

16) IRQ reprogram.

INTEL은 256개의 인터럽트 테이블중 처음 32개(00h - 1Fh)를 사용하지 못하도록 하였으나, IBM은 이를 어겼다.[12] 보호모드가 생기기 전까지는 **INT 8 ~ INT F**가 CPU에서 사용되지 않은채 예약된 상태로 있었다. IBM은 이들을 자의로 BIOS나 timer, coprocessor, printer등과 같은 device들을 위해 IRQ 0 - IRQ 7을 할당한 것이다.

결국 이것이 보호모드가 있는 286기종이상에서는 문제를 일으키게 되었다. 보호모드에서 발생

---

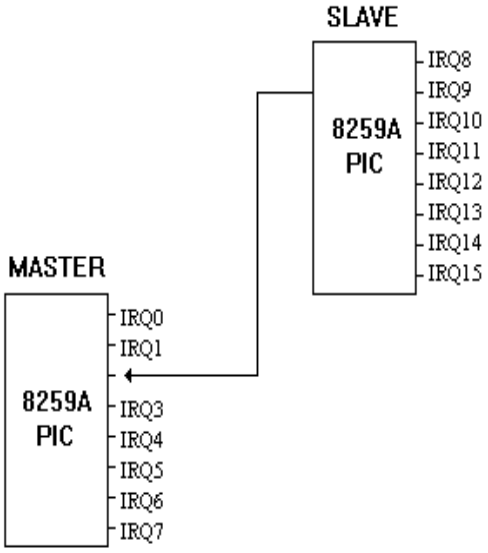
3) Ralf Brown의 Interuppt list 참조.

12) 컴퓨터 구조이야기-영진출판사(이승택 저) 165page

되는 다양한 예외를 처리하기 위해 드디어 CPU가 이들을 사용하게 되었고, 그래서 현재 **INT 8 ~ INT F** 범위에서는 IBM에서 제공하는 기능과 INTEL에서 제공되는 기능이 충돌을 일으키는 것이다.

리눅스는 이러한 혼란을 막기위해 IRQ들을 새로운 인터럽트 번호들과 연결한다. 즉 INT 0x20 ~ INT 0x2F에 각각 IRQ 0 ~ IRQ 1을 할당한다. The Undocumented PC-Wesley의 821 page에 동일한 코드가 실려있다.
PIC(INTERRUPT CONTROLLER) 8259A에 대해서는 관련 서적을 참고하라.[13]

◆ 0x20 port에 0x11(이것이 OCW1에 해당)을 out한 후에는 0x21에 값을 세차례 out함에 따라, OCW2,OCW3,OCW4에 차례로 값이 들어가면서 INT번호 전환이 이루어진다.[14]



<그림2.20> PIC에서 master와 slave의 연결상태.

13) 컴퓨터 구조이야기-영진출판사(이승택 저) 163 page.
14) 본인은 보다 자세한 내용을 위한 자료를 찾지 못했다.

17) **PROTECTED mode**로 전환.

PE set 후에는 반드시 flush해야한다. processor는 instruction prefetch queue에 다른 명령어도 미리 가져온다. 실 모드일때의 정보를 flush시키는 것이다.[15]

```
msw <= 0x0001 (PE bit)
```

18) jmp 1000h, KERNEL_CS (kernel이 있는 곳으로 jump)

◆ KERNEL_CS는 selector로서 0x10.

이진수로 0001 0000, selector번지는 10 (RPL : 00, TI : 0)

KERNEL_DS는 selector로서 0x18.

이진수로 0001 1000, selector번지는 11 (RPL : 00, TI : 0)

▷ TI bit : 지역 테이블인지 전역 테이블인지를 가리킨다.

2.4.3 head.S 코드 수행과정

0x1000로 제어가 넘어가면 1장에서 기술한대로 zboot/head.S(압축풀기 코드)가 수행되면서 압축이 풀린 kernel이 0x100000(1M위치)로 이동하게 된다.

◆ 여기서부터는 32 BIT 코드이면서 AT&T System V/386 문법을 따르고 있다.[16] 따라서 boot-sect.S와 setup.S는 16bit assembler인 as86을 사용하나, 지금부터는 32 bit assembler인 as(GNU AS)를 사용해야 한다.

DOS의 Macro assembler와 리눅스의 as86은 Intel 문법을 따르고 있다. 참고로 head.S 코드 내에서 1f나 2b와 같은 코드는 현위치 이후(forward)에 있는 1: lable과, 현위치 이전(backward)에 있는 2: lable을 각각 의미한다.

1) KERNEL_DS인 18h가 ds,es,fs,gs에 들어간다.

2) kernel/sched.c에 있는 stack_start구조체 pointer를 ss,sp에 넣는다.

따라서 ss에는 KERNEL_DS가, sp에는 user_stack pointer가 들어가며 stack size는 **1 page**가 된다.

3) **edata**에서 **end**까지(BSS 영역) 초기화(clear)

---

15) 80386 프로그램 입문 - 교학사. 184page

16) GNU AS MANNUAL 의 8.9 80386 Dependant Features를 참조.

BSS(block started by symbol)는 초기화되지 않은(uninitialized)영역으로서 프로그램이 시작될때 커널에 의해 초기화된다.[17]

edata와 **end**는 초기화된 data의 끝과 전체 프로그램의 끝을 나타내며, **etext**는 프로그램 코드의 끝을 나타낸다. 이들 변수는 gcc에 의해 정의된다.

4) setup idt.

idt를 ignore_int로 향하도록하여 interrupt가 발생하면 무시하도록 하다.

KERNEL_CS		ignore_int	
00	00	8E	00

위 디스크립터 256개로 코드 뒷부분의 _idt영역을 채운다.

이렇게 만들어진 table은 이후 lidt명령으로 idt_descr(코드 뒷부분)을 IDTR(IDT Register)에 넣음으로써 interrupt descriptor table의 base번지와 limit가 지정된다.

base번지는 선형번지가 된다.

다음은 IDTR에 들어가는 descriptor이다.

0xc0000000+_idt (base)
256*8-1(limit)

DOS의 경우는 인터럽트 벡터의 단위가 4byte이나 보호모드에서는 interrupt descriptor가 윗그림처럼 8byte단위로 포인터가 이루어진다.

5) A20 line enable인지 check.(아니면 무한 loop에 의한 부팅중지)

6) eflag 초기화.

7) boot parameter와 default command line parameter를 0x105000(empty_zero_page)로 옮긴다.

8) EFLAG의 **AC** bit를 change 시켜본다.(실패이면 386 processor 이하)

이 bit는 486에서 바뀔수 있다. 386이하에서는 바뀌지 않는다.[18]

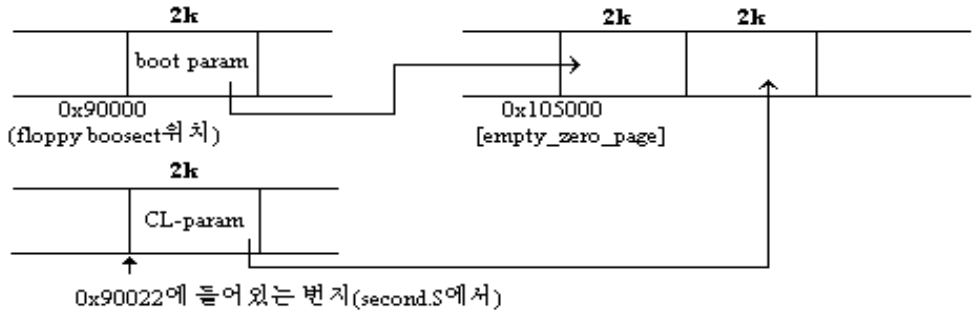
9) EFLAG의 **ID** bit를 change 시켜본다.(실패이면 486 processor 이하)

10) 둘다 실패이면 사용 CPU 확인.

다음은 PC통신을 통하여 얻은 **pentium**명령어 목록에서 발췌한 내용이다.

---

17) Advanced Programming in the UNIX Environment (W.Richard Stevens) 167page



<그림2.24>

CPUID CPU Identification
Provides information to the software about the model of microprocessor

on which it is executing. An input value loaded into the EAX register for this instruction indicates what information should be returned by the CPUID instruction. The value returned for the Family is 5, indicating the Pentium microprocessor. The Pentium will return a 0 in the model field to indicate that it is the first model in the Pentium family. The Stepping ID field will contain a unique identifier for the revision level. The remaining bits are reserved.

Flags: No flags affected.
Encoding: ***00001111 10100010***

Syntax	Example	Clock Cycles
CPUID	cpuid	14

11) 486은 cr0의 PG,PE,ET,AM,WP,NE,MP를 set.
12) 386은 cr0의 PG,PE,ET,MP를 set.
표<2.2>에 control register 0의 bit들이 잘 나타나 있다.

---

18) **The Undocumented PC-Wesley** 53page 참조.

bit	기 능	적용되는 기종
PG	Paging enable	386, 486
CD	Caching disable	486
NW	Writes-transparent control	486
AM	Alignment check mask	486
WP	Supervisor write protect	486
NE	Numerics exception control	486
ET	Processor extention (287, 387 확인)	초창기 386
TS	Task switch occurred	286, 376, 386, 486
EM	Emulate processor extension	286, 376, 386, 486
MP	Monitor coprocessor	286, 376, 386, 486
OE	Enable segmented protection	286,(376), 386, 486

<표2.2> cr0 bit들

13) 87 processor(coprocessor) 존재 여부 check.

14) setup paging.

리눅스에서 user 메모리는 선형번지 3G(0xc0000000)의 바로 아래까지이며, 3G부터는 kernel을 위한 선형메모리 공간이 된다.(표<2.4>참조[19])

swapper_pg_dir(0x1000)부터는 4바이트 크기의 page directory entry가 1024개 들어가며, pg0(0x2000)부터는 역시 4바이트 크기의 page table entry가 1024개 들어간다. 그림<2.22>에서 보다시피 page directory영역에는 2개의 entry만 있고, page table영역에는 entry들이 모두 채워진다.

page table의 base값들인 0x00 ~ 0x3FF는 4Mbyte의 실메모리를 page[20]단위인 4096로 나눈 것들의 각각을 가리킨다.(즉 base값이 0x01이면 실메모리의 0x01000를 의미한다.)

두 directory entry는 둘다 같은 page table인 pg0를 가리킨다. 따라서 user 선형영역의 맨 아래 4M와 kernel 선형영역의 맨 아래 4M는 같이 물리메모리 4M를 공유한다. 물리메모리가 더 설치된 경우의 초기화는 나중에 이루어지며(paging_init), 여기서는 4M만 MAPPING시킨

---

19) 독자의 편의를 위해 GNU document인 **kernel hacker's guide** 에서 발췌한

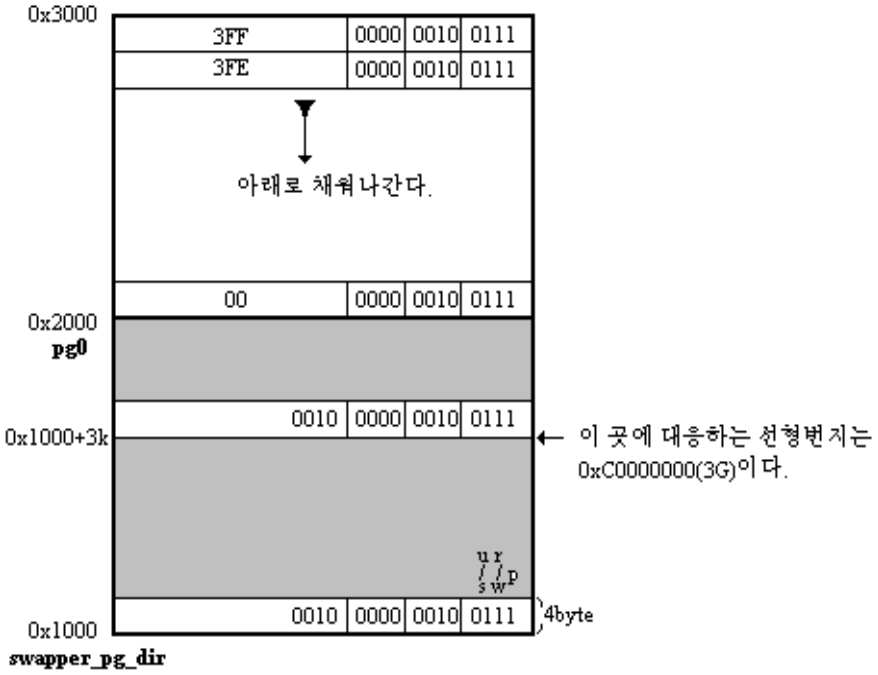
- 4.3 A user process' view of memory

와 압축kernel의

- 물리메모리(hacker's guide것과 비교해 보자.)

20) 1 page는 PAGE_SIZE인 4K이다. 이것은 386, 486에 의해 정해진 값이다.

다.



<그림2.25> paging 초기화

- RW =1 이면 R/W 가능상태이다.
- P =1 이면 물리메모리에 page가 존재한다.
- U/S = 1 이면 user상태
  0 이면 supervisor상태

	FREE	momory_end or high_memory
	mem_map	mem_init()
	inode_table	inode_init()
	device data	device_init()
&end	more **pg_tables**	paging_init()
&etext	kernel data	
	kernel code	
0x106000	floppy_track_buffer	
0x105000	zero_page	
0x103000	bad_pg_table bad_page	used by **page_fault_handler**s to kill processes gracefully when out of memory.
0x102000	pg0	the first kernel page table.
0x101000	swapper_pg_dir	the kernel page directory.
0x100000	null page	
0x0A0000	system BIOS	640K
low_memory_start	**FREE**	
0x000000	**RESERVED**	

<표2.3>

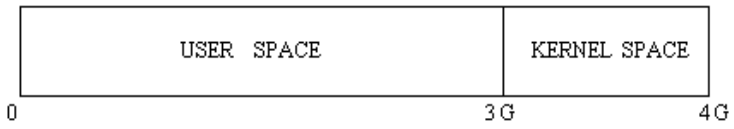
0xc0000000	The invisible kernel	reserved
	initial stack	
	room for stack growth	4 pages
0x60000000	shared libraries	
brk	unused	
	malloc memory	
end_data	uninitialized data	
end_code	initialized data	
0x00000000	text	

<표2.4>

현재 Kernel이 실메모리 0x100000에 있으므로 paging후 Kernel은 선형번지 0xc0100000에 있는 것이 된다. 그러나 Kernel은 여전히 선형번지 0x100000에 있기도 하다.<그림2.25>

kernel은 선형공간의 초기 4M와 mapping되어져야 한다. paging이후 kernel은 선형공간에서 수행된다. 즉 kernel내부의 모든 번지값들은 선형번지가 되는 것이다. 그것은 kernel이 paging 이후에도 컴파일할때 번지값들을 그대로 가지고 있기 때문이다. 따라서 선형공간에서도 여전히 물리메모리에서와 같은 위치에 있어야 한다.

3G이후의 mapping은 user공간과 kernel공간과의 연결을 위한 통로로 사용된다. 뒤에서 논의되겠지만 모든 process는 fork에 의해 자신의 page directory를 가지고, 4G의 선형공간을 소유하게 된다. 각 process의 선형공간은 그림<2.23>과 같이 아래 3G는 user공간으로, 위쪽 1G는 kernel공간으로 사용된다.



<그림2.26> process의 선형공간

실제로 특권레벨이 낮은 user공간(dpl=3)[21]에서 레벨이 높은 kernel공간(dpl=0)을 곧바로 접근할수는 없다. kernel 공간을 접근하기 위해서는 gate를 사용해야 한다. gate에 대해서는 2.5.7절에서 논의된다.

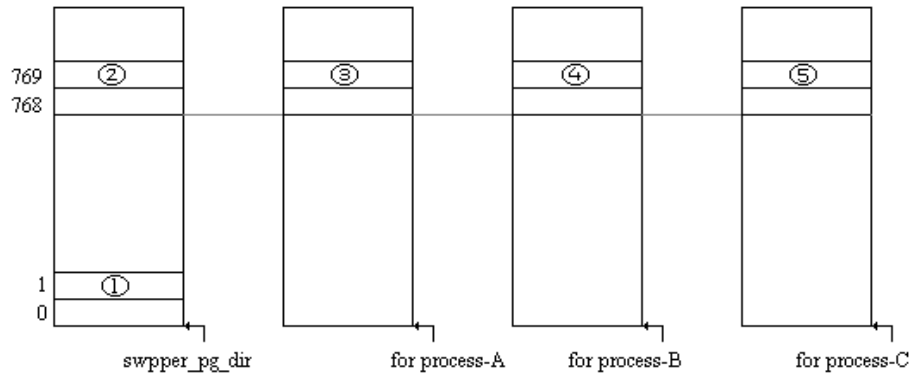
수행되는 user process입장에서 보면 3G이후에 kernel이 존재한다. 만약 kernel을 위한 page directory인 swapper_page_dir의 내용이 바뀌면 모든 process의 page directory의 768 entry부터 1023 entry의 내용도 바뀐다. 즉 그림<2.24>에서 보이는 ①이 바뀌면 ②와 process들의 ③, ④, ⑤ 가 모두 한꺼번에 갱신되는 것이다. 7.9절의 alloc_area_pages함수를 참고하라.

모든 process page direcory의 kernel을 위한 entry들은 모두가 한개의 kernel을 가리키고 있음을 잊지말자.

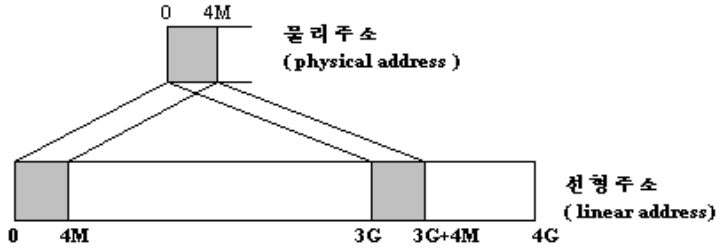
system call에 의해 kernel에 접근할때 user process(user mode)는 3G부터의 mapping을 통하지만(gate인자인 KENREL_CS에 의해) kernel mode에 진입하면 그 다음부터의 kernel 번지지정은 kernel 자신이 가지고 있는 선형공간(swapper_pg_dir)내의 아래쪽에서 수행된다. kernel선형공간에 2개의 kernel이 존재하기 때문에 가능한것이다.

---

21) descriptor's privilege level



<그림2.24> kernel과 process들의 page directory



<그림2.28> 물리주소와 선형주소의 mapping

마지막으로 swapper_pg_table 위치를 cr3 레지스터에 넣고, cr0 레지스터의 PG를 set하여 pagi-ng을 enable시킨다.

15) GDT setup.

descriptor 2 0x10 kernel 1G code
descriptor 3 0x18 kernel 1G data
descriptor 4 0x23 user 3G code
descriptor 5 0x26 user 3G data

앞에서 설명했듯이 0x10,0x18,0x23,0x26은 이들 네 segment에 접근하기 위한 selector값들이

다. 각 값들의 3, 4bit에 의해 gdt에서의 descriptor의 위치인 2,3,4,5가 정해진다. 표<2.5> descriptor들의 base값에서 보듯이 user공간은 0 ~ 3G까지, kernel공간은 3G ~ 4G까지이다. 물리메모리의 특정부분이 user공간이거나, kernel공간인 것은 아니다. 선형공간에 접근하기 위해서는 요구된 선형번지에 해당하는 물리 page가 필요하고 그때 선형번지공간과 물리 page사이에 mapping이 이루어진다.[22] 이 과정은 7장 메모리관리에서 논의된다.

22) 이미 앞에서 소개한 386관련 책들과 다음의 책을 참고할수 있다.
80X86 Architecture & Programming Vol. I 과 Vol. II: **Agarwal - Prentice Hall**

* : data=0, code=1

base	G B O AVL	limit	p DPL 1	* E W A	base	
base			limit			descriptor 구조
00	00		00		00	GDT start
00			00			0
00	00		00		00	
00			00			1
0xC0	1 1 0 0	0x03	1 0 1	1 0 1 0	0x00	
00			0xFFFF			2
0xC0	1 1 0 0	0x03	1 0 1	0 0 1 0	0x00	
00			0xFFFF			3
0x00	1 1 0 0	0x0B	1 3 1	1 0 1 0	0x00	
00			0xFFFF			4
0x00	1 1 0 0	0x0B	1 3 1	0 0 1 0	0x00	
00			0xFFFF			5
00	00		00		00	
00			00			6
00	00		00		00	
00			00			7
						TSS0
						LDT0
						TSS1
						LDT1
						나머지 entries (for LDT and TSS)

<표2.5>

LDT와 TSS는 TASK당 1쌍씩 주어진다. 즉 task[0]에 해당하는 TSS와 LDT는 <표 2.5>에서 TSS0와 LDT0이다. 최고 128(NR_TASK)쌍의 LDT와 TSS가 GDT에 상주할수 있다. 그래서 GDT 최대 descriptor수는 8+128*2가 된다.[23]

23) <표2.5>는 include/linux/sched.h 참조

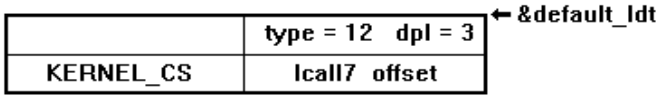
16) GDTR과 IDTR을 set하고 gdt entry들을 set하였으므로 ljmp 명령어를 사용하여 selector를 통한 instruction pointer(ip)의 이동을 한다.

17) LDTR이 0로 초기화되며 차후 load_LDT()라는 함수에 의해 다시 set되어진다.

18) init/main.c의 start_kernel 함수로 제어가 넘어간다.

2.5 start_kernel routine

2.5.1 iBCS emulator



<그림2.29> 1default local descriptor

1. lcall7을 set_call_gate한다.(kernel/syscall.S) call gate descriptor는 그림<2.26>과 같다. default_ldt은 task가 다른 local descriptor를 사용하지 않을때 사용된다.[24] 즉 task가 local des-criptor를 access하면 곧 lcall7 함수가 호출된다. 이 후 **call gate**를 set 하는 경우는 없다.[25]

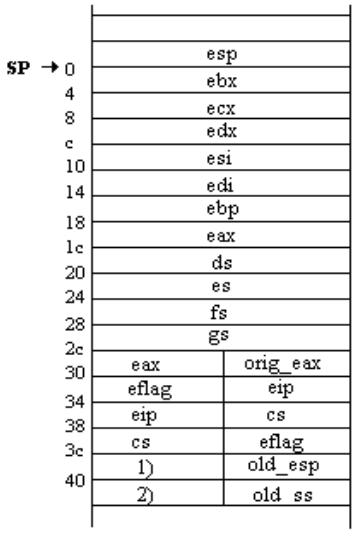
2. lcall7 함수 (kernel/syscall.S)는 kernel 1.0에서는 지원 안되는 iBCS emulator를 위한 call gate이다.

   <1> 그림<2.27>에서 맨 아랫부분의 왼쪽 반은 C함수를 어셈블리어로 구현하면서 함수 호출시 스택에 push된 내용이다. 1)과 2)는 C함수의 parameter들[26]이다.
   다음 함수를 호출하기 위해 **struct pt_regs**의 형태인 오른쪽 반의 내용으로 스택값들을 이동시킨다.

---

24) 커널 source의 include/sched.h에서 task_struct를 참조할것.
25) 이것을 마지막으로 이후 call gate를 set하는 경우는 없다.
즉 version 1.0에서는 set_call_gate 매크로가 한번 호출된다.
26) C함수를 assembly어로 구현하는 것에 대한 관련서적 참조.

<2> stack pointer(SP)[27]를 parameter로 함수 iABI_emulate함수를 호출한다.



<그림2.27>

3. iBCS에 대해

iBCS(Intel Binary Compatibility Standard)는 Intel processor에서 동작하는 binary file의 forma -t에 대한 표준안으로써, 적용되는 object file format으로는 **ELF**(Executable and Linking Format)와 **COFF**(common object file format)등이 있다.[28]

ELF는 ABI(Application Binary Interface)를 위해 USL(UNIX System Laboratories)에 의해 개발, 공개되었다.[29]

2.5.2 device 정보를 위한 변수지정

empty_zero_page(0x5000<head.S>)에 있는 device정보(ROOT_DEV,drive_info,screen_info,mouse유무)를 지정된 변수에 넣는다.

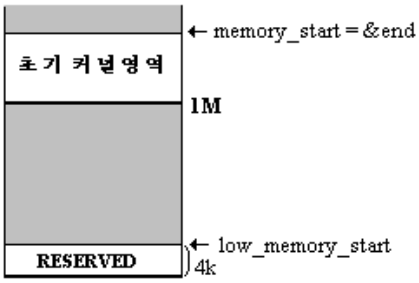
2.5.3 memory 정보를 위한 변수지정

---

27) 그림<2.5.2>는 그림 아래쪽의 번지가 높게 그려져 있다.
28) linux.faq 참조.
29) ftp.intel.com site의 /pub/TIS에서 ELF관련자료 참조.

memory_end : bios에 의해 얻은 전체메모리양.
memory_start : 컴파일된 커널에서 text와 data 영역의 끝.(&end)
low_memory_start : 사용가능 메모리 시작.

kernel영역의 마지막(&end)인 1M를 넘으면 이것은 압축커널임을 말한다. memory_start 부터 device driver를 위해 메모리를 사용하는데, 압축커널의 경우는 메모리에 load된 kernel영역이 끝나는 지점부터가 되며, 비압축 kernel의 경우는 1M가 memory_start가 된다.
low_memory_start는 압축커널의 경우 아래쪽 1M에 비어있는 영역에 대해 차후 memory를 초기화할때 사용가능 영역으로 표시하기 위해 4KB(PAGE_SIZE)[30]의 값이 되며, 비압축커널의 경우는 커널영역이 끝나는 지점부터 사용가능표시를 하기위해 &end의 값이 된다.



<그림2.28> 압축커널의 경우

2.5.4 PAGE ALIGN

page를 align시킨다는 것은 물리메모리에서 PAGE_SIZE인 4K단위로 page를 정렬시키기 위한 것이다. source내에서 메모리를 align시키는 부분이 자주 나타나는데 이는 메모리의 사용을 효율

30) 물리페이지 page0는 공란으로 비어있다. 이곳에는 bios에 의한 interuppt vector가 들어가며, paging_init()소스에 붙은 주석대로라면 386,486 laptop의 BIOS와 SMM(momory module) chip에 의해 사용되고 있다고 한다. 차후 mem_init()에 의해 이 곳은 RESERVED 상태가 되어 process에 의한 직접적인 access가 금지된다.
BIOS에 의한 interuppt vector들은 리눅스에 의해 사용되지 않으며, interuppt descriptor에 의해 interuppt code들이 재지정된다. 그러나 BIOS의 보존은 16bit program의 수행(DOS emulat-ion)을 위해 사용된다.

적으로 하기 위함이다.

2.5.5 initialize page

head.S에서 물리메모리 초기 4M를 위한 page table을 초기화 시킨것을 기억할 것이다. 여기서는 그 page table을 갱신하고 더불어 나머지 물리메모리에 대한 page table들을 만든다. <그림 2.29>에 RAM 8M의 경우를 예로 들었다. 그래서 단지 1개의 page table이 더 요구된다. 선형메모리의 커널 번지와 물리메모리의 커널 번지를 동일시 하기위해 table당 여전히 두개의 dir entry를 가진다.
table entry 속성에서 추가된 a bit는 accessed bit로서 프로세서는 어떤 page에 access할때 이 bit를 set한다. 일반적으로 운영체제는 이 bit에 의해 swapping의 후보가 되어야 할 최근 가장 사용하지 않은 페이지를 구분한다.[31] 물론, 리눅스에서도 이 방법을 사용한다. mm.h내의 PAGE_ACCESSED 를 참고하라.

2.5.6 bus방식 check

system bus 방식이 EISA(Extended ISA)인지를 확인한다.
물리메모리 1M byte바로아래에 4byte의 "EISA"라는 문자열이 발견되면 이는 EISA bus 방식임을 나타내는 것이다.

2.5.7 trap_init

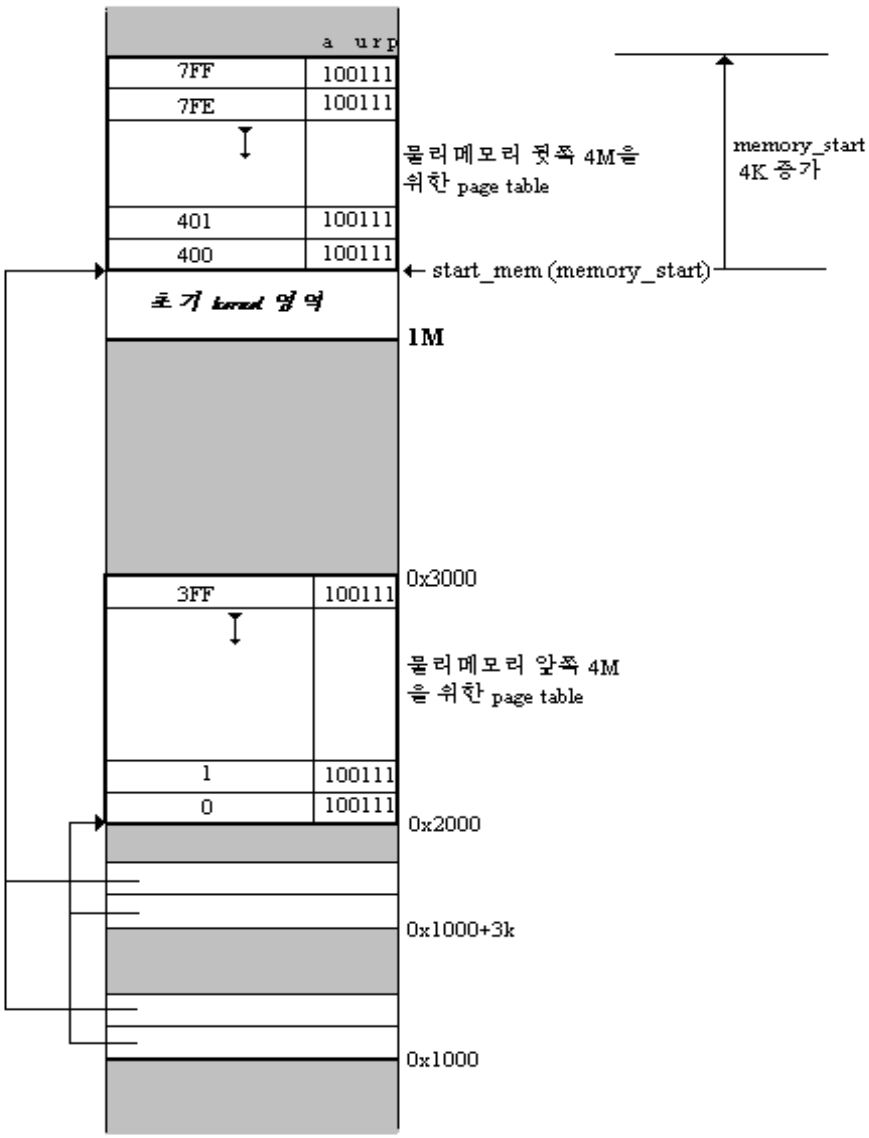
1) 개요.

head.S에서 ignore_int로 초기화시켜둔 IDT에다가 예외처리 vector descriptor를 set한다.
trap descriptor가 idt[0]<head.S>에서 idt[48]까지 set되는데, 실제 trap처리는 idt[0] ~ idt[17]까지이며 나머지는 예약된 상태이다. 앞에서도 밝혔듯이 리눅스에서는 BIOS에서 제공되는 in-terrupt가 아닌 자체에서 만들어진 interrupt descriptor에 의해 예외 처리가 이루어진다.

표<2.5>에서는 BIOS에 의한 예외처리와 리눅스에서 만들어서 사용하는 예외처리를 비교하여 보았다.[32]

---

31) 80386프로그램 입문-교학사 169 page 참조.



<그림2.32>

◆ INT 8 ~ INT 15까지의 내용에서
BIOS란의 내용은 IBM에서 HARAWARE와 BIOS를 위해 만든것.
리눅스란은 보호모드를 위해 INTEL에서 지정한 것.

---

32) **BIOS 와 비고 부분은 프로그래머를 위한 PC SOURCE BOOK-매크로** 에서 발췌.

IRQ들을 다른 INT[num]로 옮기지 않으면 보호모드에서 양쪽 기능은 충돌한다.
리눅스에서는 setup.S에서 이들을 옮겨 충돌을 막는다.

INT 번호	BIOS	LINUX	비 고
0	DIVIDE ERROR	DIVIDE ERROR	
1	SINGLE STEP	DEBUG	
2	NMI	NMI	
3	BREAK POINT	INT3	
4	OVERFLOW	OVERFLOW	
5	PRINT SCREEN	BOUNDS	
6	RESERVED	INVALID_OP	
7	RESERVED	DEVICE_NOT_AVAILABLE	
8	TIME SERVICE	DOUBLE_FAULT	IRQ0 TIMER 0
9	KEYBOARD SERVICE	COPROCESSOR_SEGMENT _OVERUN	IRQ1
10 (0AH)	RESERVED	INVALID_TSS	IRQ2 slave 8259A
11 (0BH)	COM2	SEGMENT_NOT_PRESENT	IRQ3
12 (0CH)	COM1	STACK_SEGMENT	IRQ4
13 (0DH)	HARD DISK SERVICE /PRINTER SERVICE	GENERAL_PROTECTION	IRQ5 PC:DISK ADAPTOR AT:LPT2
14 (0EH)	FLOPPY DISK SERVICE	PAGE_FAULT	IRQ6
15 (0FH)	PRINTER SERVICE	RESERVED	IRQ7 LPT1
16 (10H)	VIDEO I/O	COPROCESSOR_ERROR	
17 (11H)	주변기기 I/O	ALIGNMENT_CHECK	

<표2.5>

2) BIOS와 리눅스 예외처리 비교.[33]

INT 0 : DIV명령에 의한 나누기에러 발생시 호출.
<예> ① 분모 0로 나눌경우.

33) 다음 문헌을 참고하였다.
Ralf Brown의 interrupt list v42.와
80386 programing 입문-교학사, 그리고
The Undocumeted PC-Wesley (가장 많은 내용을 참고하였다.)
Blown의 list는 PC통신등을 통해 구할수 있다.

② 나누었을때 몫이 들어갈 register 크기를 몫이 초과한 경우.
리눅스에서는 SIGFPE signal을 current process에 보낸다.

INT 1 :

① SINGLE STEP
flag에 TF(trap flag)가 set되면 각 명령어 다음에 발생한다.
MS-DOS의 debugger의 T 명령처럼 한개의 명령어가 수행되는 과정을 추적하는데 사용될수 있다.

② DEBUGGING EXCEPTION
386이상 기종에서 사용되며 dubugger program과 관련된 여러가지 경우에 의해 발생.
리눅스에서는 자체에서 정의된 코드 사용하여 trap signal을 프로세서에 보낸다.
(SIGTRAP은 default로 수행중지)
user에 의해 trap handler가 정의되었다면
수행모드가 kernel mode인 경우 db7(debugger register)을 clear후 계속 수행된다.
수행모드가 user mode인 경우는 계속 수행.

INT 2 : Non-Maskable interrupt시 발생.
리눅스에서는 단지 관련 메세지 출력.

INT 3 : 1 byte breakpoint 명령에 의해 발생.(SOFT-ICE등의 debugger에서 사용.)
리눅스에서는 SIGTRAP signal을 current process에 보낸다.
descriptor 작성시 dpl=3이 되어 user process에서 이용 가능.

INT 4 : OF flag가 set되었을때 INTO(On Overflow Call Interrupt Procedure)[34]명령어 수행에 의해 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.
descriptor 작성시 dpl=3이 되어 user process에서 이용 가능.

INT 5 : BOUND 명령어 실행시 경계초과에 의해 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.
descriptor 작성시 dpl=3이 되어 user process에서 이용 가능.

---

34) 386명령어 목록 참고.

◆ 이 예외는 IBM의 BIOS에 의한 기능과 INTEL에서 부여한 기능이 양립하는 첫번째 예외이다.[35]

BIOS에 의한 PRINT SCREEN기능과 INTEL이 부여한 BOUND기능이 있으며, 이는 286이상 기종에만 적용된다.

PRINT SCREEN 기능은 IRQ 1(리눅스에서는 INT 0x21)의 KEYBOARD interrupt handler에 코드를 넣어줌으로써 이 기능을 구현할수 있다.[36]

(kernel 1.0에서는 PRINT SCREEN이 지원되지 않음)

INT 6 : 명령어 op code error에 의해 발생.

리눅스에서는 SIGILL signal을 current process에 보낸다.

INT 7 : coprocessor 부재.

리눅스에서는 SIGSEGV signal을 current process에 보낸다.

INT 8 : Sytem Timer로서 IRQ 0에 연결되어 있다.

이 예외는 또한 **DOUBLE FAULT**에서도 발생한다. 모든 예외가 DOUBLE FAULT를 일으키지는 않는다.

예외 발생중 표<2.6>에 나와있는 것들 중에 또 하나가 발생하면 DOUBLE FAULT를 일으킨다.[37]

INT num	FAULT Description
0	Divide Error
A	Invalid Task State Segment(TSS)
B	Segment not present
C	Stack fault
D	General Protection Fault(GPF)
E	Page Fault

<표2.6>

리눅스에서는 SIGSEGV signal을 current process에 보낸다.

INT 9 : Keyboard로서 IRQ 1에 연결되어 있다.

35) The Undocumented PC-Wesley 217 page참조
36) The Undocumented PC-Wesley 706 page 참조.
37) The Undocumented PC-Wesley Table 7-4

또한 80286/80386 Protected mode에서 coprocessor명령어가 coprocessor에 전달될때, page 또는 segment 침해에 의해 발생한다.
80486에서는 이것이 INT 0Dh로 옮겨졌다.
리눅스에서는 SIGFPE signal을 current process에 보낸다.

INT 10 : PIC 8259A의 cascade에 사용.
286이상 기종에서는 task switch동안에 새로운 TSS가 타당하지 않을때 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.

INT 11 : serial port인 com2,com4로서 IRQ 3와 연결.
또한 286이상기종의 보호모드에서 load된 segment가 시스템에 존재하지 않을때 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.
IBM 하드웨어에 의한 IRQ 4기능과 CPU에 의한 segment not present기능은 충돌하지 않는다. 이것은 시스템에 의해 ISR(In Servie Register)를 조사하는 것에 의해 가능하다.[38]

INT 12 : serial port인 com1,com3로서 IRQ 4와 연결.
또한 286이상기종의 보호모드에서 stack segment 변경시 limit violation에 의해 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.
IBM 하드웨어에 의한 IRQ기능과 CPU에 의한 stack exception기능은 충돌하지 않는다. 이것은 시스템에 의해 ISR(In Servie Register)를 조사하는 것에 의해 가능하다.

INT 13 : hard disk로서 IRQ 5 와 연결.
또한 286이상기종의 보호모드에서 심각한 에러가 CPU에 의해 감지되면 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.

INT 14 : floppy diskette로서 IRQ 6과 연결.
또한 386이상기종에서 paging 시스템의 page access 에러시 발생한다.
리눅스에서는 page falut handler 수행.
INT 15 : printer로서 IRQ 7과 연결.

---

38) The Undocumented PC-Wesley

리눅스에서는 SIGSEGV signal을 current process에 보낸다.

INT 16 : vedio함수에 대한 access 및 coprocessor error에 의해 발생한다.
리눅스에서는 do_coprocessor_error hadler 수행.

INT 17 : BIOS의 POST에 의해 eguipment configuration을 실메모리 40:10h주소에 return. 또한 80486이상기종에서 AC flag를 set하였을 경우 메모리 alignment에러에 의해 발생한다.
리눅스에서는 SIGSEGV signal을 current process에 보낸다.

3) gate

보호모드에서는 특권레벨이 0,1,2,3으로 나누어져 있는데 숫자가 작을수록 특권레벨이 높다. 리눅스에서는 0과 3만을 사용하는데 0은 kernel mode를, 3은 user mode에 적용된다. 특권레벨이 낮은 곳에서 높은곳으로 코드수행이 옮겨가려면 gate를 사용해야 한다. 즉 user mode에서 kernel mode로 접근하기 위해서는 gate를 통과해야 하며 그러기 위해서는 user process와 마찬가지로 gate의 특권레벨도 3이여야 한다.
보호모드에서 제공하는 gate는 다음과 같이 4가지가 있다.

- call gate
- task gate
- trap gate
- interrupt gate

이들은 비슷한 descriptor 형태를 하고 있는데, 구분하는 요소중 가장 특징적인 것은 type이다.
다음은 /include/asm/system.h의 내용일부이다.

```
#define set_intr_gate(n,addr) \
        _set_gate(&idt[n],14,0,addr)

#define set_trap_gate(n,addr) \
        _set_gate(&idt[n],15,0,addr)

#define set_system_gate(n,addr) \
        _set_gate(&idt[n],15,3,addr)
```

```
#define set_call_gate(a,addr) \
        _set_gate(a,12,3,addr)
```

task gate는 LDT나 IDT에 상주하며, TASK전환을 위해 사용될수 있는데 현재 리눅스에서는 이용되고 있지 않다. 리눅스에서 call gate는 LDT에 &default_ldt pointer가 상주한다. user process에 의해 접근이 가능하도록 dpl이 3임을 주목하자. trap gate와 interrupt gate의 차이는 마스크 가능한 interrupt를 제어하는 IF flag에 대한 부분이다. interrupt gate는 IF flag를 disable시킬수 있는데 반해, trap gate는 그렇지 못하다[39]. 따라서 trap handler수행중 다시 trap이 발생할수가 있는 것이다. INT 8의 double fault를 참고하라.
리눅스에서는 trap gate를 interrupt table(_idt영역)을 set하기 위해 사용한다. interrupt table은 이 절에서 다루는 **예외처리**부분과 8장에서 다룰 **system call**을 처리하기 위한 INT 0x80으로 나뉜다. system call의 경우는 user process가 접근하기 위해 항상 dpl=3이며, 예외처리는 **INT 3,4,5**만이 dpl이 3이다. 위의 매크로중 set_system_gate가 이를 위해 사용된다. 리눅스에서 intruppt gate는 IRQ hander를 set하는데 사용된다. IRQ발생처리는 항상 dpl=0인 높은 특권에서만 수행됨을 주목하자.

gate	type	Linux에서의 사용	비 고
call gate	12	**LDT에 상주**	
task gate	5	**미사용**	offset을 사용하지 않는다.
trap gate	15	interrupt table	
interrupt gate	14	IRQ handler	disable IF

<표2.7> Linux에서의 gate 사용

4) trap handler를 set하는 과정.

① trap.c의 DO_ERROR의 매크로가 수행되면서 INT 1,2,14,16을 제외한 나머지 trap handler를 정의한다.
② INT 1,2,14,16은 각각 do_nmi, do_debug, do_page_fault, do_coprocessor_error함수들이 trap handler가 된다.
③ 앞에서 ignore_int로 초기화 했던 head.S의 idt영역의 처음부터 48번째까지를 trap을 위한 interrupt descriptor로 채운다.

39) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 145 page

5) trap handler가 수행되는 과정.

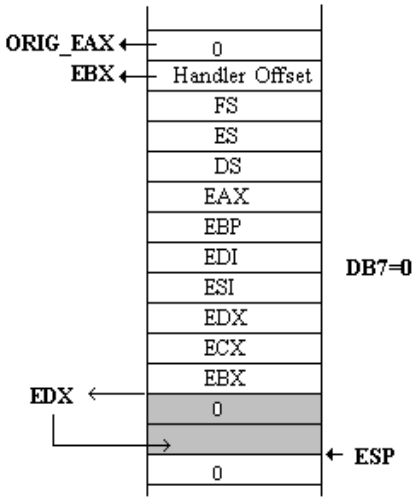
① 예외에는 CPU에 의해 발생하는 error code가 있는 것과 없는 것이 있다.[40]
있는 경우와 없는 경우에 따라 예외 발생시 stack에 들어가는 내용에는 약간 차이가 있다.
표<2.8>은 80386예외 error code의 유무를 나타낸다.[41]

INT 번호	예외 내용	에러코드 유무
0	DIVIDE ERROR	NO
1	DEBUG	NO
3	INT3	NO
4	OVERFLOW	NO
5	BOUND	NO
6	INVALID OP	NO
7	DEVICE NOT AVAIL	NO
8	DOUBLE FAULT	YES(항상 0)
9	SEGMENT OVERRUN	NO
10	INVALID TSS	YES
11	SEGMENT NOT PRESENT	YES
12	STACK EXECPTION	YES
13	GENERAL PROTECTION	YES
14	PAGE FAULT	NO
16	COPROCESSOR ERROR	

<표2.8>

② 표<2.8>에서 error code가 없는 것에 대해서 리눅스에서는 default값으로 0로 둔다. 그래서 handler의 offset을 stack에 push하기 전에 push $0가 선행된다.
결과적으로 그림<2.30>과 같은 stack 상태에서 handler가 call된다. 따라서 c언어로 짜여진 do_page_fault, do_nmi, do_debug,...(trap.c)등의 handler에 넘어오는 **pt_reg구조체 pointer**와 **error code**가 어떻게 이루어지를 알수 있다.

40) 80386 프로그램 입문-교학사 7.4 error code 절을 참조
41) 80386 프로그램 입문-교학사 표 7.3 을 참조.



<그림 2.30> handler call하기 직전의 SP 위치

그림<2.30>에서 0가 들어 간것은 error code가 없는 예외발생의 경우이며, 이 곳에 실제로 CPU에서 보낸 error code가 들어간다.

2.5.8 init_IRQ

1) IRQ를 위한 interrupt handler들을 set하는 과정.

irq.c의 BUILD_IRQ 매크로의 수행에 의해[42] interrupt[16]배열의 IRQ0_interrupt ~ IRQ15_interrupt와, fast_interrupt[16]배열의 fast_IRQ0_interrupt ~ fast_IRQ15_interrupt와, bad_interrupt[16]배열의 bad_IRQ0_interrupt ~ bad_IRQ15_interrupt의 handler가 정의된다.
heas.S에서 IRQ들을 위한 interrupt번호를 INT 0x20 ~ INT 0x2F로 옮겼으므로 이들을 위한 interrupt descriptor들은 _idt에서 0x20번째 부터 들어가게 된다.

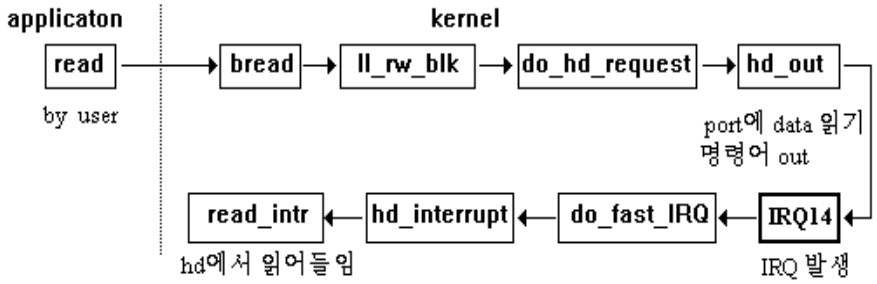
2) include/asm/irq.h의 BUILD_IRQ 매크로에 의해 1)에서 기술한 세 종류의 매크로들이 만들어진다. 이들에 의해 do_IRQ와 fast_do_IRQ함수가 호출된다.
예를 들어 하드디스크 접근에 대한 신호인 IRQ 5가 발생하는 경로는 다음과 같다.

① 일반적으로 어떠한 파일에서 data를 읽어 들일때는 block(1K)단위로 읽는다. 그때 커널 함수인 bread함수가 호출된다.[43]

---

42) BUILD_IRQ 매크로의 정의는 include/asm/irq.h에 있다.

② 계속해서 hd_out()함수가 호출되고, SET_INTR 매크로에 의해 DEVICE_INTR=read_intr 이 되며(blk.h에 의해 정의된 do_hd()라는 함수는 kernel내에 없다.),
hard disk를 제어하는 명령어를 port를 통해 보내게 되고, 이 때 IRQ 14신호가 발생한다.
미리 설명한대로 do_IRQ나 fast_do_IRQ가 호출되고 연이어 hd.c내에 정의된 sigaction구조체에 의해 sa->sa_handler인 hd_interrupt()가 호출된다.
DEVICE_INTR인 read_intr()이라는 함수가 호출되어 파일에서 데이타를 읽는다.



<그림2.31> hard disk에서 읽어들이는 경로.

3) ACK_FIRST 와 UNBLK_FIRST는 8259A의 master chip에 대한 것이며, ACK_SECOND 와 UNBLK_SECOND는 8259A의 slave chip에 대한 것이다.
ACK_##chip은 irq신호를 받았다는(Acknowledge)신호로서 해당 irq를 disable시키고, EOI(End of Interrupt)를 발(issue)한다. EOI는 hardware interrupt service routine의 끝에서 발하는 것으로, 인터럽트 컨트롤러(PIC)가 추가 인터럽트를 받아들이도록 한다. slave에 대해서는 먼저 포트 A0h에 EOI를 발하고,차례로 포트 20h에 EOI를 발한다.[44]
EOI를 발하는 것은 포트 0x20와 0xA0에 0x20를 out함으로써 이루어진다.

```
mov 0x20,al
out al,0x20
out al,0xAo
```

UNBLK_##chip은 irq에 대한 handler가 수행되었으므로 다시 해당irq를 enable시킨다.

4) fast_IRQ_interrupt와 IRQ_interrupt, bad_IRQ_interrupt의 차이

① bad_IRQ_interrupt는 원하지 않는(리눅스가 필요한 irq가 아닌경우) irq신호가 발생했을 경우 해당 irq를 ACK_##chip에 의해 disable시키고 irq에 대한 handler또한 준비되어 있지

43) 차후 우리가 file을 읽을때 사용하는 *read* 함수를 설명할때 자세히 다룬다.
44) The Undocumented PC 823, 827 page

않으므로 다른 작업은 하지 않는다.
init_IRQ() 코드의 첫부분에서 모든 irq들에 대해 default로 일단은 이것을 set시킨다.

② fast_IRQ_interrupt와 bad_IRQ_interrupt는 초기 레지스터값을 저장하기 위해 SAVE_MOST를 사용하며, IRQ_interrupt에서는 SAVE_ALL을 사용한다.
IRQ_interrupt는 handler에 대한 call이 있은 후, system call시와 동일하게 ret_from_syscall routine로 들어가서 signal check가 이루어진후, irq service routine이 끝난다. 따라서 ret_from_syscall 코드에서 요구하는 모든 레지스터가 save된다. 그러나 fast_IRQ_interrupt는 handler를 call한 후 바로 끝나기 때문에 c언어로 짜여진 handler에 필요한 레지스터만을 save한다.

③ IRQ_interrupt는 sti명령어에 의해 handler동작중에도 interrupt가 걸리게 된다.

④ <표2.9> 에는 IRQ_interrupt와 fast_IRQ_interrupt을 사용하는 디바이스에 대해 요약되어 있다.
빠른 처리속도가 요구되는 hard disk,serial line등은 fast_IRQ_interrupt을 사용하며, 상대적으로 처리속도가 덜 요구되는 keyboard,timer등은 IRQ_interrupt을 사용한다.

fast_IRQ_interrupt	IRQ_interrupt
hard disk	keyboard
CD Rom	mouse
serial line	timer
floppy	coprocessor
line printer	

<표2.9>

병렬프린터는 기본적으로 polling방식이지만 interrupt를 지원하는 카드를 사용하는 경우는 fast_IRQ_interrupt을 사용한다.[45] library 함수인 ioctl(lp_ioctl()로 이어진다)에서 IRQSET cmd(parameter 명령)에 의해 irq신호에 의한 interrupt방식 프린터가 사용가능하다.

5) irqaction함수와 request_irq함수.

45) GNU kernel hacker guide

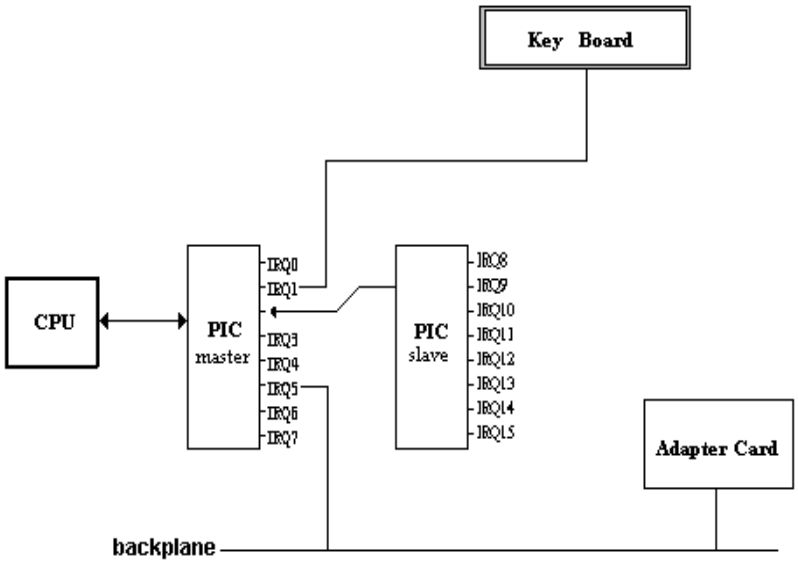
① request_irq 함수는 irq 번호와 handler를 parameter로 받아서 sigaction struct의 field 값들을 set한다.
sa_flag는 SA_INTERRUPT인 경우 해당 irq의 service routine이 fast_IRQ_interrupt가 된다. sa_mask(1:enable)는 해당 irq가 enable상태인지를 나타낸다.
request_irq 에서는 sa_flag를 0으로 하기 때문에, 이 함수에 의해 irq를 enable시키면 이 irq의 service routine은 IRQ_interrupt가 된다.
sigaction struct field값들을 채운후 irqaction함수를 호출한다.

② irqaction 함수는 irq 번호와 sigaction 구조체를 받아서 해당 irq를 enable시키다.
sa_mask를 1로 set하여 이 irq가 enable상태임을 명시하고, sa_flag에 의해 적당한 irq ser-vice routine을 정하며, irq를 enable시킨다.

6) IRQ 2

IRQ 2는 <그림 2.32>에서 보는 바와 같이 master에서 slave와 연결을 위해 사용된다.
이 IRQ 2를 **cascade**라고 부르기도 한다.
IRQ 2 신호는 사용되지 않으므로 ignore_IRQ struct의 **no_action**을 handler로 set한다.

7) 주변기기와 PIC와 CPU의 연결.



<그림2.32> PIC와 keyboard의 연결

① keyboard는 IRQ1과 연결되어 있다.

우리가 keyboard의 key를 누르면 IRQ1신호가 PIC에 전달되고 PIC는 IRQ1을 INT 0x21로 바꾸어 CPU에게 전달되며, CPU는 이 INT 0x21을 사용하여 kenel내의 irq handler를 수행한다.

리눅스에서 irq를 INT 0x21에 지정하지 않았다면, 물론 INT 0x09의 인터럽트 벡터에 의해 BIOS에서 제공하는 interrupt handler가 수행될것이다.

IRQ 1신호는 우리가 key입력시 key가 눌려졌을때와 key를 놓을때, 즉 2번 전달된다.[46]

② 많은 irq가 특정 하드웨어(타이머,키보드,하드디스크등),주변기기에 물려있다.

그러나 irq중에는 adapter card(병렬 port나 net card등)와 연결이 가능하도록 motherboard의 backplane상에 놓여있는 경우도 있다. IRQ 5가 그 대표적인 예이다.[47]

---

46) 보다 자세한 내용은 11장에서 keyboard handler와 함께 설명한다.

47) The Undocumented PC의 812 page에는 IRQ 5 신호에 의한 PIC와 CPU의 동작이 상세히 기술되어있다.

2.5.9 sched_init (kernel/sched.c)

여기서는 초기화 과정만을 설명한다. 이들에 대한 자세한 논의는 6장 프로세서 관리 에서 다룰것다.

1) timer를 위한 bottom half routine set.(timer_bh함수)

2) first task(task[0] or init_task) descriptor set.
이 task는 INIT_TASK 초기값(sched.h)으로 초기화된다.

```
struct task_struct init_task = INIT_TASK;
```

init task[48]는 수행중인 process가 없을때, 수행되기 위해 필요하다.
start_kernel(init/main.c) 코드의 뒷부분에서 init process를 fork하고 난후 kernel은 무한 loop 를 돌게 된다. 이 무한 loop가 init task이다.

```
for(;;)
    idle();
```

이 task는 sys_idle(mm/swap.c)에 해당된다.

```
asmlinkage int sys_idle(void)
{
        need_resched = 1;
        return 0;
}
```

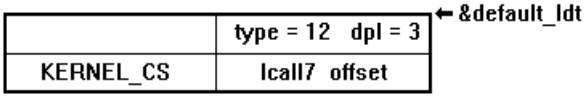
수행중인 process가 없으면 init task는 새로운 process가 생길때까지 kernel은 계속해서 sche -duler를 호출한다. init task는 잠자지도 않으며, 정지시킬수도, 종료시킬수도 없다. 실제 pro -cess가 아니라, kernel내의 routine이기 때문이다.
init_task와 scheduler에 대해서는 6장 프로세서 관리에서 더 논의된다.

---

48) init task는 **swapper**라고도 불리운다. 그러나 Linux에서는 swapping을 위해 init task가 하는 일은 없다. 이러한 명칭은 단지 역사적인 이유에 근거한다고 한다. (hacker' guide)
Linux에서는 init task를 위한 page directory명을 **swapper_pg_dir**로 두었다.

3) first local(for task[0]) descriptor set.

2.5.1절에서 논의한 다음 descriptor가 set된다. ldt로서 NULL이 들어갈 자리에는 이것이 들어가게 된다.



<그림2.36>

앞에서도 언급했듯이, task에서 이것이 호출되면 별다른 동작을 하지 않는다.

4) set_system_gate

8장 시스템 콜 에서 설명된다.

5) 나머지 127개 task를 위한 tss, ldt가 0로 초기화된다.

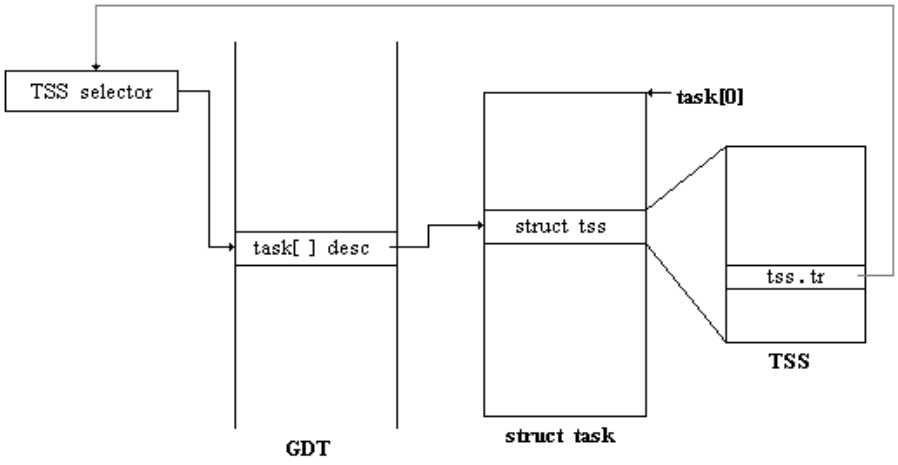
```
typedef struct desc_struct {
        unsigned long a, b
      } desc_table[256];
struct desc_struct *p;
```

시스템이 동시에 수행할수 있는 최대 task수는 128개(NR_TASK)이다.

00	00	00	00	
00		00		entry 6
00	00	00	00	
00		00		entry 7
task [0]				TSS 0
default_ldt				LTD 0
00	00	00	00	이하는 모두
00		00		0로 초기화
00	00	00	00	
00		00		

<표2.10> task[0]의 TSS와 LDT

그림<2.34>는 tss descriptor selector, tss descripor, TSS와의 관계를 나타내었다. 점선은 같은(equal)것임을 나타낸다. 즉 tss.tr에는 TSS selector가 들어간다.



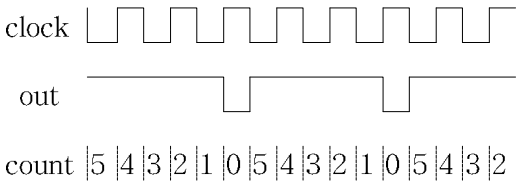
<그림2.34>

task[0]의 descriptor는 task_struct가 아닌, **tss_struct**를 가리킨다. 결국 TSS selector에 의해 TSS에 접근할수 있다. task전환시 process는 task descriptor에 의해 tss 내용을 갱신(save or load)할수 있다.
task[0]->tss.tr에는 exec에 의해 process가 기동할때 TSS descriptor를 가리키는 TSS select-or값이 들어감을 기억하자.

6) 최초의 process인 task[0]의 TSS와 LDT를 tr(task register), ldtr에 각각 load한다.

7) 8254 timer를 초기화한다.

100Hz(HZ=100)정도의 주기로 timer interrupt가 발생하는데 이때 수행되는 interrupt handler는 do_timer(sched.c)이다.
system timer 8254에서 제공하는 모드는 6가지의 mode가 있다.[49] 이중 무한히 연속적으로 CPU interrupt를 발생시키는 mode는 mode 2, mode 3이다. 리눅스는 mode 2를 사용하며, MINIX의 경우는 mode 3을 사용한다. system programmer가 사용하는 방법에 있어서 두 mode는 큰 차이가 없다. 여기서는 mode 2에 대해 설명한다.
system timer는 1.1932MHZ주기로 clock이 발생한다. 우리는 clock주기를 이용하여 interrupt pulse를 발생시키는 횟수를 programming할수 있다. 이것은 1193200을 적당한 값으로 나누어 줌으로써 이루어지는데, 그 값을 count라고 부른다.
예를 들어, count가 6이라면 6번의 clock 주기마다 pulse가 발생한다. count값은 5에서 시작하여 계속 줄어가다가 0이 되면 pulse를 발생시키고 다시 count는 6이 된다.



pulse가 발생하면 timer interrupt handler가 수행된다. 리눅스는 count값이 11932(LATCH)로서 HZ=100주기로 interrupt가 걸리도록한다.

```
outb_p(LATCH & 0xff , 0x40);    /* LSB */
outb(LATCH >> 8 , 0x40);        /* MSB */
```

49) The Undocumented PC Chapter 16. SYSTEM TIMER 참조.

8) BSD(binary coded decimal)와 binary
data를 메모리에 저장하는 방식에는 BSD와 binary두가지 방식이 있다.
binary방식은 16진수로 data를 저장시키는 것을 말하고, BSD는 10진수로 저장시키는 것을 말한다. BSD는 다시 packed와 unpacked라는 두가지 방식이 있다. 다음 그림을 참고하자.[50]
1193200이라는 숫자를 저장하는 것을 예로 들었다.

106	105	104	103	102	101	100	←byte address
'0'	'0'	'2'	'3'	'9'	'1'	'1'	

< Unpacked BSD로 저장 >

unpacked방식에서는 문자(string)형태로 저장된다.

104	103	102	101	100	←byte address
0	2 0	9	3	1 1	

< packed BSD로 저장 >

packed방식에서는 binary와 비슷하지만, 0x0A ~ 0F는 사용하지 않는다.

보통은 binary를 사용하지만, BSD를 사용하는 program들이 있다.
대표적인 예는 CMOS RAM에 data를 저장하는 방식이다. 축전지에 의해 유지되는 CMOS RAM에는 hard disk,memory,time등에 관한 system정보가 들어 있다.
time_init(kernel/time.c) 코드를 참조하자. 이 함수에서는 kernel내에서의 시간관리를 위해 부팅시 CMOS RAM의 BSD값들을 읽어 binary로 바꾸어 가지게 된다.

9) timer mode설정에서 16 bit mode란 count를 사용하는데 있어 8 bit를 사용하느냐, 16 bit를 사용하느냐를 나타낸다. 8 bit를 사용하는 경우는 아랫 byte만을 사용하는 LSB(least significant byte) mode와 윗 byte만을 사용하는 MSB(most significant byte) mode가 있다. 사용하지 않는 나머지 1 byte는 port에 count를 써 넣을때 0로 한다.

10) timer interrupt handler를 **do_timer함수**로서 설정한다.

---

50) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 18page

2.5.10 parse_options

LILO로 부팅시 "boot:" 프롬프트 다음의 command를 분석(parsing)한후 환경변수로 setup한다.

2.5.11 kmalloc_init

위의 구조체는 malloc의 모든 page앞에 메모리할당정보로 들어가게 된다.

```
struct page_descriptor {
        struct page_descriptor *next;
        struct block_header *firstfree;
        int order;
        int nfree;
};
```

kmalloc(size,priority)호출에서는 할당이 요구되는 size에 따라 다른 block크기를 사용하게 된다. 이는 메모리의 효율적인 사용을 위한 것이다. size가 1000이라면 sizes[5]에 해당하는 blocksize인 1020byte가 할당될 것이다. kmalloc초기화에서는 위의 sizes 배열들이 (size × nblocks + page_descripor크기) 1 page크기(4096byte)보다 작은지 확인한다. kmalloc 코드는 7장 메모리 관리에서 다룬다.
kmalloc_init은 start_memory에는 영향을 미치지 않는다.

```
struct size_descriptor {
        struct page_descriptor *firstfree;
        int size;
        int nblocks;

        int nmallocs;
        int nfrees;
        int nbytesmalloced;
        int npages;
};

struct size_descriptor sizes[] = {              /*  size × nblocks  */
        { NULL,  32,127, 0,0,0,0 },             /*     4064         */
        { NULL,  64, 63, 0,0,0,0 },             /*     4032         */
        { NULL, 128, 31, 0,0,0,0 },             /*     3968         */
```

```
        { NULL, 252, 16, 0,0,0,0 },          /*     4032         */
        { NULL, 508,  8, 0,0,0,0 },          /*     4064         */
        { NULL,1020,  4, 0,0,0,0 },          /*     4080         */
        { NULL,2040,  2, 0,0,0,0 },          /*     4080         */
        { NULL,4080,  1, 0,0,0,0 },          /*     4080         */
        { NULL,   0,  0, 0,0,0,0 }           /*        0         */
};
```

2.5.12 chr_dev_init

문자(character) device들을 초기화한다.

1) register_chrdev (fs/devices.c)

메모리 device를 등록한다.

```
#define MEM_MAJOR    1      /* major.h  */

int register_chrdev(unsigned int major, const char * name, struct file_operations
                    *fops)
{
     if (major >= MAX_CHRDEV)
             return -EINVAL;
     if (chrdevs[major].fops)
             return -EBUSY;
     chrdevs[major].name = name;   /*  major=1, name= "mem"  */
     chrdevs[major].fops = fops;   /*  fops = memory_fops  */
     return 0;
}

static struct file_operations memory_fops = {
        NULL,                /* lseek */
        NULL,                /* read */
        NULL,                /* write */
        NULL,                /* readdir */
        NULL,                /* select */
        NULL,                /* ioctl */
```

```
        NULL,                /* mmap */
        memory_open, /* just a selector for the real open */
        NULL,                /* release */
        NULL         /* fsync */
};
```

메모리 device file(/dev/mem, /dev/ram, /dev/null등)들을 open할려고 하면 memory_open함수가 기동하여 각각에 대한 file operation함수들을 제공한다.

2) · tty_init terminal 초기화
· con_init console 초기화
· rs_init RS-232C terminal관련 초기화.
12장 터미날 시스템에서 다룬다.

3) kbd_init
11장 키보드 시스템에서 다룬다.

4) lp_init
line printer 초기화, 10.3절에서 다룬다.

2.5.13 blk_dev_init

block device들을 초기화한다.
block device는 문자 device와는 달리 buffer cache를 사용한다. 그래서 읽기, 쓰기를 할때 먼저 buffer를 대상으로 하게된다. 만약에 어느 block device에서 읽을려고 하면, 그 내용이 buffer에 있는지를 확인한 후, 없으면 그때서야 device에서 읽는다. 읽은 내용은 buffer에 먼저 담기게 된다. 이렇게 buffer에서 요구되는 내용이 buffer cache에 있는지의 여부에 따라 실제 device를 access하는 것을 다루는 코드를 *strategy routine*이라고 한다.
그래서, block device는 원하는 위치의 data를 대상으로도 읽고 쓰기가 가능한 반면, 문자device는 data가 일렬로 움직이는 형태여서, 시간적으로 앞선 data에 대해 우선적으로 작업이 이루어질 수 밖에 없다. 이렇게 작업대상 data의 순서가 시간의 제약을 받는 것을 동기적(syncronous)이라고 하며, 그렇지 않은 block device는 비동기적(asyncronous)이라고 한다. block device는 사용될 것이라 예상되는 data를 buffer에 미리 올려놓을수 있다.(breada함수)
그리고, filesystem은 block device에만 놓일수 있다.

1) request구조체 배열 초기화.

buffer가 아닌 실제device를 대상으로 작업을 할 것을 요구하는 것을 *request*라고 한다.

```
static struct request all_requests[NR_REQUEST];
```

read only device에 대해서는 다음 배열에 해당 bit가 set된다.

```
static long ro_bits[MAX_BLKDEV][8];
```

다음은 request구조체와 read only bit를 초기화하는 부분이다.

```
req = all_requests + NR_REQUEST;
while (--req >= all_requests) {
        req->dev = -1;
        req->next = NULL;
}
memset(ro_bits,0,sizeof(ro_bits));
```

2) hd_init[51]

문자 device를 등록하는 것과 같은 방법으로 이루어진다.

```
int register_blkdev(unsigned int major, const char * name, struct file_operations *fops)
{
        if (major >= MAX_BLKDEV)
                return -EINVAL;
        if (blkdevs[major].fops)
                return -EBUSY;
        blkdevs[major].name = name;
        blkdevs[major].fops = fops;
        return 0;
}
```

나머지 관련함수등록에 대한 내용은 10장 디바이스 드라이버에서 논의된다.

---

51) 본서에서는 block device들중 hard disk에 대한 것만 다룬다.

2.5.14 calibrate_delay

1초에 빼기(dec명령)를 몇번할수 있느냐로 processor의 속도를 잰다. 다음에 나와있는 __delay함수를 보자.(delay.h) loops값을 1씩 감소시키며 looping을 계속한다.

```
extern __inline__ void __delay(int loops)
{
        __asm__(".align 2,0x90\n1:\tdecl %0\n\tjns 1b": :"a" (loops):"ax");
}
```

loops값을 1부터 시작해서 2배씩(loops_per_sec << 1) 증가시켜가며, delay하는데 1초이상 걸릴때까지 계속한다. 1초이상 (ticks >= HZ) 걸리기 위해서는 loops_per_sec값이 얼마가 되어야 하는지를 구한다음, 다음 공식[52]에 의해, 1초가 걸리기 위해 요구되는 loops_per_sec값을 구한다.

$$\text{1초동안의 loop예상치} = \frac{\text{loops\_per\_sec} \times \text{HZ}}{\text{ticks}}$$

이 값을 500000으로 나눈 것이 *BogoMips*[53]이다.

BogoMips는 리눅스를 개발한 Linus씨가 창안한 processor속도를 비교하기 위한 단위이다. 보다 자세한 내용과 여러 processor들의 BogoMips값을 알기위해서는 bogomips.howto를 참고하기 바란다.

2.5.15 inode_init (fs/inode.c), file_table_init (fs/filetable.c)

이들에 대해서는 5장 파일시스템에서 논의된다.

2.5.16 mem_init

memory map을 만든다. 이 곳은 해당 page가 사용중인지, 자유로운지, 그리고 시스템을 위해 보존(reserved)되어 있는지를 명기한다.

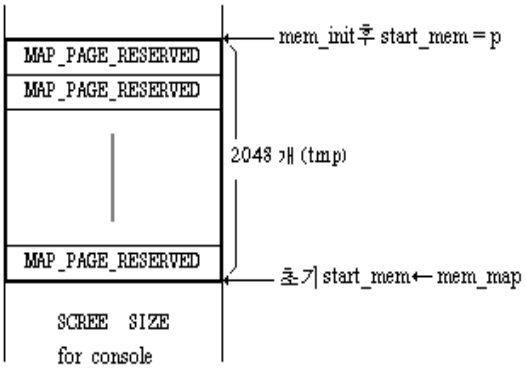
1) high_memory(메모리 끝 : 시스템에 설치된 RAM크기, 편의상 8M를 예로 한다.)을 page 단위

52) loops_per_sec : ticks = 우리가 구할려는 값 : HZ
53) BogoMips : Bogus Millions of instruction per second

로 mask하고, start_mem을 **16 byte**단위로 align.

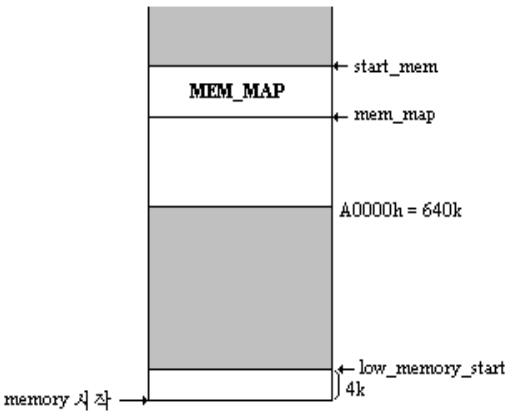
2) tmp는 2 kbyte (page갯수)

3) 일단 모두를 reserved상태(MAP_PAGE_RESERVED)로 초기화한다. 이후 kernel 부분만이 res-erved상태를 유지한다. reserved상태에 있는 page는 user process에게 강탈되어질수 없으며, swapping되지도 않는다.



<그림2.35>

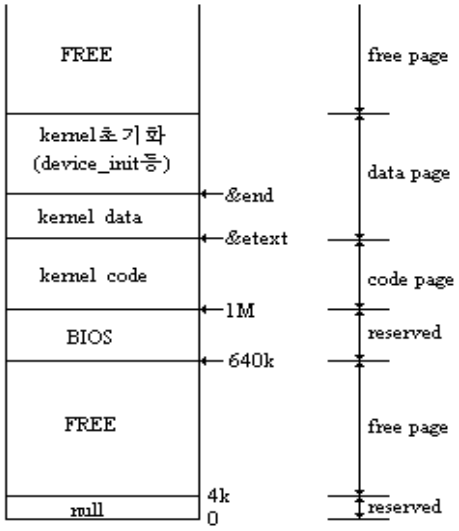
4) 자유롭게 사용가능한 부분을 **free상태**라는 표식으로 mem_map[]내에서 0로 둔다. 그림<2.36>에서 빗금친 부분은 free상태.



<그림2.36> free page

5) code page, data page, reserved page, free page의 수를 display.

이들에 의한 메모리 점유는 그림<2.37>과 같다



<그림 2.37> mem_map초기화

free page의 경우, 모든 free page의 첫 4byte는 다음 free page의 주소[54]를 가리킨다. free_

page_list는 가장 높은 주소의 free page(초기에는 주소가 8M-4k인 page)를 가리킨다. 그래서 free page를 사용할때는 가장 높은 주소(초기의 경우)의 free page가 먼저 사용되고, free_pa -ge_list는 다음 page인 8M-8K번지를 가리키게 된다.(*get_free_page*함수 참조.)

6) WP(Write protect in supervisor mode) bit

eflag의 WP bit는 486에서 지원되는 bit로서, 486에서 이 bit가 1이면, READ ONLY상태인 page를 설령 kernel이 쓰기작업을 할려고 해도 page fault가 발생한다. 그래서, page fault handler가 수행되고, handler는 *wp_works_ok*값을 1로 만든다. 80386에는 이 bit가 항상 0이다. 그래서 386에서는 user mode에서만 write proctect가 수행된다.

---

54) free page가 연달아 있을경우에는 높은 주소의 page의 첫 4byte에 바로 다음 낮은 주소의 page 주소가 들어간다. 즉,0x2000주소에는 0x1000값이, 0x3000주소에는 0x2000값이 들어있다.

2.5.17 buffer_init

4장. 버퍼 캐쉬에서 설명한다.

2.5.18 time_init

여기서는 kernel이 관리할 현재 시각을 CMOS memory를 통해 얻는다.

PC에는 앞에서 설명한 system timer 8255 chip외에도 MC146818A chip으로 이루어진 Real Time Clock(**RTC**)이 있다. CMOS memory에는 여러가지 system data가 기록되어 있는데, CMOS chip 내부에 현재시간을 유지하는 RTC를 포함하고 있다.[55] RTC는 전원이 꺼진 상태에서는 축전지에 의해 현재시간이 유지된다. 모든 RTC 레지스터들은 BCD형(format)을 사용한다.

RTC는 매초 시각이 바뀔때마다 레지스터의 시각값을 갱신하는데, 그 동안 상태레지스터 A의 bit 7을 set한다. 그래서 상태레지스터 A의 bit 7이 set되어 있는동안, user가 시각을 읽을려고 하면, 엉뚱한 값을 읽을수 있다. bit 7은 약 2 microsecond가량 set후 다시 non-set상태가 되는데, kernel은 이 시각을 피해서 현재시각을 읽는다. set에서 non-set로 바뀌는 바로 그 순간부터 읽기 시작한다.

아래에 이탤릭체로 된 부분도 위와 같은 목적으로 사용되는 것으로 정확한 시각을 얻기위한 두번째 방법이다. 1초 이내에 시각을 2번 읽어 동일하면 정확한 값으로 인정된다. 그러나 kernel은 bit 7 을 사용하는 것으로 정확성을 보장받기를 기대하고 있다.

```
for (i = 0 ; i < 1000000 ; i++) /* may take up to 1 second... */
        if (CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP)
                break;              /* set   */
for (i = 0 ; i < 1000000 ; i++) /* must try at least 2.228 ms*/
        if (!(CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP))
                break;            /* non-set */
do { /* Isn't this overkill ? UIP above should guarantee consistency */

            /* 시각 읽어들임 */
} while (time.sec != CMOS_READ(RTC_SECONDS));
```

55) THE UNDOCUMENTED PC-FRANK VAN GULLUWE 724 page

```
                /*  BCD를  BIN으로 */
        /*  읽은 시각을 timeval 구조체에 넣는다. */
        xtime.tv_sec = kernel_mktime(&time);
```

2.5.19 floppy_init

```
void floppy_init(void)
{
1)      outb(current_DOR,FD_DOR);
2)      if (register_blkdev(MAJOR_NR,"fd",&floppy_fops)) {
                printk("Unable to get major %d for floppy\n",MAJOR_NR);
                return;
        }
3)      blk_size[MAJOR_NR] = floppy_sizes;
        blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
        timer_table[FLOPPY_TIMER].fn = floppy_shutdown;
        timer_active &= ~(1 << FLOPPY_TIMER);
4)      config_types();
        if (irqaction(FLOPPY_IRQ,&floppy_sigaction))
                printk("Unable to grab IRQ%d for the floppy driver\n", FLOPPY_IRQ);
5)      if (request_dma(FLOPPY_DMA))
                printk("Unable to grab DMA%d for the floppy driver\n", FLOPPY_DMA);
        /* Try to determine the floppy controller type */
6)      DEVICE_INTR = ignore_interrupt; /* don't ask ... */
7)      output_byte(FD_VERSION);        /* get FDC version code */
        if (result() != 1) {
                printk(DEVICE_NAME ": FDC failed to return version byte\n");
                fdc_version = FDC_TYPE_STD;
        } else
                fdc_version = reply_buffer[0];
        if (fdc_version != FDC_TYPE_STD)
                printk(DEVICE_NAME ": FDC version 0x%x\n", fdc_version);
#ifndef FDC_FIFO_UNTESTED
        fdc_version = FDC_TYPE_STD;     /* force std fdc type; can't test other. */
#endif

        /* Not all FDCs seem to be able to handle the version command
         * properly, so force a reset for the standard FDC clones,
```

```
         * to avoid interrupt garbage.
         */

8)       if (fdc_version == FDC_TYPE_STD) {
                 initial_reset_flag = 1;
                 reset_floppy();
         }
}
```

1) 0x3f2(DOR)은 diskette controller register로서, DMA[56]를 사용가능상태(enable)로 set한다. FDC(floppy driver controller)를 사용가능상태로 set한다.

2) floppy file operation 함수들 등록.

3) floppy size의 default는 1.44M(MAX_DISK_SIZE).
request함수(do_fd_request) 등록.
timer함수 등록. timer_active변수에서 FLOPPY_TIMER bit 초기화(mask off,**disable**). floppy timer는 나중에 필요하면 그 때 기동시킨다.

4) CMOS memory에서 설치된 floppy driver들의 type(1.2M or 1.44M)에 대한 정보를 얻는다.

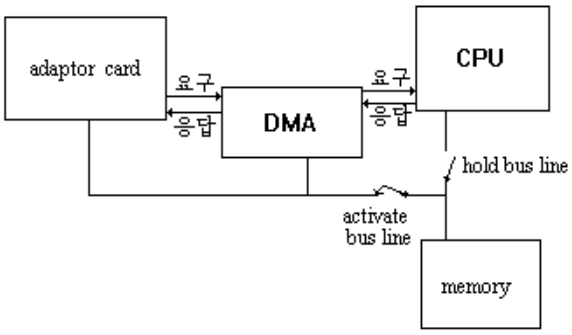
5) request DMA(Direct Memory Access)

8237 DMA chip은 adaptor card와 memory사이에 CPU의 간섭없이 직접 data를 전송할수 있도록 하기위해 사용된다. 만약에 DMA가 없다면, adaptor card에서 memory로 data를 전송하기 위해서는 CPU가 card로 전송되어온 data를 CPU내의 register에 담았다가, 다시 memory로 보내야 할 것이다. 그러나 DMA에 의한 전송은 DMA가 쓰여질 memory의 주소만으로 card와 memory사이에 line을 활성화(activate)함으로써 card에서 memory로 data가 직접 전송된다. 이것은 card가 DMA를 사용할수 있도록 초기설정을 해줌으로써 가능하다.(setup_DMA 함수 <drivers/block/floppy.c>)
그렇게 함으로써, memory에 data를 전송하기 위해 adaptor card는 DMA에게 전송요구를 하게되고, DMA는 CPU에게 memory bus를 사용하지 말것을 요구한다. 이 후 CPU의 응답(hold line)과 DMA에 의한 memory line 및 I/O line 활성화, 그리고 card에게 data를 전송해도 좋다는 응답이 차례로 이어지고, data전송이 이루어 진다.[57]

---

56) DMA에 대해서는 10장 디바이스 드라이버 에서 논의된다.

DMA의 사용은 card와 memory사이에 data전송을 전담하는 processor(DMA를 말함)를 둠으로써, 그 시간에 CPU는 다른 작업을 할수 있도록 한다.
리눅스에서는 DMA를 사용하는 floppy controller뿐만 아니라, network card, sound card등 많은 외부 card들이 DMA의 효과를 이용하고 있다. 단, hard disk는 DMA를 사용하지 않고 있다.
DMA는 7개의 channel을 사용할수 있다.(channel은 8개지만 PIC처럼 4번 channel은 cascade이다.) kernel은 2번 channel을 floppy controller를 위해 사용한다. dma_chan_busy[] table에 1을 set하여, floppy controller를 위해 DMA를 사용할 것임을 표시한다.



<그림2.38> DMA 전송에 의한 bus line 상태

6) DIVICE_INTR 변수에 대해서는 10장 디바이스 드라이버 에서 설명한다.

7) floppy driver controller가 표준형인지를 확인한다.

8) 표준형일 경우 reset_floppy .
다음 코드는 register값을 set하는 동안 delay를 위한 것이다.

```
for(i=0 ; i<1000 ; i++)

    __asm__("nop");
```

2.5.20 sock_init

57) DMA가 동작하는 보다 자세한 사항에 대해서는
THE UNDOCUMENTED PC-FRANK VAN GULLUWE 844 page를 참고하라.

이 부분은 network를 위한 socket을 초기화 시키는 부분으로, 본서에서 network에 대해서는 다루지 않는다.

2.6 Coprocessor 초기화

이 절에서는 system에 coprocessor가 있는 경우의 초기화에 대해 논의한다.
리눅스가 수행되기 위해서는 coprocessor가 없는 경우에는 CR 0의 **EM**(math emulation)[58] bit를 set해 주어야 한다.

coprocessor error발생시 제공되는 handler에는 INTEL에서 제공하는 INT 10h(이것이 **예외 16**에 해당한다)와 MotherBoard에서 제공하는 IRQ 13(DOS에서는 INT 75h, 리눅스에서는 INT 2Dh)이 있다. 일반적으로 제공되는 MotherBoard에서는 error발생시 INT 10h이 수행되지는 않는다.[59] 그러나 간혹 INTEL이 제공한 것을 따르는 경우가 있다.[60] 그래서, 예외 16 handler가 수행되는지를 확인한다.
리눅스에서 INT 10h handler와 IRQ 13 handler는 다음과 같다. IRQ 13 handler는 init_IRQ(kern-el/irq.c)에서 math_error_irq함수로 초기화 되었다.

```
static void math_error_irq(int cpl)
{
        outb(0,0xF0);    /* clear busy flag */
        if (ignore_irq13)
                return;
        math_error();
}
```

<IRQ 13 handler kernel/irq.c>

58) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 248 page
59) The Undocumented PC-Wesley 222, 236 page
60) 필자의 시스템이 그러하다. 정말로 예외 16을 사용하지 않는 것이 일반적인 경우인가?

```
asmlinkage void do_coprocessor_error(struct pt_regs * regs, long error_code)
{
      ignore_irq13 = 1;
      math_error();
}
```

< INT10h handler kernel/trap.c >

어느 handler가 먼저 수행되는지를 알기 위해 fdiv명령어를 사용하여 0으로 1을 나누어 의도적으로 math error를 발생시킨다.

```
__asm__("fldz ; fld1 ; fdiv %st,%st(1) ; fwait");
```

fldz와 fld1은 80x87 stack에 0와 1을 차례로 넣는다. %st는 stack내의 1을 %st(1)은 0를 가리킨다.

INT 10이 수행된 후에도 다시 irq 13이 다시 발생하면 뒤에 발생한 irq 13은 무시된다. 코드 위에 붙여놓은 주석이 흥미롭다.

```
/*
 * check if exception 16 works correctly.. This is truly evil
 * code: it disables the high 8 interrupts to make sure that
 * the irq13 doesn't happen. But as this will lead to a lockup
 * if no exception16 arrives, it depends on the fact that the
 * high 8 interrupts will be re-enabled by the next timer tick.
 * So the irq13 will happen eventually, but the exception 16
 * should get there first..
 */
```

2.7 리눅스 version display

다음은 tools/version.h의 내용이다.

```
#define UTS_RELEASE "1.0"
#define UTS_VERSION "#1 Fri Jul 8 21:15:20 GMT 1994"
#define LINUX_COMPILE_TIME "21:15:21"
#define LINUX_COMPILE_BY "root"
```

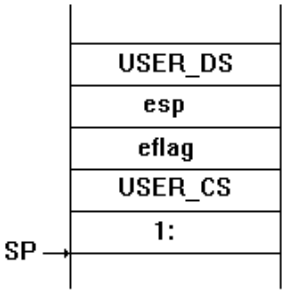
```
#define LINUX_COMPILE_HOST "darkstar"
#define LINUX_COMPILE_DOMAIN "frop.org"
```

2.8 move to user mode

다음은 system.h에 나와 있는 kernel mode에서 user mode로 이동하기 위한 routine이다.

```
#define move_to_user_mode() \
__asm__ __volatile__ ("movl %%esp,%%eax\n\t" \
        "pushl %0\n\t" \
        "pushl %%eax\n\t" \
        "pushfl\n\t" \
        "pushl %1\n\t" \
        "pushl $1f\n\t" \
        "iret\n" \
        "1:\tmovl %0,%%eax\n\t" \
        "mov %%ax,%%ds\n\t" \
        "mov %%ax,%%es\n\t" \
        "mov %%ax,%%fs\n\t" \
        "mov %%ax,%%gs" \
        : /* no outputs */ :"i" (USER_DS), "i" (USER_CS):"ax")
```

코드의 1~ 6 번째 줄까지에 의해 stack은 그림<2.39>과 같은 형태가 된다.



<그림2.39> user mode로 들어가기 직전 stack

현재 1:는 kernel 선형번지인 0xC0100000 + 1: 인 지점을 말한다. 이 시점에서 iret를 실행시키면 user선형번지인 *0x100000 + 1:* 인 곳으로 jump한다.

head.s의 14)setup paging에서 다음 내용을 기억할 것이다.

> 현재 kernel이 실메모리 0x100000에 있으므로 paging후 kernel은 선형번지 0xc0100000에 있는 것이 된다. 그러나 kernel은 여전히 선형번지 0x100000에 있기도 하다.

user mode로 이동후 user data segment(USER_DS)를 각 data용 segment들(ds,es,gs,fs)에 넣는다. 이제부터 data는 user 선형공간에 들어가게 된다.

여기서 fs는 user와 kernel을 연결시켜주는 중요한 의미를 가지는 segment로 사용된다. system call을 통해 kernel mode로 들어갈때 fs에는 USER_DS descriptor의 selector가 들어간다. 그래서 kernel이 user mode의 data를 kernel mode로 가져올때는 fs를 사용한다. get_fs_byte(user공간에서 1byte를 읽어 온다), get_fs_word, get_fs_long등과 같은 asm/segment.h의 inline 코드가 그 예이다.

3장

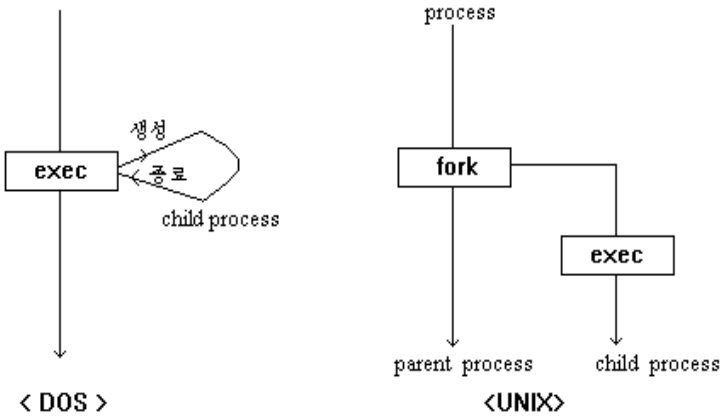
부팅과정 후반부

부팅과정 후반부의 kernel mode에서 user mode로 진행되는 부분을 다룬다.

3.1 fork system call

fork system call의 코드는 **7장 메모리 관리**에서 다루며, 이번 절에서는 부팅과정에서 fork() 함수 호출에 따른 결과가 논의된다.

fork는 UNIX가 multi-processing을 할수 있도록 만들어주는 system call로서 fork가 수행되면 부모 process의 image를 물려받은 자식(child) process가 생성된다. 이 때 fork()를 호출한 process는 부모(parent) process라고 부른다. fork후에 자식 process와 부모 process의 코드수행은 fork함수가 return하는 값에 따라 달라진다. fork함수는 자식 process에게는 0를, 부모 process에게는 자식 process의 PID를 return한다.
DOS에서 exec함수를 사용하여 자식 process를 수행시킬수 있다. 그러나, DOS에서는 자식 process가 수행을 끝낼때까지 부모 process가 정지된채 기다리고 있어야 한다. 물론 UNIX에서도 이것은 가능하다. 그러나 UNIX에서는 일반적으로 자식 process를 생성하기 위해서 fork를 먼저 호출한다. 그렇게 하여야 이 후 exec system call로 새로운 자식 process가 될 program을 수행시킴으로써 부모 process와 자식 process가 **동시에** 수행되어질수 있는 것이다.
그림 <3.1>은 DOS와 UNIX의 부모 process와 child process의 관계를 나타낸다.



<그림 3.1> child process

PC에서 microprocessor(CPU)는 1개뿐이므로, process가 동시에 수행된다는 의미는 각 process들이 CPU를 돌아가면서 점유한다는 것을 의미한다. 이를 전산용어로 시분할(time division)방식이

라고 한다. 리눅스에서는 약 3/20초 간격으로 process의 CPU점유가 있다.[61] 즉, 한 process는 이 시간동안만 CPU를 점유하고는 다른 process에게 넘겨주고 다시 자기 차례가 올때까지 기다려야 한다.
3/20초와 같은 process의 CPU점유주기를 **timeslice**라고 부른다. 시분할은 특별히 interrupt등이 발생하지 않는한 timeslice의 주기로 호출되는 *scheduler*에 의해 이루어진다. 호출된 scheduler는 어떤 process에게 CPU를 **재할당**할지를 결정한다. scheduler에 대한 보다 자세한 사항은 6장 프로세서 관리 에서 다룬다.

3.2 init process

1) *idle* system call

fork에 실시한 부모 process는 idle() loop를 수행한다. 이 때 idle loop는 수행중인 다른 proce-ss가 없을때 수행된다.

```
asmlinkage int sys_idle(void)
  {
     need_resched = 1;  /* scheduler 호출  */
     return 0;
  }
```

위 코드에서처럼 idle system call은 단지 scheduler를 호출한다. need_resched변수값이 set되면 system call수행시 scheduler가 호출된다. idle system call에 대해서는 6장 프로세서 관리 에서 자세히 다룬다.

2) *setup* system call

자식 process는 fork후 곧 setup system call을 수행한다. setup system call은 system에 설치된 partition분할을 포함한 hard disk에 대한 정보를 drive_info_struct(init/main.c)라는 구조체에 넣는다.

```
struct drive_info_struct { char dummy[32]; } drive_info;
```

setup system call은 단지 kernel부팅시에만 사용된다. 코드에 대한 논의는 9장 리눅스 파티션 에서 다룬다.

---

61) MINIX는 1초에 10번, 1/10초가 한번에 process에게 할당되는 시간이다.

3) argv_init, envp_init

"TERM=con%dx%d"는 init daemon의 argument인 envp_init으로 사용된다. argv_init과 envp_init은 main.c내의 parse_option에 의해 구해짐을 기억할 것이다.
참고로 TERM환경변수는 joe, vi, setterm 등에 의해 사용된다. 만약 TERM 환경변수로 set되어진 것이 /etc/termcap entry에 없으면 이 program들은 정상적으로 수행되지 않는다.

4) *dup* system call

/dev/tty1을 초기 console(리눅스는 가상 console을 사용한다.)을 위해 open한다. 이 때 file descriptor로서 0(STD_IN)이 할당된다. 왜냐하면, 처음 file을 open하면 file descriptor 0번이 open된 file을 위해 할당되기 때문이다. 이 후 open에 의해서는 file descriptor 1,2,3,...번이 차례로 할당될 것이다. tty1을 STD_OUT, STD_ERROR로도 사용할려면 *dup* system call을 사용해야 한다. dup는 STD_IN의 file descriptor를 비어있는 다음 file descriptor에게 복사한다. 그래서 file descriptor 1, 그 다음에는 2가 차례로 할당되고, 이들이 각각 STD_OUT, STD_ERROR에 해당되며, STD_IN file descriptor가 가지고 있는 정보와 똑 같은 내용을 갖게 된다.

5) init daemon[62)]

argv_init와 envp_init을 argument로 init daemon을 수행시키는 것으로 kernel부분의 부팅과정은 끝이 난다. init daemon이 발견되지 않으면 곧 shell이 수행된다.

init daemon은 /etc/inittab에 있는 line들을 수행시킨다. slackware에서 사용하고 있는 init program은 System V 양식(style)의 init이다.
다음은 slackware 1.0에 포함된 **inittab**이다.

```
#
# inittab       This file describes how the INIT process should set up
#               the system in a certain run-level.
#
# Version:      @(#)inittab             2.04    17/05/93        MvS
#
# Author:       Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#
```

62) daemon이란 부팅시 background로 수행되어 system이 shutdown될때까지 system의 동작을 감시하는 program을 말한다.
Advanced Programming in the Unix Environment 13. Daemon Process

```
# Default runlevel.
id:5:initdefault:                                    ①

# System initialization (runs when system boots).
si:S:sysinit:/etc/rc.d/rc.S                          ②

# Script to run when going single user.
su:S:wait:/etc/rc.d/rc.K                             ③

# Script to run when going multi user.
rc:123456:wait:/etc/rc.d/rc.M                        ④

# What to do at the "Three Finger Salute".
ca::ctrlaltdel:/sbin/shutdown -t3 -rf now            ⑤

# What to do when power fails (shutdown to single user).
pf::powerfail:/sbin/shutdown -f +5 "THE POWER IS FAILING"        ⑥

# If power is back before shutdown, cancel the running shutdown.
pg:0123456:powerokwait:/sbin/shutdown -c "THE POWER IS BACK"     ⑦

# If power comes back in single user mode, return to multi user mode.
ps:S:powerokwait:/sbin/init 2                        ⑧

# The getties in multi user mode on consoles an serial lines.
#
# NOTE NOTE NOTE adjust this to your getty or you will not be
#                 able to login !!
#
c1:12345:respawn:/sbin/getty tty1 9600               ⑨
c2:12345:respawn:/sbin/getty tty2 9600
c3:45:respawn:/sbin/getty    tty3 9600
c4:45:respawn:/sbin/getty    tty4 9600
c5:45:respawn:/sbin/getty    tty5 9600
c6:456:respawn:/sbin/getty    tty6 9600              ⑩

# Serial lines
#s1:45:respawn:/sbin/getty ttyS0 9600                ⑪
#s2:45:respawn:/sbin/getty ttyS1 9600
```

```
# Dialup lines
#s1:45:respawn:/sbin/uugetty -h ttyS0 9600
#s2:45:respawn:/sbin/uugetty -h ttyS1 9600

# Runlevel 6 used to be for an X-window only system, until we discovered
# that it throws init into a loop that keeps your load avg at least 1 all
# the time. Thus, there is now one getty opened on tty6. Hopefully no one
# will notice. ;^)
# It might not be bad to have one text console anyway, in case something
# happens to X.
x1:6:wait:/etc/rc.d/rc.6          ⑫

# End of /etc/inittab
```

inittab의 line에 붙은 번호는 수행되는 과정이라고 보아도 된다.

① init daemon이 수행되는 level이 5임을 말한다. 그래서 line의 runlevel값중에 5 가 있는 것들이 수행된다. /etc/initlevel의 값을 5로 갱신한다.
② , ③ 은 비록 runlevel 값에 5는 없지만 수행된다.
rc.K는 rc.S내에서 수행된다.
④ rc.M에서는 rc.S, rc.K와 마찬가지로 부팅과정에서 기본 system 및 network등과 관련된daem -on이 수행된다. (**cron** daemon이나, remote host에서 rlogin를 통한 접속을 해 왔을때 수행되는 **rlogind** server 등..)
⑤ warm booting시 수행된다.
⑥ , ⑦ , ⑧ 이 line들은 UPS(Uninterruptible Power Supply)가 장착되어 있는 system에 해당되는 내용이다. 전원이 OFF되면 그것을 감시하는 program이 SIGPWR[63] signal을 init proces -s 에게 보낸다. 그 때 상황에 따라 이들이 수행되는데 kernel이 이 signal에 대해 특별히 하는 것은 없다.
init 2는 off된 전원이 user mode에서 on되었을때 수행되는데, /etc/initlevel의 내용을 2로 갱신하고, SIGPWR handler가 수행되면서 inittab를 다시 읽는다.
⑨ 가상 console들을 위한 **getty** daemon이 수행된다. getty는 첫번째 argument인 /etc/tty*을 두번째 argument인 속도와 함께 open한다.
rc.S,rc.M등은 단지 *exec* system call에 의해 수행되지만, getty는 *fork*와 함께 exec에 의해 수행된다.

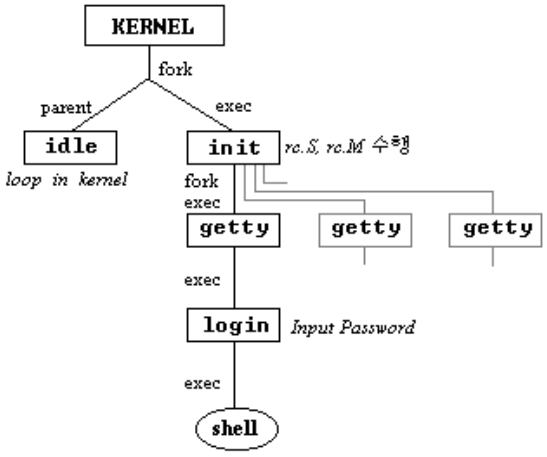
⑪ serial line terminal을 위한 open.

63) Advanced Programming in the Unix Environment 268 page

⑩ , ⑫ X-window가 수행되면 runlevel이 6이 된다. 이 때 tty6은 text console로써 사용된다. rc.6은 display manager인 xdm을 수행시킨다.

init daemon은 이 외에도 process수행정보를 가지는 /etc/utmp화일을 관리한다.

다음 그림은 부팅과정을 나타내었다. 특히 fork, exec부분에 주의하자.



<그림 3.2> fork와 exec에 의한 부팅 과정

6) getty daemon

getty daemon은 /dev/tty*와 /dev/ttyS*를 open system call을 이용하여 개방한다.
tty*는 가상 console을 위한 것이지만, ttyS*는 serial line terminal을 위한 것이다. 따라서op-en시키는 방법에도 차이가 있다. 리눅스에서 console의 경우는 console화면출력을 위해 me-morymap을 이용한다. 즉, 각 가상 console마다 screen크기의 vedio memory를 보유하고 각자의 session process[64]가 수행되면 그 memory에 출력한다. 그러다가 user가 Alt-function key에 의해 어느 한 가상 console의 화면을 보기를 원하면 자기 memory의 내용이 출력화면으로 나가게 된다. 따라서, port를 통한 작업을 할 필요가 없다. 그러나 serial line teminal의 경우는 port를 통해서만 teminal과 대화(읽기, 쓰기)할수 있으므로 모든 ttyS*는 자신의 port를 가진다.
port번호는 rs_table 구조체(drivers/char/serial.c)에 나와 있는데 다음이 그 내용이다.

```
struct async_struct rs_table[] = {
        /* UART CLK   PORT  IRQ     FLAGS        */
```

64) 가상 console에서 수행중인 forground process를 말한다.

```
        { BASE_BAUD, 0x3F8, 4, STD_COM_FLAGS },         /* ttyS0 */
        { BASE_BAUD, 0x2F8, 3, STD_COM_FLAGS },         /* ttyS1 */
        { BASE_BAUD, 0x3E8, 4, STD_COM_FLAGS },         /* ttyS2 */
        { BASE_BAUD, 0x2E8, 3, STD_COM4_FLAGS },        /* ttyS3 */

        { BASE_BAUD, 0x1A0, 9, FOURPORT_FLAGS },        /* ttyS4 */
        { BASE_BAUD, 0x1A8, 9, FOURPORT_FLAGS },        /* ttyS5 */
        { BASE_BAUD, 0x1B0, 9, FOURPORT_FLAGS },        /* ttyS6 */
        { BASE_BAUD, 0x1B8, 9, FOURPORT_FLAGS },        /* ttyS7 */

        { BASE_BAUD, 0x2A0, 5, FOURPORT_FLAGS },        /* ttyS8 */
        { BASE_BAUD, 0x2A8, 5, FOURPORT_FLAGS },        /* ttyS9 */
        { BASE_BAUD, 0x2B0, 5, FOURPORT_FLAGS },        /* ttyS10 */
        { BASE_BAUD, 0x2B8, 5, FOURPORT_FLAGS },        /* ttyS11 */

        { BASE_BAUD, 0x330, 4, ACCENT_FLAGS },          /* ttyS12 */
        { BASE_BAUD, 0x338, 4, ACCENT_FLAGS },          /* ttyS13 */
        { BASE_BAUD, 0x000, 0, 0 },     /* ttyS14 (spare; user configurable) */
        { BASE_BAUD, 0x000, 0, 0 },     /* ttyS15 (spare; user configurable) */

        { BASE_BAUD, 0x100, 12, BOCA_FLAGS },   /* ttyS16 */
        { BASE_BAUD, 0x108, 12, BOCA_FLAGS },   /* ttyS17 */
        { BASE_BAUD, 0x110, 12, BOCA_FLAGS },   /* ttyS18 */
        { BASE_BAUD, 0x118, 12, BOCA_FLAGS },   /* ttyS19 */
        { BASE_BAUD, 0x120, 12, BOCA_FLAGS },   /* ttyS20 */
        { BASE_BAUD, 0x128, 12, BOCA_FLAGS },   /* ttyS21 */
        { BASE_BAUD, 0x130, 12, BOCA_FLAGS },   /* ttyS22 */
        { BASE_BAUD, 0x138, 12, BOCA_FLAGS },   /* ttyS23 */
        { BASE_BAUD, 0x140, 12, BOCA_FLAGS },   /* ttyS24 */
        { BASE_BAUD, 0x148, 12, BOCA_FLAGS },   /* ttyS25 */
        { BASE_BAUD, 0x150, 12, BOCA_FLAGS },   /* ttyS26 */
        { BASE_BAUD, 0x158, 12, BOCA_FLAGS },   /* ttyS27 */
        { BASE_BAUD, 0x160, 12, BOCA_FLAGS },   /* ttyS28 */
        { BASE_BAUD, 0x168, 12, BOCA_FLAGS },   /* ttyS29 */
        { BASE_BAUD, 0x170, 12, BOCA_FLAGS },   /* ttyS30 */
        { BASE_BAUD, 0x178, 12, BOCA_FLAGS },   /* ttyS31 */

/* You can have up to four HUB6's in the system, but I've only
 * included two cards here for a total of twelve ports.
```

```
 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(0,0) },  /* ttyS32 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(0,1) },  /* ttyS33 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(0,2) },  /* ttyS34 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(0,3) },  /* ttyS35 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(0,4) },  /* ttyS36 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(0,5) },  /* ttyS37 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(1,0) },  /* ttyS32 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(1,1) },  /* ttyS33 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(1,2) },  /* ttyS34 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(1,3) },  /* ttyS35 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(1,4) },  /* ttyS36 */
        { BASE_BAUD, 0x302, 3, HUB6_FLAGS, C_P(1,5) },  /* ttyS37 */
};
```

getty는 login process를 실행시킨다.

7) login process

login process는 getty에 의해 terminal이 open된 후 곧 수행된다. login은 "login:"과 "Password: "[65] 를 출력하고, user의 입력을 받아 /etc/passwd의 내용과 일치하는지 확인후, shell을 수행시킨다. shell이 명령입력 prompt를 표시하면, 부팅절차가 끝난다.

8) shell

shell이란 명령해석기로써, DOS에서는 COMMAND.COM이 이에 해당한다. shell은 **/etc/profile** 및 home directory의 **.profile**을 읽어들여 환경설정을 한다.
리눅스에서 제공하는 shell로는 가장 전통적인(그러나, 기능은 막강하다) BASH(Bourne shell), TCSH(C shell), KSH(Korn shell)등이 있다. 본인생각으로는, 주석이 많이 붙어있는 BASH가 분석하기에 용이하다. BASH는 사용자들의 다양한 욕구를 충족시키기 위해 많은 사람들에 의해 손질되어 지금에 이르렀다. 친절한 주석은 아마도 사용자가 자신의 취향대로 개정하여 사용할수 있도록 배려한 것이라 여겨진다.
BASH의 명령해석을 위한 부분은 *bison*이라는 컴파일러 제작도구를 이용하였다. bison은 전통적인 컴파일러 제작도구인 yacc와 비슷하다. yacc를 위한 source는 bison에서도 사용될수(compatible)있다. 컴파일러 제작도구에는 그 외에도 lex, flex등이 있다. 그들에 대해서는 man-ual page를 참고하라.
컴파일러 제작도구와 관련된 책으로 두 권을 소개한다.

---

65) linux에서 password를 생성시키는 함수는 library에서 제공하는 *crypt*함수이다.

· UNIX를 이용한 컴파일러 설계 - 김상욱 홍릉과학출판사
· lex & yacc - O'Reilly & Associates

4장

버퍼 캐쉬

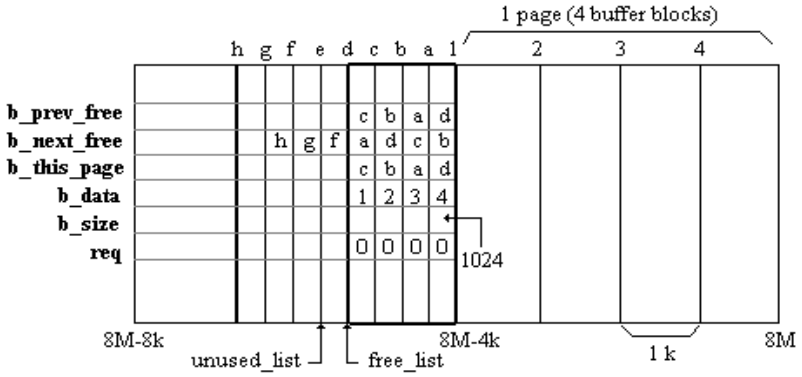
4.1 buffer_init

1) 4M이상이면 min_free_page를 200으로 잡는다. 200개만큼은 buffer로 사용하지 않는다.

2) hash_table을 0로 초기화한다.

3) grow_buffer
buffer를 확보한다. 먼저 get_free_page로 page를 2개 얻어서 그림<4.1>과 같이 초기화 한다.

얻은 2개의 page는 data가 들어갈 실제 buffer를 위한 page와 각 buffer에 대한 정보가 들어갈 **buffer head**를 위한 page로 구분된다.
1 page는 4개의 buffer(1 block 크기)로 나눈다. block size는 system마다 차이가 있을수 있으나, 리눅스에서는 1 kbyte를 사용한다. buffer head에는 해당 buffer의 정보와 다른 buffer와의 연결 관계를 나타내는 정보를 가지고 있다. buffer head의 크기는 48byte로서 1page에 85개가 들어갈 수 있다.
buffer head 구조체는 그림<4.1>과 같다.



<그림4.1> buffer초기화 상태

get_free_page시 초기에는 메모리의 뒤편에 있는 free page를 먼저 가진다.

```
struct buffer_head {
        char * b_data;                  /* pointer to data block (1024 bytes) */
        unsigned long b_size;           /* block size */
        unsigned long b_blocknr;        /* block number */
        dev_t b_dev;                    /* device (0 = free) */
        unsigned short b_count;         /* users using this block */
        unsigned char b_uptodate;
        unsigned char b_dirt;           /* 0-clean,1-dirty */
        unsigned char b_lock;           /* 0 - ok, 1 -locked */
        unsigned char b_req;            /* 0 if the buffer has been invalidated */
        struct wait_queue * b_wait;
        struct buffer_head * b_prev;            /* doubly linked list of hash-queue */
        struct buffer_head * b_next;
        struct buffer_head * b_prev_free;       /* doubly linked list of buffers */
        struct buffer_head * b_next_free;
        struct buffer_head * b_this_page;       /* circular list of buffers in one page */
        struct buffer_head * b_reqnext;         /* request queue */
};
```

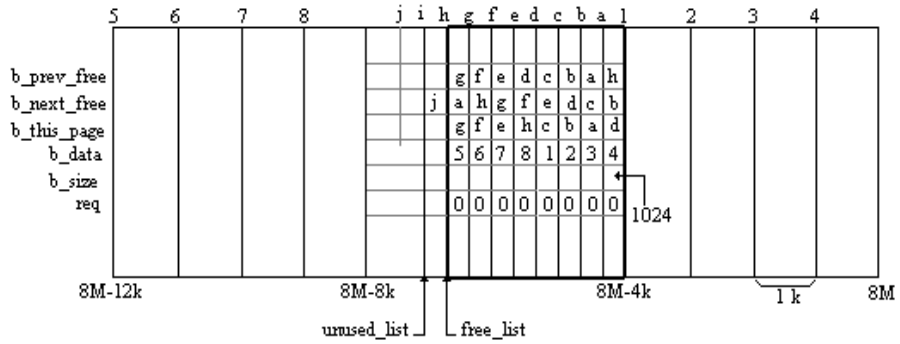
각 항목에 대해서는 이 후에 실제 코드와 함께 논의된다. 여기서는 그림<4.1>에 나타나 있는 것들만 살펴보자.

b_data는 해당 buffer의 위치pointer값을 가지고 있다. b_size는 buffer크기를 나타내며, **b_prev_free**와 **b_next_free**는 고리 형태로 연결되어, 사용중이 아닌 다음 buffer를 나타낸다. 만들어진 buffer를 사용할때는 free_list에서 부터 시작하여 b_next_free로 가면서 빈 buffer를 찾는다. **free_list**는 빈 buffer 고리의 머리에 해당한다. b_this_page는 1 page내의 buffer들만을 고리로 연결한다.(그림<4.2>참조) 어떤 page의 buffer를 해방시킬때 이 pointer 고리를 사용한다. (create_buffers함수의 마지막 부분 참조)

buffer가 필요하면 kernel은 getblk함수(fs/buffer.c)를 사용한다. getblk함수내에서는 grow_buffer-s 함수에 의해 buffer block 4개(1 page)가 더 만들어진다.

그림<4.2>는 buffer 초기화 이후 getblk에 의해 buffer가 더 만들어졌을때의 buffer구조이다.

unused_list는 사용되지 않은 buffer head 영역의 머리에 해당한다. 위에서 8개가 사용되었으므로 아직 85-8개가 남아 있다. 이들은 unused_list에서 시작하여 b_next_free로 가면서 연결된다. 이는 고리형태는 아니며, 단지 다음 사용중이 아닌 buffer head를 가리키게 된다. 여기서 buffer 영역을 가지고 있는(b_data값을 가지고 있는) buffer head의 b_next_free는 결국 buffer block들을 연결시키는 결과를 이루지만, 사용중이 아닌 buffer head의 b_next_free는 buffer head만을 연



<그림4.2> buffer block 4개 증가

결시켜줌을 알수 있다.

getblk에 의해 buffer를 더 할당할때 unused_list가 사용되며, buffer header(b_data값이 없는)들이 다 사용되면, get_more_buffer_head함수에 의해 page를 획득하여 buffer head를 더 만든다.

4.2 getblk

이 함수는 kernel이 buffer가 필요할때 사용한다. getblk에 parameter로 전달되는 dev와 blk는 실제 device(hard disk,cd-rom,floopy등 block device)와 그 device의 어떤 위치를 의미한다. 즉, block(blk)값이 0 이면 disk(예를 들어) 처음의 2개 sector(512byte)를 가리킨다.

dev와 blk에 의해 지정된 1개의 disk block에 buffer block 1개를 할당한다. 이 후에는 이 disk block에 쓰기 작업을 하면 할당된 buffer block에 먼저 쓰여지고, 나중에 sync작업에 의해 실제의 물리device(disk block)에 쓰여진다.

처음 buffer를 생성했을때는 free_list에만 있지만, 생성된 한 block을 *getblk*에 의해 특정 device block에 할당되면, 그 buffer block은 hash table이라는 곳에도 있게 된다. 이 hash table이 바로 우리가 흔히 말하는 buffer cache이다. getblk는 보통은 스스로 *get_free_page*함수를 사용하여 buffer를 만들어 device block에 할당한다.

getblk가 호출되면, 먼저 hash table에 해당 device block에 할당된 buffer block이 존재하는지 check한다. hash table에 있다는 말은 이전에 getblk에 의해 그 device block로 할당해서 사용중이던 buffer block이 있었음을 의미한다. 그때는 free_list의 새로운 buffer를 사용하지 않고, 그 buffer block을 그대로 사용하면 된다. 다음 getblk 초기코드를 보자.

```
bh = get_hash_table(dev, block, size);
if (bh) {
        if (bh->b_uptodate && !bh->b_dirt)
                put_last_free(bh);
        return bh;
}
```

이탤릭체로 된 부분은 반드시 필요하지는 않지만, 전략적인 차원에서 이루어진 것이다.
kernel은 disk에서 buffer에 data를 읽어들일때나, buffer에 있는 data를 disk에 쓸때[66], b_uptodate를 set한다.(end_request(1)[67]에 의해<blk.h>) 그리고 buffer에 있는 내용을 수정할때는 b_dirt를 set한다. b_uptodate가 set되어 있다는 것은 이 data가 유효함을 말한다. 이 값이 set되어 있으면, 이 disk에서 이 buffer로 읽기작업을 할수 없다.(bread<fs/buffer.c>함수 참조) buffer의 data를 무효화시키는 invalidate_buffers(buffer.c)함수에 의해 b_uptodate와 b_dirt는 0로 다시 초기화된다. invalidate_buffers함수는 floppy가 바뀌었을 경우에 buffer의 내용이 다른 floppy에 저장되는 것을 방지하기 위해 호출되는 check_disk_change(buffer.c)함수내에서 사용된다.
b_uptodate나 b_dirt가 set된 경우에도 kernel은 buffer의 내용을 수정할수 있다. 예를 들어, file에 쓰기 작업을 하면, 먼저 buffer에 쓰여진다. 이 때 b_dirt값이 set된다.(이 때 b_uptodate도 set됨) b_dirt는 buffer block의 내용과 disk block의 내용이 다르므로 sync작업이 요구됨을 의미한다. 어떤 devic -e block에 할당된 buffer block을 다른 device block에 할당할려고 할때 b_dirt가 set되어 있으면, sync_buffers함수에 의해 sync작업이 먼저 이루어진다.
윗 코드에 따르면, hash table에 존재하는 buffer block에 있는 data가 유효하면, 그리고 disk block과 buffer block의 내용이 동일하면, free_list의 마지막에 둔다. 일반적으로 최근에 사용된 buffer block은 free_list의 마지막에 두게 된다. 그렇게 해야만 오래전에 사용된 buffer block이 다른 block에 할당된 가능성이 높아지고, 최근에 사용된 것은 할당된 device lock과의 관계를 오랫동안 유지할수 있게 된다.[68]

이 후의 코드는 hash table에서 원하는 device block에 해당하는 buffer block이 없는 경우에 대한 것이다. grow_buffers에 의해 buffer를 더 만들어 우리가 원하는 device block에 할당하고 (b_dev,b_blocknr에 값을 넣음으로써) hash table에도 넣는다.
hash table은 다음 매크로에 의해 이루어진다.

```
#define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
#define hash(dev,block) hash_table[_hashfn(dev,block)]
```

그런데 한개의 hash table 배열에는 여러개의 device block에 각각 해당하는 buffer block들이 들어갈수 있음을 알수있다. 예를 들어, hash_table[301]에는 dev=300,block=1인 것뿐만 아니라 dev=300, block=998에 해당하는 것도 들어 갈수 있다. hash_table은(즉 hash(dev,block)매크로는) type이 **buffer header구조체**이다.

```
struct buffer_head  hash_table[NR_HASH]
```

---

66) 이것을 sync라고 한다.
67) device driver와 함께 논의될 것이다.
68) code에서 b_dirt가 set되어 있을 경우 free list의 끝으로 보내지 않는것은 sync작업이 조속히 이루어질수 있도록 하기 위한 것으로 보인다. Linus!! I am right?

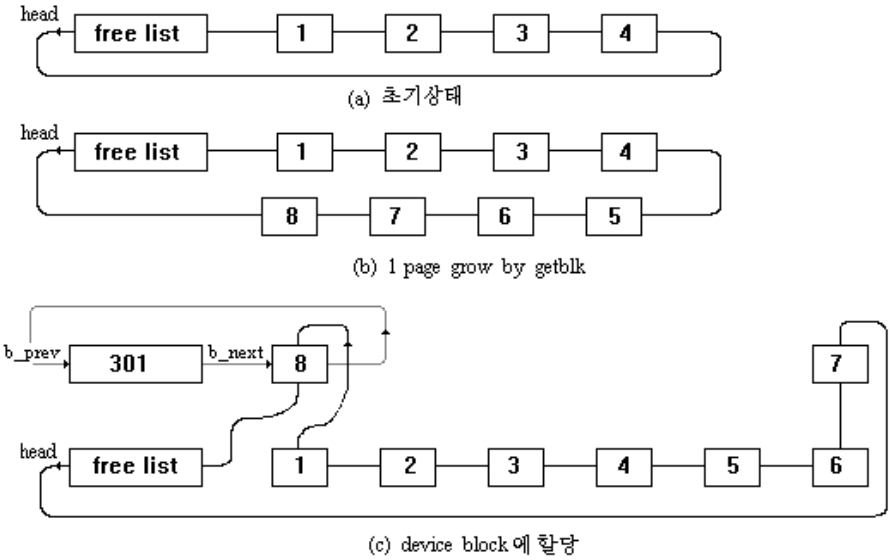
```
#define  NR_HASH  997
```

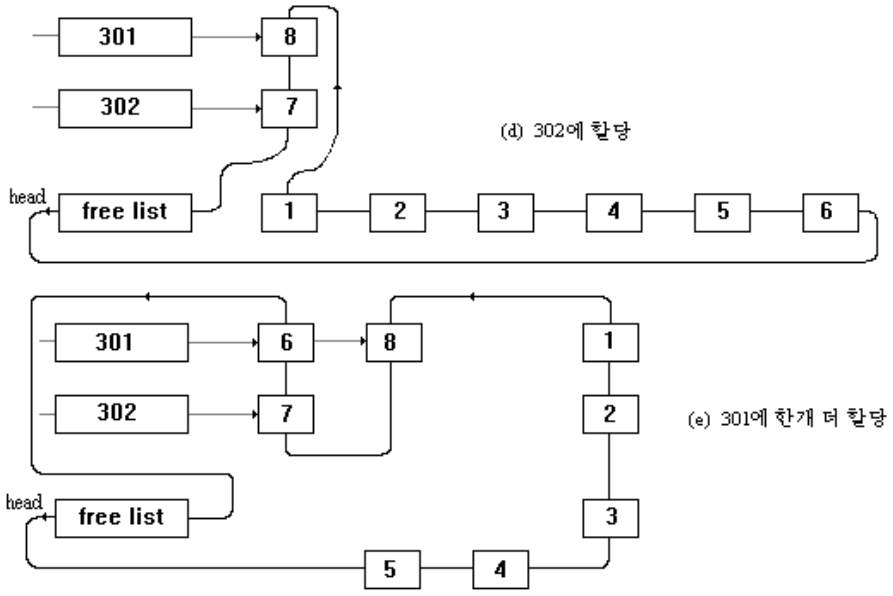
hash table과 할당된 buffer block는 free list처럼 환형의 고리형태로 이루어져 있다.(다음 그림의 (c)의 점선을 보라.) 이들을 연결시키는 고리는 b_next와 b_prev에 의해 이루어진다. 하지만, 편의상 우리는 앞으로 b_next만을 다룬다. 사실, hash table에서 어떤 buffer를 찾을때는 b_next를 사용한다.

4.2.1 hash table

다음은 초기상태에서부터 buffer block이 hash table에 할당되고 제거되는 과정을 설명한다.
처음 free_list에는 buffer초기화(buffer_init)에 의해 다음 그림의 (a)와 같이 4개의 buffer block이 고리를 이루고 있다. (b)처럼 get_blk에 의해 grow_buffers함수가 사용되면, 4개의 buffer block들이 더 생기게 된다. dev=300,block=1을 parameter로 getblk를 호출했을때 free_list의 선두에 있는 buffer block 8번이 hash table 301에 할당된다. 그리고 buffer block 8번은 free_list의 마지막으로 가게 된다. 이때 hash_table[301]에는 8번 buffer_header pointer값이 들어간다.(c)

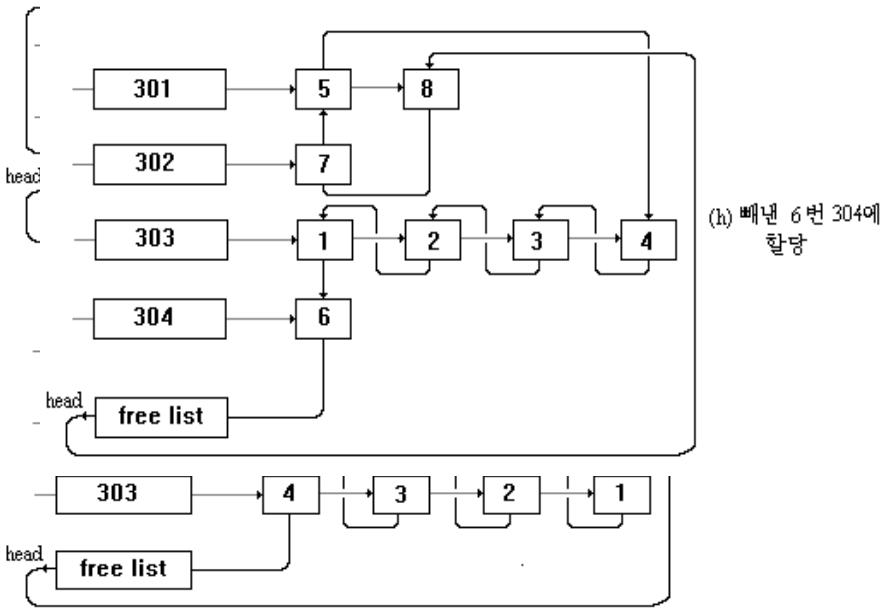
이후 dev=300,block=2에 의한 getblk호출로 hash table 302에 buffer 7번을 할당한다.(d)
다시 dev=300,block=998에 의한 getblk가 호출되면, 6번 buffer block는 hash table 301의 선두에 위치하게 된다. 이 때 hash table[301]에는 6번 buffer block의 buffer_head pointer값이 들어 가게 되고, 6번 buffer header의 b_next는 8번 buffer header를 가리키게 된다.(e)





hash table 301에 buffer block 5번을 추가하자(f)
이제는 설명을 쉽게 하기 위해 page가 고갈되어 buffer 초기화때 만든 buffer까지 모두 사용한 상태에서 hash table 304를 위한 buffer 한개가 요구되었다고 가정하자.(g) 더불어, free_list의 선두에 있는 buffer block 8번과 7번이 다른 process에 의해 사용중이거나 (b_lock), b_dirt가 set되어 sync작업이 필요한 buffer block들이라고 가정하자.

이 때, kernel은 6번 block을 사용할 것을 결정한다. 그래서 6번을 hash table과 free list에서 제거한다.(*remove_from_queues* 함수에 의해) 그 다음, hash table 304에 끼워넣는다.
그리고, free list의 마지막(tail)에 넣는다.(*insert_into_queues* 함수에 의해)(h)



4.2.2. LRU 알고리즘

LRU (least used used)는 최근에 가장 적게 쓰인 것이 제거될 가능성이 높음을 의미한다. kernel은 LRU알고리즘을 사용하여 buffer cache를 관리한다. 최근에 사용된 것은 free list의 맨 끝으로 보낸다. 따라서, 오래전에 사용된 것은 점점 free list의 선두로 오게 되고 다른 device block을 위해 사용될 가능성이 높아지는 것이다.
free list에는 모든 buffer들이 존재하지만, hash table에는 사용중인 buffer만이 들어가게 된다. 이

것은 5장 파일시스템에서 논의되는 inode와 같은 방식으로 관리된다.

4.2.3 lock

process가 어떤 buffer block에 작업을 하는 중에는(즉, buffer에 읽기, 쓰기작업을 하는 중에는) 다른 process가 접근하지 못하도록 **b_lock**를 set한다.[69] 그 때, 다른 process는 buffer block을

69) "locked"와 "busy"는 같은 의미이다. 더불어 "unlocked"와 "free"도 같은 의미이다.

사용할수 있을때까지 (즉, 그것을 사용중인 process가 읽기,쓰기 작업이 끝나 end_request함수에 의해 lock이 해제될때까지) *wait_on_buffer*를 통해 수면(sleep)에 들어간다.
다음은 wait_on_buffer의 코드이다.

```
extern inline void wait_on_buffer(struct buffer_head * bh)
{
        if (bh->b_lock)
                __wait_on_buffer(bh);
}

void __wait_on_buffer(struct buffer_head * bh)
{
        struct wait_queue wait = { current, NULL };

        bh->b_count++;
        add_wait_queue(&bh->b_wait, &wait);
repeat:
        current->state = TASK_UNINTERRUPTIBLE;
        if (bh->b_lock) {
                schedule();
                goto repeat;
        }
        remove_wait_queue(&bh->b_wait, &wait);
        bh->b_count--;
        current->state = TASK_RUNNING;
}
```

현재 process를 wait queue에 넣고 state를 sleep(TASK_UNINTERRUPTIBLE)상태로 한 다음, scheduling을 한다. scheduler에 의해 이 process가 기동하면, 잠자는 동안 lock이 풀렸는지 확인하고 그렇지 않으면 다시 수면에 들어간다. lock이 풀렸으면, wait queue에서 제거하고, 다시 수행상태로 바꾸어 해제된 buffer block를 사용하게 되는 것이다. b_count는 그 buffer block을 사용중인 process를 포함하여 buffer block을 사용하기 위해 기다리는 process들의 수를 나타낸다. 그래서 한 process만이 이 buffer block을 사용한다면 b_count값은 1이 된다.

4.2.4 sync

앞에서도 설명했듯이, b_dirt가 set된 buffer block은 다른 device block에 할당되기위해서는 sync 작업이 필요하다. 이것은 *sync_buffers*함수에 의해 이루어진다.

static int sync_buffers(dev,wait)

dev : 특정 device에 해당하는 buffer block만을 sync. 이 값이 0 이면 모든 buffer block을 대상으로 한다.

wait : 1 이면, b_dirt가 set된 buffer block에 lock이 걸려있을 경우(b_lock set) lock이 풀리기를 기다려가며 모든 buffer block,을 sync한다.
0 이면, lock이 걸리지 않은 buffer block만을 sync한다.

4.2.5 getblk code

이제 코드를 살펴보자

```
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)

struct buffer_head * getblk(dev_t dev, int block, int size)
{
        struct buffer_head * bh, * tmp;
        int buffers;
        static int grow_size = 0;

repeat:
1)      bh = get_hash_table(dev, block, size);
        if (bh) {
                if (bh->b_uptodate && !bh->b_dirt)
                        put_last_free(bh);
                return bh;
        }
        grow_size -= size;
        if (nr_free_pages > min_free_pages && grow_size <= 0) {
2)              if (grow_buffers(GFP_BUFFER, size))
                        grow_size = PAGE_SIZE;
        }
        buffers = nr_buffers;
        bh = NULL;

3)      for (tmp = free_list; buffers-- > 0 ; tmp = tmp->b_next_free) {
                if (tmp->b_count || tmp->b_size != size)
                        continue;
                if (mem_map[MAP_NR((unsigned long) tmp->b_data)] != 1)
```

```
                        continue;
                if (!bh || BADNESS(tmp)<BADNESS(bh)) {
                        bh = tmp;
                        if (!BADNESS(tmp))
                                break;
                }
        }

4)      if (!bh) {
                if (nr_free_pages > 5)
                        if (grow_buffers(GFP_BUFFER, size))
                                goto repeat;
                if (!grow_buffers(GFP_ATOMIC, size))
                        sleep_on(&buffer_wait);
                goto repeat;
        }

5)      wait_on_buffer(bh);
        if (bh->b_count || bh->b_size != size)
                goto repeat;
6)      if (bh->b_dirt) {
                sync_buffers(0,0);
                goto repeat;
        }
/* NOTE!! While we slept waiting for this block, somebody else might */
/* already have added "this" block to the cache. check it */
7)      if (find_buffer(dev,block,size))
                goto repeat;
/* OK, FINALLY we know that this buffer is the only one of its kind, */
/* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
8)      bh->b_count=1;
        bh->b_dirt=0;
        bh->b_uptodate=0;
        bh->b_req=0;
        remove_from_queues(bh);
        bh->b_dev=dev;
        bh->b_blocknr=block;
        insert_into_queues(bh);
        return bh;
```

}

1) hash table에서 원하는 device block의 buffer block이 있는지 찾는다. 찾은 buffer block이 lock상태이면, 해제될때까지 기다린다.
2) free memory가("물리 page가" 라는 말과 동일함) 최소 page(200)이상이면 grow_buffer를 사용하여 일단 buffer를 생산한다. 이 때 buffer는 1 page가 만들어지고, grow_size값은 4k인데, 4K가 모두 사용되면 다시 grow_buffer가 호출된다. 따라서 buffer_init에서 만들어진 buffer block 4개는 초기에는 사용되지 않는다. 메모리가 부족하면 그때서야 사용한다.
3) b_count값이 있다는 것은 적어도 1개의 process가 buffer block을 사용하고 있다는 말이 된다.
   mem_map[]의 값은 해당 page에 대한 현재 상태를 나타내는데, 상태값은 다음과 같다.

MAP_RESERVED (0x8000)	system(kernel)에서 사용하는 부분.
0	free 상태
1	해당 page를 현재 process만이 사용.
2 이상	해당 page를 다른 process와 같이 사용.

<표 4.2> mem_map 상태

   get_free_page(grow_buffers)에 의해 구한 page의 mem_map[]값은 1이다. 그래서 이 값이 1이 아니란 말은 2 이상이란 말이 된다.
   buffer의 BADNESS는 lock이 걸려 있느냐, b_dirt가 set되어 있느냐를 말한다. 일단 이들buffe-r들은 선택대상에서 제외된다. 그러나, 이들중 하나를 고르는 것이 불가피하다면 lock걸린 것을 택하게 된다.(*b_dirt<<1* in BADNESS macro)
4) free_list에 buffer가 없다면, buffer를 더 생산한다.
   memory가 부족하여 더 이상 buffer를 만들수 없으면, memory가 생길때까지 wait queue에서 수면을 취한다.
5) lock이 걸려 있으면 해제될 때까지 기다린다.
6) b_dirt가 set되어 있으면 sync.
7) wait_on_buffer수행하면서 수면하는 중에 혹시 다른 process에 의해 hash table에 원하는 buff-er block이 생겼을지도 모른다.
8) buffer block을 구하면 곧 b_count값을 1로 set한다.

4.3 brelse 함수

user process에 의해 disk와 buffer사이에 data가 오가는 작업이 있으면(make_request에 의해),

kernel은 buffer에 lock을 건다. 그 작업이 끝나면 kernel은 다시 lock을 푼다.(end_requset에 의해)

```
void brelse(struct buffer_head * buf)
{
        if (!buf)
                return;
        wait_on_buffer(buf);  /* wait until unlock  */
        if (buf->b_count) {
                if (--buf->b_count)
                        return;
                wake_up(&buffer_wait);
                return;
        }
        printk("VFS: brelse: Trying to free free buffer\n");
}
```

wait_on_buffer에서 lock을 풀릴때까지 기다린다. b_count를 1 줄인다. 이것으로써 자신이 이 buffer를 사용하지 않음을 표시한다. 그리고 다른 process가 buffer를 기다리고 있었다면

```
        if(--buf->b_count)
```

그냥 return하지만, 자신이 그 buffer를 혼자서 사용하고 있었다면, buffer가 필요해도 메모리 부족으로 buffer를 얻지 못한 process가 없는지 찾아서 깨운다. getblk의 다음 코드를 보라.

```
        sleep_on(&buffer_wait);
```

4.4 bread 함수

dev,block의 data를 buffer block으로 읽어들인다.

```
struct buffer_head * bread(dev_t dev, int block, int size)
{
    struct buffer_head * bh;

    if (!(bh = getblk(dev, block, size))) {
        printk("VFS: bread: READ error on device %d/%d\n",
```

```
                                        MAJOR(dev), MINOR(dev));
        return NULL;
     }
    if (bh->b_uptodate)        /*  buffer block의 data가 유효하면 읽지 않는다. */
        return bh;
    ll_rw_block(READ, 1, &bh); /* bh에 해당하는 device block에서 buffer로 읽어들인다. */
    wait_on_buffer(bh);        /* lock이 풀릴때까지 기다린다. */
    if (bh->b_uptodate)        /* uptodate되어 있어야 한다. end_request함수에 의해 */
         return bh;
    brelse(bh);                /* error발생. buffer release */
    return NULL;
}
```

ll_rw_block함수후에는 wait_on_buffer이 호출되는 것이 보통이다.
ll_rw_block은 buffer block에 lock을 걸고는 disk에서 읽거나 쓰기작업을 한다. disk controller에 이러한 명령을 내린 후에도 코드는 계속 진행되므로 작업이 완성될때까지 wait_on_buffer에 의해 sleep할 필요가 있다. 작업이 완성되면, lock은 해제되고, 비로소 buffer block을 사용할수 있게 된다.

4.5 breada 함수

breada함수는 한 buffer block을 disk block에서 읽을때, 이 후에 사용될것이라고 예상되는 다른 disk block들도 같이 읽는다. 즉, 첫번째 block이 읽혀져 사용되는 동안, 두번째 이후의 block들이 disk에서 미리 읽혀질수 있다. 그래서, 두번째 block부터는 disk에서 읽어들이는 시간을 절약할수 있다.
함수에 전달된 첫번째 parameter인 block은 bread함수에서 하는 것처럼 그냥 읽어들인다. 두번째 block parameter부터가 미리 읽어들이는 block들인데, 이들은

```
ll_rw_block(READA,tmp,&bh)[70]
```

에 의해 읽어들이게 된다. 이 함수에서는 미리 읽어들일려는 block에 해당하는 buffer가 lock되어 있으면 기다리지 않고, 즉시 return하게 된다.[71] 코드 나머지 부분은 READ 명령과 READA명령이 동일하게 동작한다. 하지만, uptodate상태이면 읽지 않고 count값을 줄이기만 한다. getblk는

---

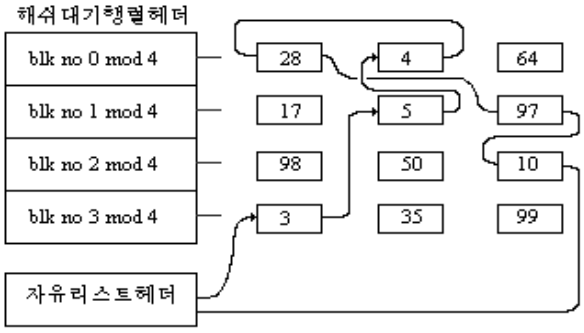
70) READA는 함수에 전달되는 명령이다. ll_rw_block함수에서 사용하는 명령어에는 READ, READA, WRITE, WRITEA등 4가지가 있다.
71) lock이 풀리기를 기다리는 것은, block을 미리 읽어 disk에서 읽는 시간을 줄이겠다는 의도에 상반된다.

buffer block을 할당할때 count를 1 증가시킨다. 이것을 원래대로 되돌려 놓는 것이다.

4.6 System V의 buffer cache

UNIX 운영체제의 설계 - Maurice J.Bach 라는 책에 나와있는 System V Release 2의 buffer 와 리눅스 buffer를 비교한다.
이 책의 76page(한글판)를 보면 다음의 두 구절이 있다.

1. 임의의 빈 버퍼를 찾고 있으면 자유리스트(free list)에서 버퍼를 지운다.
   리눅스에서는 hash table에서 찾을수 없어서 새로운 buffer를 할당할때는, free list의 선두에서 구해서, 구한 buffer를 free list의 마지막에 둔다.
2. 버퍼는 항상 해쉬대기행렬상에 있지만 자유리스트상에는 있을 수도 없을 수도 있다.
   System V에서는 새로운 buffer가 만들어지면 hash table과 free list에 존재하지만, 어떤 buffer가 할당되면 그 buffer는 free list에서 빠진다. 그러나 리눅스에서는 buffer가 만들어지면, 만들어진 buffer는 무조건 free list에 있게 되며, 할당된 buffer만이 hash table에 들어가게 된다.
   참고로 이 책에 나오는 그림을 옮겨놓았다.



(a) 첫번째 해쉬대기행렬에서 블록 4를 검색

5장

파일시스템

5.1 리눅스 filesystem

리눅스에서 지원하는 filesystem의 종류에는 여러가지가 있다. 다음은 kernel.howto에서 발췌된 내용이다.

3.3.8. Filesystems

You'll be prompted for support of a number of filesystems. They are:

Standard (minix) - Newer distributions don't create minix filesystems,

```
and many people don't use it, but it may still be a good idea to
configure this one. Some ``rescue-disk'' programs use it, and still
more floppies may have a minix filesystem.

Extended fs - This was the first version of the extended filesystem,
which isn't used much anymore. Chances are, you'll know it if you need
it.

Second extended - This is widely used in new distributions. You
probably have one of these.

xiafs filesystem - At one time, this was pretty common, but at the
time of this writing, I didn't know anyone running it.

msdos - Well, you guessed it; if you want to use your MS-DOS hard disk
partitions, or mount MS-DOS formatted floppy disks, say ``y.''

umsdos - This is a fairly slick filesystem which can make an MS-DOS
filesystem have more features, like long filenames, etc. It's not
really useful for people (like me) who don't run MS-DOS.

/proc - One of the most slick filesystems (idea shamelessly stolen
from Bell Labs, I guess). It's not anything you partition disks with;
but a filesystem interface to the kernel and processes. Many process-
listers (like ``ps'') use it. If you've got it installed, try ``cat
/proc/meminfo'' or ``cat /proc/devices'' sometime.  Some shells, like
rc, use /proc/self/fd  (known as /dev/fd on other systems) for i/o.

NFS - If you're on a network and you want to share files, say ``y.''

ISO9660 - Found on most CD-ROMs.

OS/2 HPFS - At the time of this writing, a read-only fs for OS/2 HPFS.

System V and Coherent - for partitions of System V and Coherent
systems.
```

이들중 리눅스 filesystem으로는 second extended filesystem(*ext2fs*)을 주로 사용한다. ext2fs에서는 file이름이 256 byte까지 가능하며, 4 terabyte까지의 filesystem을 구축할수 있다.

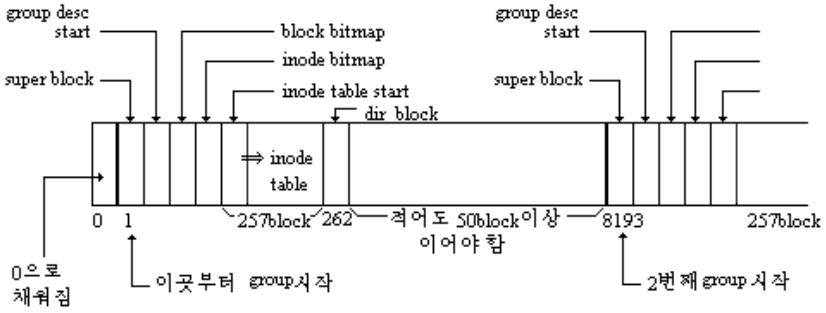
5.2 filesystem 구조

slackware에서 제공되는 install 디스켓으로 filesystem을 설치하기 위해 setup을 실행시킨다. 이것은 setup shell script에서 mke2fs라는 명령어에 의해 filesystem이 설치된다.

```
mke2fs -c $ROOT_DEVICE $ROOT_SIZE
```

"c" 옵션은 filesystem을 설치하기 전에 hard disk에 물리적인 error(bad block)가 없는지 check 한다. ROOT_DEVICE는 filesystem을 설치할 partition을 의미하며, ROOT_SIZE에는 일반적으로 그 partition의 크기가 들어간다.

우선 mke2fs 유틸리티에 의해 설치된 기본적인 filesystem에 대해 설명한다. 편의상 본인이 설치하여 사용중인 filesystem을 예로 들것이다. 본인은 ROOT_DEVICE로 /dev/hda3을 사용하며, 리눅스 설치용량은 248 Mbyte로서, ROOT_SIZE는 254016(block수)이 된다.[72]
그림<5.1>은 이러한 ROOT_SIZE에서의 초기 filesystem모습이다.



<그림5.1> filesystem 구조

72) 참고로 본인이 사용중인 swap partition크기는 8M이고 /dev/hda2에 할당되어 있다. hda1은 DOS용으로 사용한다.

5.3 super block

첫번째 block(block 0, Null block)은 0로 채워져 있다. 두번째 block부터는 group으로 나누어져 있다. 한 group의 크기는 8 Mbyte이다. 그래서 /dev/hda3의 경우 31개의 group으로 이루어짐을 알수 있다. 각 group마다 super block, group descriptor block, block_bitmap block, inode_bitmap block, inode table block, directory entry block들이 있다. 이 중 super block과 group descriptor block은 group과 상관없이 동일하다.[73] 왜냐하며 이 두 곳에는 filesystem전체에 대한 정보가 들어가기 때문이다. 나머지는 해당 group의 상태에 따라 내용이 달라진다.
두번째 block(block 1)은 super block이다.
물론 8192를 더한 block 8193도 super block이며, block 1의 내용과 동일하다. 그렇지 않다면, filesystem에 이상이 있는 것이다. 이때는 e2fsck 유틸리티를 사용하여 filesystem을 고칠수 있다.

```
# e2fsck -a /dev/hda3  /* -a는 error 정정여부를 묻지 않고 자동적으로 정정하라는 것. */
```

super block은 include/linux/ext2_fs.h에 나와 있는 다음 구조체로 이루어져 있다. 이 구조체의 크기는 정확히 1kbyte이다. 오른쪽의 이탤릭체들은 초기 값들이다.
구조체 내용이 어려우면, 이 장의 나머지를 먼저 읽어보면 이해할수 있을 것이다.

```
/*
 * Structure of the super block
 */
struct ext2_super_block {
        unsigned long  s_inodes_count;         63736   /* Inodes count */
        unsigned long  s_blocks_count;         253953  /* Blocks count */
        unsigned long  s_r_blocks_count;       12697   /* Reserved blocks count */
        unsigned long  s_free_blocks_count;            /* Free blocks count */
        unsigned long  s_free_inodes_count;            /* Free inodes count */
        unsigned long  s_first_data_block;     1       /* First Data Block */
        unsigned long  s_log_block_size;       0       /* Block size */
        long           s_log_frag_size;        0       /* Fragment size */
        unsigned long  s_blocks_per_group;     8192    /* Blocks per group */
        unsigned long  s_frags_per_group;      8192    /* Fragments per group */
        unsigned long  s_inodes_per_group;     2056    /* Inodes per group */
```

73) 이것은 e2fsck에 의해 filesystem을 정상화시킬때 사용된다. e2fsck는 첫번째 super block이 파괴되었다고 판단되면 두번째 super block을 이용한다.

```
        unsigned long  s_mtime;                         /* Mount time */
        unsigned long  s_wtime;                         /* Write time */
        unsigned short s_mnt_count;                     /* Mount count */
        short          s_max_mnt_count;        20       /* Maximal mount count */
        unsigned short s_magic;                EF53     /* Magic signature */
        unsigned short s_state;                1        /* filesystem state */
        unsigned short s_errors;               0       /* Behaviour when detecting errors*/
        unsigned short s_pad;                  0       /*  */
        unsigned long  s_lastcheck;            0       /* time of last check */
        unsigned long  s_checkinterval;        0       /* max. time between checks */
        unsigned long  s_reserved[238];        0      /* Padding to the end of the block */
};
```

- s_inodes_count 는 4Kbyte당 inode 1개씩을 할당하면 inode수는 충분하다고 본다. 실제값보다 다소 큰것은 group마다 같은 양이 배분되도록 계산되어 진것이다. 이 값은 system이 사용할수 있는 최대 inode 수이다.
  보다 정확한 계산은 mke2fs의 source를 참고하기 바란다.
- s_blocks_count는 filesystem전체의 block 수이다. 실제 patition block수보다 작은 것은 8192 block으로 group를 할당하고 마지막에 할당하기에 부족한 block들은 사용하지 않는다. 그래서 이 값은 다음과 같다.

  group수 × 8192 + 1

- s_r_blocks_count는 s_block_count의 5%에 해당한다.
- s_free_blocks_count과 s_free_inodes_count 는 사용하고 남은 block과 inode의 수이다.
- s_first_data_block은 첫번째 super block 번호를 가리킨다.
- s_log_block_size과 s_log_frag_size는 block 크기로서 0이면 1024 byte, 1이면 2048 byte, 2이면 4096 byte를 나타낸다.
- s_blocks_per_group과 s_frags_per_group은 한 group의 block 수이다.
- s_inodes_per_group은 한 group의 inode 수이다.
- s_mtime은 filesystem을 mount 한 시간이다.
- s_wtime은 super block의 내용을 갱신한 시간이다.(/ext2/super.c의 ext2_write_super함수참고)
- s_mnt_count는 filesystem을 mount한 횟수이다. system을 재부팅시킬때마다 이 값이 1씩 증가함을 발견할수 있다.[74] (ext2_setup_super함수에 의해)

74) e2dump라는 유틸리티를 사용하면 원하는 block의 내용물을 dump해 볼수 있다.

- s_max_mnt_count는 한번에 mount할수 있는 최대수이다.(ext2_fs.h참고)
- s_magic은 ext2 filesystem의 super magic number이다.
- s_state은 현재 filesystem의 상태로서 1이면 정상, 2이면 error이다.(ext2_fs.h참고)
- s_errors은 error가 발견되었을때 set된다.
- s_pad은 사용하지 않는 것으로 보인다.
- s_lastcheck는 mke2fs와 e2fsck가 마지막으로 system을 check한 시간.
- s_checkinterval는 lastcheck후 이 시간이 지나면, mount시킬때 다음과 같은 메세지를 출력한다.

  " EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended "

5.4 group descriptor

group descriptor에는 해당 group에 대한 정보가 들어있다.
그 내용은 다음과 같다.

```
struct ext2_group_desc
{
        unsigned long  bg_block_bitmap;         /* Blocks bitmap block */
        unsigned long  bg_inode_bitmap;         /* Inodes bitmap block */
        unsigned long  bg_inode_table;          /* Inodes table block */
        unsigned short bg_free_blocks_count;    /* Free blocks count */
        unsigned short bg_free_inodes_count;    /* Free inodes count */
        unsigned short bg_used_dirs_count;      /* Directories count */
        unsigned short bg_pad;
        unsigned long  bg_reserved[3];
};
```

이 구조체는 32 byte 크기이다. 따라서 1 block에 32개가 들어갈수 있다. 본인의 system은 31개의 group이므로 group descriptor block으로 한개의 block이면 된다.

5.5 block bitmap과 inode bitmap

그 외에도 file system을 분석하기에 유용한 기능들을 갖추고 있다.

block bitmap block내의 1 bit은 1 block을 나타낸다. 따라서 1개의 bitmap block으로 8192개(1 group)의 block 사용상태를 표현할수 있다. 예를 들어 Null block은 bitmap block의 첫 byte의 bit 0에 mapping된다. 이 block은 초기에 set되어 사용중임을 표시한다. 즉 bit가 set됨은 해당 block이 사용중임을 나타낸다. 따라서 이것으로서 kernel은 비어있는 block을 찾아 data를 hard disk에 쓰기작업을 할수 있다. Null block뿐만 아니라 super block, group descriptor block, inode table등의 filesystem 정보를 담고 있는 block들 모두가 초기에 사용중인 것으로 표시된다.

inode bitmap도 같은 방식(1 inode / 1 bit)이다. 한 group에 할당된 inode수에 대한 mapping이 1 bitmap block에 의해 mapping된다. group당 inode수가 2056개 이므로 1 bitmap block의 1/4만으로 각 inode를 표현할수 있다.

5.6 inode table

각 file은 각자 1개의 inode를 가지고 있다. 이 inode table block에 들어가는 inode descriptor의 구조를 보자.

```
/*
 * Structure of an inode on the disk
 */

struct ext2_inode {
        unsigned short i_mode;          /* File mode */
        unsigned short i_uid;           /* Owner Uid */
        unsigned long  i_size;          /* Size in bytes */
        unsigned long  i_atime;         /* Access time */
        unsigned long  i_ctime;         /* Creation time */
        unsigned long  i_mtime;         /* Modification time */
==>     unsigned long  i_dtime;         /* Deletion Time */
        unsigned short i_gid;           /* Group Id */
        unsigned short i_links_count;   /* Links count */
        unsigned long  i_blocks;        /* Blocks count */
        unsigned long  i_flags;         /* File flags */
        unsigned long  i_reserved1;
```

```
==>     unsigned long  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        unsigned long  i_version;       /* File version (for NFS) */
        unsigned long  i_file_acl;      /* File ACL */
        unsigned long  i_dir_acl;       /* Directory ACL */
        unsigned long  i_faddr;         /* Fragment address */
        unsigned char  i_frag;          /* Fragment number */
        unsigned char  i_fsize;         /* Fragment size */
        unsigned short i_pad1;
        unsigned long  i_reserved2[2];
};
```

1) dtime

file이 삭제된 시간. file을 삭제하는 것에 대해서는 5.9절에서 더 논의 된다.

2) i_block

kernel이 file을 hard disk에 기록할 때, 이 i_block[]배열에 file위치를 기록한다. 배열에는 file data가 있는 block에 대한 block 번호가 들어있다. 그런데 배열만으로는 15(EXT2_N_BLOCKS) 개의 block 밖에 가리킬수 없다. 그런데, 만약 file 크기가 15 kbyte를 넘으면 어떻게 할 것인가? 실제로는 다음과 같이 사용된다.

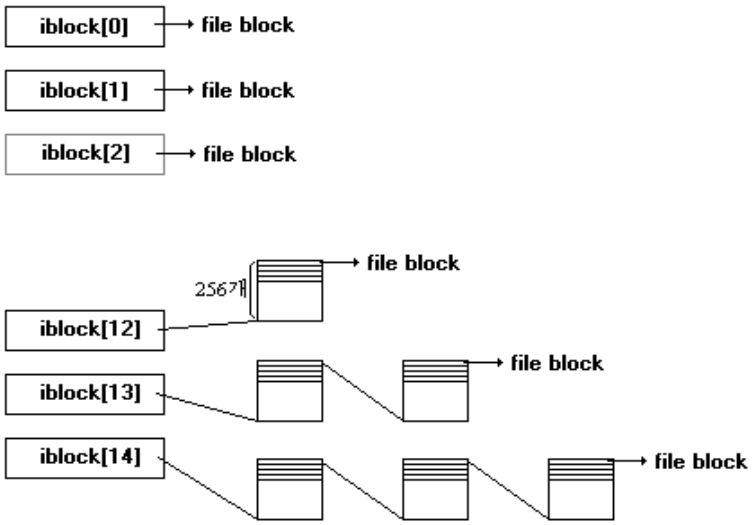
i_block[0] ~ i_block[11]까지만이 실제 file data가 있는 block을 가리킨다. 그러면, 만약에 13 kbyte인 file의 13번째 block은 누가 가리키는가? 이것은 i_block[12]가 가리키는 block에 가보면 있다. i_block[12]의 처음 4 byte가 가리키는 것이 13번째 block이다.

그럼 file 크기가 14 kbyte이면? 당연히 i_block[12]의 5,6,7,8 byte가 14번째 block이다. 이런식이면 한 block에 256개의 block번호(4 byte)가 들어갈수 있으므로 12+256개의 block을 가리킬 수 있음을 알수있다. (그림<5.2>)

kernel은 file을 disk에 기록할 때, 12kbyte가 넘으면 즉시 놀고있는 block을 할당하여 i_block [12]에 그 block번호를 넣고, 할당된 block에다 file이 기록되어질 block번호들을 쓰기 시작한다. 그럼 만약에 file이 268 kbyte를 넘으면 어떻게 될까? 그 때 kernel은 역시 놀고있는 다른 새로운 block을 할당하여 그 block번호를 i_block[13]에 넣는다. 그런데 이번에 할당된 block에 쓰여질 block번호는 실제 data가 있는 block이 아니다. 즉, i_block[12]와 같은 block을 256개나 가리키는 block이다. file이 269 kbyte이면 269번째 block 번호는 어디에 기록되는지를 보자. kernel은 i_block[13]에 번호가 들어갈 block을 할당후, 1개의 새로운 block을 더 할당한다. 즉 연달아 두개를 할당해야 한다. 그리고, 두번째 할당된(사실 12 kbyte를 넘을때도 한개 할당했으므로 3번째이다.) block 번호를 첫번째 할당한 block(i_block[13]이 가리키는)의 1,2,3,4 byte에 기록한다. 자! 여기서 우리는 i_block[13]이 가리킬수 있는 block은 결과적으로는 256×256개임

을 알수 있다. 그래서 i_block[0] ~ i_block[13]까지를 사용하여 12+256+256×256개의 block들을 가리킬수 있게 된다.

i_block[14]도 같은 방식으로 256×256×256(16 Gbyte)개의 block을 가리킨다.



그림<5.2> file의 사용block 관리체계

그림<5.2>과 표<5.1>은 직접 file block번호를 가리키는 방법과 간접, 이중간접, 삼중간접으로 file block을 가리킴에 따라 가능한 file 크기를 나타내었다.

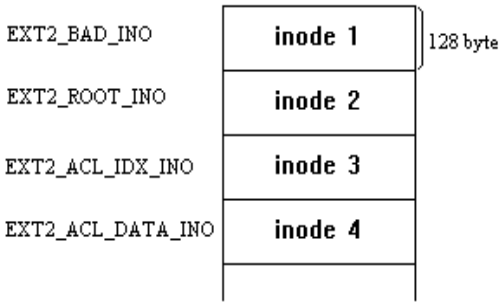
직접	iblock[0-11]	12 kbyte
간접	iblock[12]	256 kbyte
이중간접	iblock[13]	64 Mbyte
삼중간접	iblock[14]	16 Gbyte

<표5.1>

이것으로서 이론적으로 가능한 리눅스 file의 최대크기는 16 Gbyte + 64 Mbyte + 268 kbyte

이다.

ext2 inode구조체는 128 byte의 크기를 가진다. inode table에는 다음과 같은 구조로 inode들이 들어간다.
inode는 할당된 inode 번호를 가진다. inode table은 inode 번호 1 인 inode부터 차례로 들어간다. 따라서 file의 inode 번호만 알면, inode descriptor를 찾아서 그 file의 disk상의 위치를 알수있다. 그림<5.3>에 나타나 있는 4개의 inode는 항상 예약되어 있는 것들이다. 이 외에도 예약되어있는 것들이 더 있다.



<그림5.3> inode table start

```
/*
 * Special inodes numbers
 */
#define EXT2_BAD_INO             1      /* Bad blocks inode */
#define EXT2_ROOT_INO            2      /* Root inode */
#define EXT2_ACL_IDX_INO         3      /* ACL inode */
#define EXT2_ACL_DATA_INO        4      /* ACL inode */
#define EXT2_BOOT_LOADER_INO     5      /* Boot loader inode */
#define EXT2_UNDEL_DIR_INO       6      /* Undelete directory inode */
#define EXT2_FIRST_INO          11      /* First non reserved inode */
```

inode table에서 처음 10개의 inode는 예약된 것들이다. 따라서, 첫 inode는 11번째부터이며 mke2-fs는 이 inode 번호에 "/lost+found"를 할당한다. e2fsck는 root directory(/root를 가리키는 것이 아님)와 연결되지 않은 directory나 비정상적인 file에 대해서는 기존의 directory에 제거시키고,

/lost+found와 연결(reconnect,link)시킨다. 만약에 /lost+found directory가 존재하지 않으면 direct-ory를 생성시킨후 연결한다.

EXT2_BAD_INO는 e2fsck에 의해 관리되며, 물리적으로 error가 난 block을 표시하기 위해 사용된다. 이 inode의 i_block[]에는 bad block 번호가 들어간다. hard disk에서 사용하지 못하는 bad block을 표시하는 절차는 다음과 같다.

- inode 1 (EXT2_BAD_INO)에 해당 block을 명시한다.
- inode bitmap에 "사용중"인것으로 표시한다.

EXT2_ROOT_INO는 root directory를 위한 inode 이다. 기본 filesystem이 이 root directory 상에서 만들어진다. system이 부팅될때 kernel은 이 directory에 filesystem을 mount시킨다.

EXT2_ACL_IDX_INO, EXT2_ACL_DATA_INO, EXT2_BOOT_LOADER_INO, EXT2_UNDEL_DIR_INO는 EXT2_BAD_INO의 경우처럼 어떠한 program에 의해 결정되거나, 앞으로 만들어질 기능인 것들로 예상된다. 특히 EXT2_ACL_IDX_INO, EXT2_ACL_DATA_INO는 version 1.0에서는 완성이 덜 된 경우로 추정된다.(fs/ext2/acl.c의 주석 참고)

EXT2_FIRST_INO는 user가 사용할수 있는 inode의 시작번호이다. 앞에서도 말했듯이 filesyste-m을 구축한 초기에는 이것이 /lost+found 디렉토리에 해당된다.

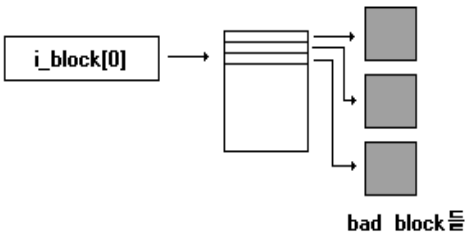
inode block은 block 5에서 시작하여 (group당 inode수에 따라) 257개의 block이 할당된다. 다음은 block 5를 dump하였다. 초기에는 예약 inode중 단지 inode 1과 2만이 사용중에 있다.

```
Block 5:
00 00 00 00 00 00 00 00 45 BD 26 2F 45 BD 26 2F   inode 1
45 BD 26 2F 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

ED 41 00 00 00 04 00 00 90 A1 E9 2F 16 A1 E9 2F   inode 2
```

16 A1 E9 2F 00 00 00 00 00 00 13 00 02 00 00 00
00 00 00 00 00 00 00 00 **12 01 00 00** 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 inode 3

inode 1에서 굵은 글씨체로 표시된 부분을 주목하자. 이 곳은 i_block[0]에 해당하는 곳이다. 만약에 bad block가 있으면 여기에 bad block들의 번호가 들어있는 block의 block번호가 들어간다. 그림<5.4>을 보자.



<그림5.4> EXT2_BAD_INO

위에 나타나 있는 inode 1에 따르면 본인의 hard disk는 깨끗하다는 것을 알수 있다.
inode 2의 i_block[0]는 block 274를 가리키고 있다. 즉, block 274는 root directory를 위한 directory block이다. 이 곳에는 root directory에 있는 file 및 directory들의 이름, 이름길이 등의 정보가 들어있다. 이것은 이 inode가 directory에 해당되기 때문이며, file inode의 경우는 이곳에 file이 상주하고 있는 block번호가 들어갈 것이다. directory block에 대한 자세한 것은 5.7절에서 다룬다.

ED 41 00 00 00 30 00 00 C4 8A E9 2F 44 BD 26 2F *inode 11*
44 BD 26 2F 00 00 00 00 00 00 02 00 18 00 00 00
00 00 00 00 00 00 00 00 06 01 00 00 07 01 00 00

```
08 01 00 00 09 01 00 00 0A 01 00 00 0B 01 00 00
0C 01 00 00 0D 01 00 00 0E 01 00 00 0F 01 00 00
10 01 00 00 11 01 00 00 00 00 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

위의 inode내용은 block 6에 있는 inode 11이다. 이것은 /lost+found directory에 대한 것인데, mke2fs에 의해 12개의 directory block이 미리 할당되어 있다. 다른 일반적인 directory들은 directory entry[75]의 수가 늘어남에 따라 block이 더 할당된다. 마지막 할당된 directory block은 block 273이다. 그 다음 block부터 root directory block이 시작됨을 주목하자.
12개의 할당된 block에는 불량 directory, 불량 file들이 directory entry로서, 들어갈 것이다.

5.7 directory block

directory block에는 해당 directory의 entry들에 대한 정보가 다음의 구조체(ext2_fs.h)에 의해 이루어진다.

```
/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
        unsigned long  inode;                   /* Inode number */
        unsigned short rec_len;                 /* Directory entry length */
        unsigned short name_len;                /* Name length */
        char           name[EXT2_NAME_LEN];     /* File name */
};
```

이 구조체는 file 또는 directory이름에 따라 구조체의 크기가 가변적이다. rec_len은 해당 file의 구조체의 크기를 나타낸다. 이 값은 다음 정의에 의해 4의 배수로 이루어진다.

---

75) file 또는 sub-directory

```
/*
 * EXT2_DIR_PAD defines the directory entries boundaries
 *
 * NOTE: It must be a multiple of 4
 */
#define EXT2_DIR_PAD                    4
#define EXT2_DIR_ROUND                  (EXT2_DIR_PAD - 1)
#define EXT2_DIR_REC_LEN(name_len)      (((name_len) + 8 + EXT2_DIR_ROUND) & \
                                         ~EXT2_DIR_ROUND)
```

다음은 root directory block인 block 274의 앞부분 내용이다.
굵은 글씨체는 file명(name)이다. inode 2는 "."과 ".."이고 inode 11은 "/lost+found"이며, inode 59854는 "/proc"이다.

Block 274:

02	00	00	00	0C	00	01	00	**2E**	00	00	00	02	00	00	00
0C	00	02	00	**2E**	**2E**	00	00	0B	00	00	00	14	00	0A	00
6C	**6F**	**73**	**74**	**2B**	**66**	**6F**	**75**	**6E**	**64**	00	00	CE	E9	00	00
10	00	04	00	**70**	**72**	**6F**	**63**	62	00	00	00	09	08	00	00

5.8 file 찾기(file lookup)

지금까지 논의한 것으로 kernel이 임의의 file을 어떻게 찾는지를 알수 있다. 예로서 open system call에 의해 임의의 file을 kernel이 찾아야 할 경우를 생각해 보자. parameter로 제공되는 file명에 의해 어느 directory에서 찾아야 할지를 결정해야 한다.

```
fd = open("/foo",O_RDWR);
```

일 경우는 root directory에서 "foo" file을 찾아야 하고,

```
fd = open("foo",O_RDWR)
```

일 경우는 현재 directory에서 "foo" file을 찾아야 한다.

task_struct내의 member인 root와 pwd를 기억하는가? 이들의 type은 inode구조체 pointer이다. "/"는 root directory로서 부팅시 filesystem이 mount될때[76] ext2_read_super함수(fs/ext2/super.c)와 mount_root함수(fs/super.c)에 의해 set된다. root에는 inode 2(EXT2_ROOT_INO)의 pointer가 들어간다.
아래 코드에서 이탤릭체로 된 **sb->mounted**를 기억해 두자.

```
/*
 * set up enough so that it can read an inode
 */
sb->s_dev = dev;
sb->s_op = &ext2_sops;
if (!(sb->s_mounted = iget (sb, EXT2_ROOT_INO))) {
```

< ext2_read_super함수 내에서>

```
sb = read_super(ROOT_DEV,fs_type->name,root_mountflags,NULL,1);
      if (sb) {
              inode = sb->s_mounted;
              inode->i_count += 3 ;
              sb->s_covered = inode;
              sb->s_flags = root_mountflags;
              current->pwd = inode;
              current->root = inode;
```

< mount_root함수 내에서>

pwd는 현재 작업중인 directory로서, directory를 이동할때마다 set된다.

현재 directory에서 "foo"를 찾는다고 하자. kernel은 먼저 pwd inode에서 i_block[0]을 이용하여

76) *setup* system call 내에서 root directory가 mount된다.

현재 directory의 첫번째 directory block을 찾는다. 그 block에는 현재 directory 내에 있는 file들의 *dir_entry descriptor*들이 있을 것이다. 그 곳에서 "foo"라는 이름을 가진 descriptor를 찾는다. 찾은 descriptor에는 inode 번호가 있다. inode table에는 inode 구조체가 inode번호 순서대로 있으므로, "foo"의 inode를 찾을수 있다. OK! Here we are. 그러면 실제 file이 있는 곳은? "foo" inode의 i_block[]에 의해 찾으면 된다.

그런데 dir block에서 "."과 ".." entry는 특별하다.
"."은 현재 directory를, ".."은 바로 윗 directory(parent directory)를 나타낸다. 예로서 "/lost+found"의 directory block인 block 262의 내용을 보자.

Block 262:
0B 00 00 00 0C 00 01 00 2E 00 74 24 **02** 00 00 00
F4 03 02 00 2E 2E 00 83 7E 04 00 7F 06 66 C7 46
04 01 00 66 83 7E 06 00 7F 06 66 C7 46 06 01 00

byte 0의 OB는 "/lost+found"의 inode 번호이고, byte 12의 02는 "/"의 inode 번호로서 root directory를 가리킨다. byte 24부터는 쓰레기(trash)값들이다.

dir_entry수가 많아서 확장된 dir block에는 "."와 ".."는 없으며, directory inode의 i_block[1]부터가 확장된 block에 해당된다.

이제 적어도 "cd"와 "find"명령어가 수행되는 과정은 이해했을 것이다.

5.9 file 삭제

file을 삭제시키는 system call은 unlink(sys_unlink함수)이다.

unlink는 다음과 같은 작업을 한다.

1) directory block에서 해당 file descriptor를 찾아 inode number를 0 로 만든다. 따라서, 이 후 kernel이 file을 찾을려고 하면, inode table에서 해당 file의 inode를 찾을수 없게 된다. inode number는 file 이름과 inode를 연결시켜주는 유일한 수단이다. 그리고, 지울려는 file의 rec_len

값을 바로 앞에 놓인 file의 rec_len에 합친다.(ext2_delete_entry함수<fs/ext2/namei.c>)

2) iput함수에 의해 inode를 free시킨다. inode bitmap block에서 해당 bit를 "미사용"으로 만든다. 그리고 삭제시각인 i_dtime(ext2_free_inode함수<fs/ext2/ialloc.c>)을 set한다.

3) i_nlink는 현재 그 inode를 사용하는 file수를 말한다. 5.11.2 절에서 논의된다.

5.10 inode구조체(fs.h)와 ext2_inode구조체(ext2_fs.h)

inode구조체는 memory에서 kernel이 inode를 관리하기 위한 구조체이며 process와 관련된 변수들이 들어 있다. 한편, ext2_inode구조체는 앞에서도 설명했듯이 filesystem의 inode table에 들어가 있는 내용이다. open system call은 file을 개방할때 ext2_read_inode함수에 의해 ext2_inode구조체의 내용을 읽어 inode구조체에 넣는다.

5.11 symbolic link와 hard link

5.11.1 symbolic link

ext2_new_inode<ialloc>함수에 의해 놀고있는 inode를 하나 구한다. 이 inode는 symbolic link에 의해 만들어질 target file의 inode이다. 그런데, 이 inode의 i_block[EXT2_N_BLOCK][77]에는 15자 이내라는 조건으로 source file의 이름이 들어간다. 만약 source file의 이름이 15자를 넘는다면 새로운 block을 할당하고, 그곳에 이름을 적어 놓는다. 그리고, target inode->i_block[0]에다가 그 block 번호를 넣는다.
이렇게 함으로써 target file은 단지 이름만으로 source file을 찾아간다.

5.11.2 hard link

77) memory에 있는 inode구조체의 i_data[]는 hard disk에 있는 i_block[]과 같다고 보면 된다. file을 open할때 kernel은 i_block[]배열의 값을 그대로 i_data[]에 copy한다. (ext2_read_inode함수)

hard link는 inode를 더 만들지 않는다. 즉 source file의 inode를 공유한다. hard link는 i_nlink값을 1 증가시키고, directory block에 target file의 entry를 넣는다. hark link는 한 inode에 file 이름이 두개 있는 것이다.
symbolic link시에 "ls -l" 로 확인해 보면 source file의 link수가 변하지 않는 것을 볼수 있다.

```
# ln -sf gcc foo
# ls -l gcc foo
-rw-r--r--   1 root     root       149399 May  6 02:27 gcc
lrwxrwxrwx   1 root     root            3 Jun 23 20:58 foo -> gcc
```

target file과 다른 inode를 사용하기 때문이다. 그리고 file 크기가 source file의 이름길이인 3 byte임을 볼수 있다.
그러나 hard link시에는 source file의 link수가 1 증가한다. 더불어 target file의 link수는 source file의 link수와 동일하다. i_nlink는 *inode구조체*(dir_entry가 아닌)에 속해있지 않은가?

```
# ln gcc foo
# ls -l gcc foo
-rw-r--r--   2 root     root       149399 May  6 02:27 gcc
-rw-r--r--   2 root     root       149399 May  6 02:27 foo
```

마지막으로, 이들의 생성시각을 살펴보자. symbolic link시에는 target file의 시각이 link를 실시한 시각이다. 그러나 hard link시에는 source file의 생성시각을 따라간다. 같은 inode를 사용하므로 source file의 시각이든, link를 한 시각이든간에 둘중에 하나만 사용할수밖에 없는 실정이다.

5.12 open system call

```
#define NR_OPEN 256          /* 한 process가 다룰수 있는 file수  */
#define NR_FILE 1024         /* 전체 process가 다룰수 있는 file수 */
```

1) file descriptor

open system call은 먼저 개방할려는 file에 file descriptor 번호를 할당한다. 이것이 open 호출시 되돌려지는 값이며, open system call을 호출한 process에게 관리가 한정된 번호이다. 그리고 번

호 0, 1, 2는 standard input/output/error을 위해 특정 terminal device에 미리 할당되어 있다. 부팅과정에서 setup system call 다음에 console을 위해 **standard** file descriptor를 할당한 다음 코드를 기억하자.

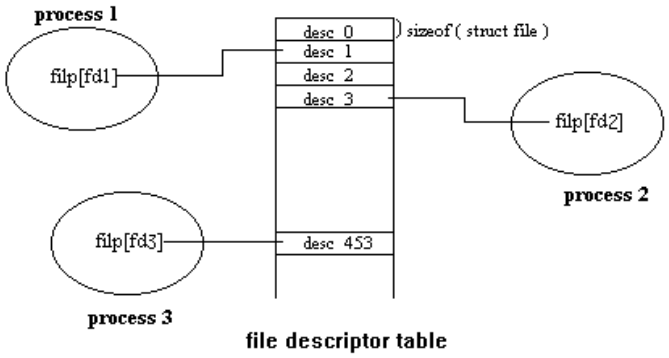
```
(void) open("/dev/tty1",O_RDWR,0);
(void) dup(0);
(void) dup(0);
```

file구조체(struct file) 형식인 file descriptor는 memory상에 고리로 연결된 table모양을 이루며 초기에는 1024개가 상주한다. 처음 생성되었을때는 자유로운 상태인데, 그것을 process가 필요하면 얻어다 사용하는 것이다. 그 때 process는 descriptor번호에 사용할 file descriptor를 연결시킨다.

```
current->filp[fd] = file descriptor pointer
```

memory상의 자유로운 file descriptor는 어느 process든지 필요하면 가져다 사용하고, 다 사용하고 나면 반납한다.

그림<5.5>은 file descriptor번호와 descriptor와의 관계를 나타내었다.



<그림5.5> file descriptor

다음 구조체는 file descriptor의 구조를 나타낸다.

```
struct file {
        mode_t f_mode;
        dev_t f_rdev;                   /* needed for /dev/tty */
        off_t f_pos;
        unsigned short f_flags;
        unsigned short f_count;
        unsigned short f_reada;
        struct file *f_next, *f_prev;
        struct inode * f_inode;
        struct file_operations * f_op;
};
```

f_count 값만큼의 process들에 의해 이 file이 사용중에 있음을 나타낸다.

2) open_namei

이 함수는 open system call 코드의 대부분을 차지한다.

```
1)      if ((inode->i_count > 1) && (flag & 2)) {
                for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
            ==>         struct vm_area_struct * mpnt;
                        if (!*p)
                                continue;
                        if (inode == (*p)->executable) {
                                iput(inode);
                                return -ETXTBSY;
                        }
                        for(mpnt = (*p)->mmap; mpnt; mpnt = mpnt->vm_next) {
                                if (mpnt->vm_page_prot & PAGE_RW)
                                        continue;
                                if (inode == mpnt->vm_inode) {
                                        iput(inode);
                                        return -ETXTBSY;
                                }
                        }
                }
        }
```

위의 open_namei코드에서 1)로 표시된 이 조건문은 한 file을 두개이상의 process가 사용하고 있는 상태에서(inode->i_count > 1), file에 write하려고(flag&2) 시도한 경우이다.
vm_area_struct는 unix에서 흔히 *영역*이라고 불리우는 memory 관리체계와 관련된 구조체로서 자세한 것은 7장 메모리 관리에서 다룬다. 여기서는 일단 이 구조체가 *mmap* system call 수행시 disk와 mapping이 이루어진 memory영역에 대한 정보를 담고 있음을 알아두자. 코드의 이탤릭체에서 vm_inode는 mapping된 file의 inode이며, vm_page_prot는 이 영역에 대해 읽기와 쓰기 작업이 수행가능한지 여부를 나타내고 있다. 쓰기작업이 가능(PAGE_RW)하지 않다면, 위 코드에서 flag와의 부조화로 open_namei작업은 실패로 돌아가게 된다.(return -ETXTBSY)
p->executable은 현재 process로서 수행중인 file(binary)[78]의 inode이며, 그래서 여러 process가 이 file을 공유하고 있는 상태라면 file에 대한 쓰기작업을 할수없다.(return -ETXTBSY) 즉 현재 process로서 수행중인 binary에는 쓰기작업을 할수 없다. 이 변수는 exec sytem call에 의해 pro-gram이 수행될때 set된다. exec system call에 대해서는 7장 메모리 관리에서 다룬다.

3) truncate
open system call에 의해 전달된 O_TRUNC flag에 의해 file크기를 0로 만드는 것을 truncate라고 한다. ext2 화일시스템을 위한 notify_change함수는 없다.(super.c)

5.13 mount

5.13.1 개요

filesystem마다 file을 open, read, write를 하는 방법이 달라야 할것이다. 간단히 말해서 mount라는 작업은 바로 이것을 지정해 주는 것이다.
예를 들어 /mnt에 MS-DOS filesystem을 mount하였다고 가정하자.

```
# mount -t msdos /dev/hda1 /mnt
```

이 후 /mnt directory에 들어가면

```
# cd /mnt
```

78) bash process라면 disk상의 /sbin/bash file을 말한다.

이때부터는 MD-DOS를 위한 file operation함수 open, read, write 수행된다. 물론 /mnt directory를 빠져나오면, root filesystem type에 맞는 file operation함수가 수행될것이다.
초기에는 부팅시 sys_setup system call에 의해 '/' (root directory)에 filesystem이 mount된다.(mount_root함수)
지금부터 file operation을 set하는 과정을 살펴보자.

5.13.2 mount system call

mount system call의 가장 큰일은 mount 시킬려는 filesystem의 super block을 읽어들이는 것이다. 읽어들인 내용은 ext2_super_block(include/linux/ext2_fs.h)구조체에 들어간다.

```
1) struct super_block super_blocks[NR_SUPER];

   static struct super_block * read_super(dev_t dev,char *name,int flags,
                                    void *data, int silent)
   {
           /*  중   략  */

2)      check_disk_change(dev);
3)      s = get_super(dev);
        if (s)
                return s;
4)      if (!(type = get_fs_type(name))) {
                printk("VFS: on device %d/%d: get_fs_type(%s) failed\n",
                                          MAJOR(dev), MINOR(dev), name);
                return NULL;
        }
5)      for (s = 0+super_blocks ;; s++) {
                if (s >= NR_SUPER+super_blocks)
                        return NULL;
                if (!s->s_dev)
                        break;
        }
          /*  중   략  */

   }
```

1) 각 filesystem의 super block을 읽어들여, 그 내용을 담아두는 곳이다.

(NR_SUPER == 35)

2) port 3f7(floppy.c)을 읽어들이는 것에 의해 floppy가 바뀌었는지를 알수있다. 이것은 cdrom이나 floppy가 바뀜으로 해서 발생되는 buffer문제를 확인하기 위해 사용된다.
3) super_blocks 배열에서 mount시킬려는 filesystem에 대한 것이 이미 존재하고 있는지 check한다. 그렇다면 그냥 return.
4) mount 옵션중에 type이 있다. 즉 name은 'msdos', 'sysv'등에 해당한다. /fs/filesystems.c를 보면 지원되는 filesystem의 super block을 읽어들이는 함수가 나와 있다. 만약 MS-DOS filesystem을 mount시킬려고 한다면, msdos_read_super를 가진 배열항목 pointer가 return될 것이다.
5) super_blocks[]에서 비어있는 항목을 찾는다.

지금부터는 ext2 filesystem에서 DOS filesystem을 '*/mnt*' direcory에 mount시킨다고 가정한다.

```
# mount -t msdos /dev/hdb1 /mnt
```

이후 '/' directory와 '/mnt' directory에서 inode를 읽어들이는 함수는 다르다. 즉 '/' directory에서는 ext2_read_inode함수가 수행될것이고, '/mnt' 에서는 msdos_read_inode함수가 수행된다.

다음 코드는 msdos_read_super함수의 일부이다.

```
        s->s_op = &msdos_sops;
        if (!(s->s_mounted = iget(s,MSDOS_ROOT_INO))) {
```

msdos_sops함수들은 DOS filesystem내부에서 필요한 inode읽기등의 작업이 포함된다.

```
static struct super_operations msdos_sops = {
        msdos_read_inode,
        msdos_notify_change,
        msdos_write_inode,
        msdos_put_inode,
        msdos_put_super,
        NULL, /* write_super added in 0.96c */
        msdos_statfs,
        NULL    /*    remount    */
};
```

위 코드에서 s_mounted가 포함된 줄을 주목하자. 이 변수에는 mount시킨 filesystem의 root dire -ctory inode구조체 pointer가 들어가며, mount system call 뒷부분에서 mount시킨 directory inode의 i_mount에 보관된다.

다음 코드는 do_mount함수의 일부이다.

```
1)      if (dir_i->i_count != 1 || dir_i->i_mount) {
                iput(dir_i);
                return -EBUSY;
        }
        if (!S_ISDIR(dir_i->i_mode)) {
                iput(dir_i);
                return -EPERM;
        }
2)      if (!fs_may_mount(dev)) {
                iput(dir_i);
                return -EBUSY;
        }

        sb = read_super(dev,type,flags,data,0);
3)      if (!sb || sb->s_covered) {
                iput(dir_i);
                return -EBUSY;
        }
4)      sb->s_covered = dir_i;
5)      dir_i->i_mount = sb->s_mounted;
```

1)2)3) 우리는 mount명령어 수행중에 간혹 다음과 같은 에러를 접하게 된다.

```
# mount -t ext2 /dev/fd0 /mnt
mount: wrong fs type, /dev/hda1 already mounted, /mnt busy, or other error
```

1)은 /mnt directory가 사용중이다.
2)는 다른 process가 이 file system을 점유하고 있다.
3)은 wrong fs type(super block에 문제가 있거나)이거나 /dev/hda1이 이미 mount되어 있다.,

예를 들어 만약 A: 드라이브에 들어 있는 부팅디스켓을 mount시킬려고 하면 **super block이**

없다는 이유로 mount가 되지 않는다.

4) s_covered는 '/mnt' directory(**mount시킬 directory**)의 본래inode이다. 이 후 논의되겠지만 mo-nt에 의해 /mnt의 inode는 바뀌게 된다. 그래서 '/mnt' directory내에서 **부모 directory**인 '/'로 빠져나올때를 위해 저장된다.
다음은 lookup함수의 일부이다.

```
if (len==2 && name[0] == '.' && name[1] == '.') {  /* cd ..명령의 경우  */
      if (dir == current->root) {
              *result = dir;
              return 0;
      } else if ((sb = dir->i_sb) && (dir == sb->s_mounted)) {
              sb = dir->i_sb;
              iput(dir);
              dir = sb->s_covered;
              if (!dir)
                      return -ENOENT;
              dir->i_count++;
      }
```

5) mount된 상태에서 lookup함수를 사용하여 /mnt/command.com을 찾는다고 가정하자.
'/'에서 '/mnt'를 찾을때는 ext2 filesystem에서 찾아야겠지만, '/mnt'에서 'command.com'을 찾을때는 DOS filesystem에서 찾아야 한다.
lookup함수는 먼저 '/'에서 '/mnt'를 찾을 것이다. 이 때 iget함수가 사용되는데, 이곳에서 '/mnt'의 inode를 DOS의 root inode(i_mount)로 바꾼다. iget함수의 다음 코드를 보자.

```
if (crossmntp && inode->i_mount) {
        struct inode * tmp = inode->i_mount;
        tmp->i_count++;
        iput(inode);
        inode = tmp;
        wait_on_inode(inode);
}
```

이제 '/mnt'는 DOS의 root directory가 된다. 더불어 '/mnt'내에서는 inode[79]를 읽기위해

79) 사실 DOS에서는 inode table이 없으므로, disk에서 FAT(file allocation table) 바로 다음에

msdos_read_inode를 사용하게 되는것이다.

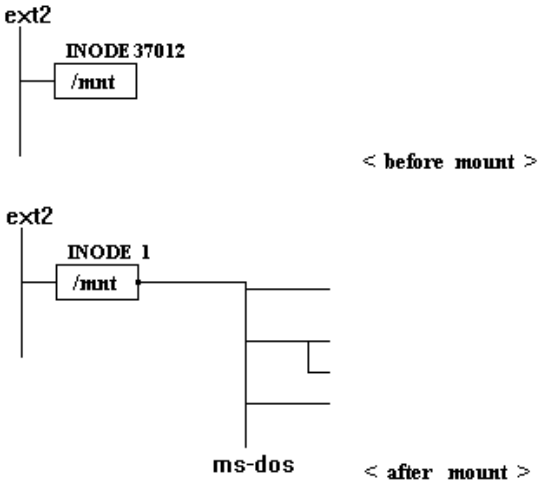
e2dump를 사용하여 mount에 의해 '/mnt' inode가 어떻게 바뀌는지를 보자.

```
# e2dump -n /mnt /dev/hda3

INODE 37012
File mode 40755 (directory)
Owner Uid 0 Group Id 0
File size 1024
Access time      : Sun Jul  2 22:59:59 1995
Creation time    : Thu Jan 26 16:10:40 1995
Modification time: Thu Jan 26 16:10:40 1995
Links count: 2
512-Blocks count: 2
Version: 1
Allocation:147745

# mount -t msdos /dev/hdb1 /mnt
# e2dump -n /mnt /dev/hda3

INODE 1
File mode 0 Owner Uid 0 Group Id 0
File size 0
Access time      : Wed Jan 25 21:06:13 1995
Creation time    : Wed Jan 25 21:06:13 1995
Modification time: Wed Jan 25 21:06:13 1995
Links count: 0
512-Blocks count: 0
Version: 0
```

/mnt의 inode 번호가 37012에서 1로 바뀌었다. DOS의 root inode의 inode번호는 1 이다. (msdos_fs.h)

```
#define MSDOS_ROOT_INO   1  /* == MINIX_ROOT_INO  */
```

존재하는 *root directory*를 읽게된다.



<그림5.6> mount direcotry의 inode변화

이제 '/proc'에 대해서도 e2dump를 사용해 보길 바란다. 역시 PROC_ROOT_INO인 inode번호 1을 발견할것이다. 참고로 ext2 filesystem 의 root inode번호는 2 (EXT2_ROOT_INO)이다.

inode에는 i_dev라는 항목이 있는데, 특정 filesystem이 존재하는 block device를 나타낸다. i_dev는 major번호와 minor번호를 가지고 있는데, 이들에 의해 device(보다 정확하게는 partition)가 시작하는 sector위치를 알수 있다. sector위치는 setup syste call이 BIOS정보를 이용하여 set한다. setup system call과 block device의 i_dev구조에 대해서는 9장 리눅스 파티션 에서 다룬다. 따라서 inode를 안다면 어떤 device에서 file을 읽고, 쓰는작업을 해야할지를 알수있게 된다. 다음 inode는 memory에 위치하는 inode(**memory inode**)이다. 뒷부분의 공용체(union)는 해당 filesyst-em의 disk inode를 담고있다. disk에서 읽어들인 disk inode는 이 공용체에 들어가게 된다.

```
struct inode {
        dev_t           i_dev;
        unsigned long   i_ino;
        umode_t         i_mode;
        nlink_t         i_nlink;
        uid_t           i_uid;
        gid_t           i_gid;
        dev_t           i_rdev;
        off_t           i_size;
        time_t          i_atime;
        time_t          i_mtime;
        time_t          i_ctime;
        unsigned long   i_blksize;
        unsigned long   i_blocks;
        struct semaphore i_sem;
        struct inode_operations * i_op;
        struct super_block * i_sb;
        struct wait_queue * i_wait;
        struct file_lock * i_flock;
        struct vm_area_struct * i_mmap;
        struct inode * i_next, * i_prev;
        struct inode * i_hash_next, * i_hash_prev;
        struct inode * i_bound_to, * i_bound_by;
        struct inode * i_mount;
        struct socket * i_socket;
        unsigned short i_count;
        unsigned short i_flags;
        unsigned char i_lock;
        unsigned char i_dirt;
        unsigned char i_pipe;
        unsigned char i_seek;
        unsigned char i_update;
        union {
                struct pipe_inode_info pipe_i;
                struct minix_inode_info minix_i;
                struct ext_inode_info ext_i;
                struct ext2_inode_info ext2_i;
                struct hpfs_inode_info hpfs_i;
```

```
                struct msdos_inode_info msdos_i;
                struct iso_inode_info isofs_i;
                struct nfs_inode_info nfs_i;
                struct xiafs_inode_info xiafs_i;
                struct sysv_inode_info sysv_i;
        } u;
};
```

5.14 kernel의 inode 관리

kernel이 inode를 관리하는 것은 buffer cache를 관리하는 방식과 유사하다.
새로운 file이 하나 생성되었다고 가정하자. 현재 이 file에 대한 inode는 없기 때문에 inode를 하나 할당해야한다. 이 때 *ialloc*함수가 사용된다.
ialloc함수는 먼저 inode bitmap에서 사용중이지 않은 inode번호(inode)를 구한다.(inode번호가 사용중이면 해당 bit가 set된다.) 그리고 구한 inode 해당 bit를 set하며, inode hash table에 그 inode를 끼워넣는다. inode hash table은 buffer hash table과 유사하다. 한번 disk에서 읽어들인 inode를 최대한 hash table에 보관하여 다시 읽어들이지 않기 위하여 사용된다.
당연히 방금 새롭게 할당된 inode도 또한 hash table에 들어가게 된다. 어떤 filesystem의 inode table에서 임의의 inode를 읽어들이기 위해서는 iget함수를 사용한다.

```
iget(* super block, inode number);
```

iget함수는 먼저 hash table에서 찾아보고 없으면, disk에서 inode를 읽어들인다. 읽어들인 inode는 hash table에 들어가게 된다. iget함수는 hash table뿐만 아니라 *inode free list*를 관리한다. inode free list는 disk inode를 읽어들일 물리memory공간(memory inode)을 고리로 연결하고 있다.
다음 iget함수의 코드를 보자.

```
struct inode * __iget(struct super_block * sb, int nr, int crossmntp)
{
            /*  중  략  */
1)      h = hash(sb->s_dev, nr);
repeat:
        for (inode = h->inode; inode ; inode = inode->i_hash_next)
```

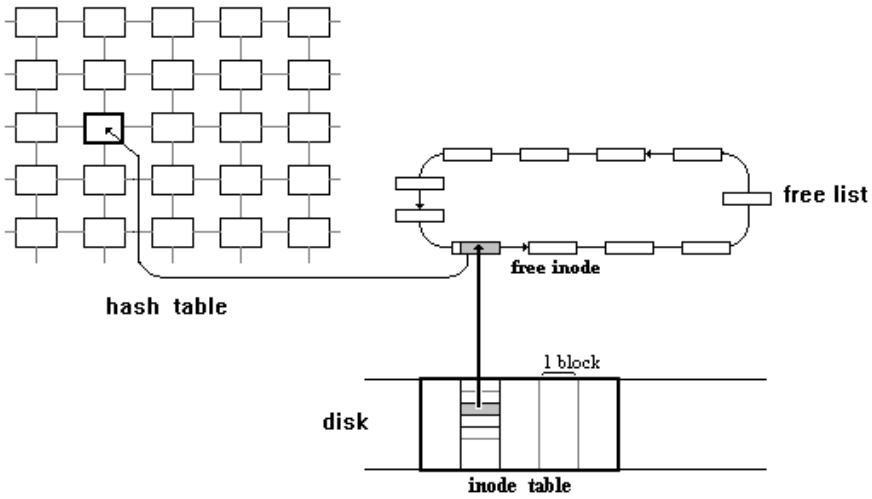
```
                if (inode->i_dev == sb->s_dev && inode->i_ino == nr)
2)                      goto found_it;
        if (!empty) {
                h->updating++;
3)              empty = get_empty_inode();
                if (!--h->updating)
                        wake_up(&update_wait);
                if (empty)
                        goto repeat;
                return (NULL);
        }
        inode = empty;
        inode->i_sb = sb;
        inode->i_dev = sb->s_dev;
        inode->i_ino = nr;
        inode->i_flags = sb->s_flags;
4)      put_last_free(inode);
        insert_inode_hash(inode);
5)      read_inode(inode);
        goto return_it;

found_it:
                /* 중    략 */
        return inode;
}
```

1) 먼저 hash table에서 찾는다.
2) hash table에서 발견.
3) hash table에서 발견하지 못했다. 놀고있는 memory inode를 구한다.
   memory inode는 grow_inodes함수에 의해 만들어지는데, get_free_page에 의해 page하나를 구해서 inode크기로 나눈다.

```
nr_free_inodes += PAGE_SIZE / sizeof(struct inode)
```

   이렇게 만들어진 memory inode들은 고리(inode free list)로 연결되어, disk에서 inode를 읽을 때 꺼내서 disk inode내용을 담는다.
4) 사용중인 memory inode는 free list의 마지막에 둔다.
5) disk inode를 읽어 memroy inode에 담는다.

그림<5.7>은 hash table과 inode free list를 설명하고 있다.
disk inode에서 올려진 memory inode는 free list의 마지막에 붙고, hash table의 고리에 들어가게 된다. free list에는 모든 memory inode가 있지만, hash table에는 사용중인 inode만이 있게 된다.



<그림 5.7> hash table과 inode free list

5.15 /proc

/proc directory는 가상 filesystem에 mount되어 있다. **/proc** directory에 보이는 file들은 실제 disk상에 존재하는 것이 아니다. disk상에는 단지 **/proc** directory만이 존재하며, 그 이후에는 "proc" super operation함수들, open, read, write system call등에 의해 kernel과 process관련정보들이 user에게 마치 실제 file들처럼 보이도록 kernel이 관리하고 있는 것이다. 그래서 **/proc** filesystem은 memory내에 존재하고 있다고 볼수있다.
/proc는 부팅중에 /etc/fstab file에 의해 mount되며, ps명령어가 **/proc**내의 정보를 이용하고 있다.

6장

프로세서 관리

6.1 서 론

리눅스에서 process를 관리하는 중요한 두 함수는 do_timer(sched.c)와 schedule(sched.c)이다. do_timer함수는 timer interrupt handler로서 1초에 100(HZ)번씩 수행되면서, 프로세서 관리를 하게된다. schedule함수는 timeslice등의 조건에 의해 task전환을 수행하는 scheduler이다.

6.2 timer interrupt handler 함수

6.2.1 tms 구조체와 itimerval, timeval 구조체

1) tms

```
struct tms {                    /* times.h  */
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
};
```

library함수인 *times*를 사용하면, tms구조체에 경과된 process 시간이 들어가게 된다. GNU syste-m에서는 clock_t는 long int형이다.
process는 user mode와 kernel mode를 번갈아 오가게 된다. 만약 read system call을 수행한다면 user mode에서 kernel mode로 진입했다가 다시 user mode로 나오게 된다. 이럴때 user mode와 kernel mode가 각각에서 얼마만큼의 CPU를 사용했는지에 대한 정보가 이 구조체에 들어가게 된다.

- **tms_utime**
  process에서 명령어를 수행시키기 위해 사용되는 CPU 시간.
  즉, 순수하게 process(**user mode**)에서 사용한 시간을 말한다.
- **tms_stime**
  process에서 kernel mode로 들어감으로써, kernel에 의해 사용된 시간.
- **tms_cutime**
  wait이나 waitpid에 의해 terminate되었음이 알려진 child들이 사용한 process 전체 tms_utime과, 부모 process의 tms_utime을 합한 시간.
- **tms_cstime**
  wait이나 waitpid에 의해 terminate되었음이 알려진 child들이 사용한 process 전체 tms_stime과, 부모 process의 tms_stime을 합한 시간.

02) timeval

time(time_t *result)함수는 result에 현재시각을 되돌린다. 현재시각은 *1970년 1월 1일 00:00:00 (epoch)*에서 경과된 시간이다. 그럼, kernel은 현재 시각을 어떻게 아는가?

time_init(time.c)함수에서 CMOS RAM을 읽어 현재 시각을 알게 된다.

```
xtime.tv_sec = kernel_mktime(&time);
```

kernel은 현재시각을 다음 구조체에 넣는다.

```
struct timeval {          /* time.h  */
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* microseconds */
};
```

- **tv_sec**
  epoch에서 경과된 지금까지 시간.
- **tv_usec**
  epoch에서 경과된 시간이 microsecond(1/1000000초)로 들어가게 된다.

3) alarm setting.

각 process는 3가지 독립된 timer를 가지고 있다.

① clock time을 재는 실시간 timer. 이 timer는 정해진 시간이 지나면 process에게 SIGALR-M signal을 보낸다.

② process 자체에 의해 user mode에서 사용된 CPU 시간을 재는 가상 timer. 이 timer는 정해진 시간이 지나면 process에게 SIGVTALTM signal을 보낸다.

③ process 자체에 의해 user mode에서 사용된 CPU 시간과 system call에 의해 사용된 시간을 재는 profiling timer. 이 timer는 정해진 시간이 지나면 process에게 SIGPROF signal을 보낸다.

```
struct  itimerval {           /* time.h  */
        struct  timeval it_interval;    /* timer interval */
        struct  timeval it_value;       /* current value */
};
```

- **it_interval**
  연속되는 두개의 SIGALRM signal사이의 간격이다. 이 간격에 의해 SIGALRM sig-nal이 계속해서 보내어지게 된다.
  이 값이 0이면, alarm은 단지 한번 보내어질것이다.
- **it_value**

첫번째 SIGALRM signal까지의 간격이다. 이 값이 0이면 alarm기능은 없다.

이 값들에 대해서는 이 후 계속 논의된다.

6.2.2 do_timer 함수

다음은 sched.c 코드에 나와있는 것들중에서 이 함수에서 사용되는 변수들이다.

```
long tick = 1000000 / HZ;               /* timer interrupt period */
volatile struct timeval xtime;          /* The current time */
int tickadj = 500/HZ;                   /* microsecs */

/*
 * phase-lock loop variables
 */
int time_status = TIME_BAD;     /* clock synchronization status */
long time_offset = 0;           /* time adjustment (us) */
long time_constant = 0;         /* pll time constant */
long time_tolerance = MAXFREQ;  /* frequency tolerance (ppm) */
long time_precision = 1;        /* clock precision (us) */
long time_maxerror = 0x70000000;/* maximum error */
long time_esterror = 0x70000000;/* estimated error */
long time_phase = 0;            /* phase offset (scaled us) */
long time_freq = 0;             /* frequency offset (scaled ppm) */
long time_adj = 0;              /* tick adjust (scaled 1 / HZ) */
long time_reftime = 0;          /* time at last adjustment (s) */

long time_adjust = 0;
long time_adjust_step = 0;
```

- 변수 tick에서 분모 1000000은 microsecond단위(timeval.tv_usec)이다. 즉, tick은 1/100초로서 interrupt handler의 수행주기가 된다.

- *volitile* type[80]
  최적화 compile시에 compiler는 자동변수값을 memory뿐만 아니라, 레지스터에도 써 넣는다. 이 때문에 longjmp나 interrupt handler수행후 변수값을 memory가 아닌, register에서 읽어들

80) Advanced Programming in the Unix Environment 178page

임으로써 setjmp가 수행될때나, interrupt handler가 수행될때 수행전의 값들이 그 후 변했더라도 변하지 않은 결과가 나타나는 수가 있다. 이것은 전역변수와 정적변수에서는 나타나지 않는 현상이다. volitile type은 register에는 값을 써 넣지 말라는 의미이다.
그래서, 지역을 크게 벗어나는 jump에도 자동변수값이 일관성을 유지하기를 바란다면 volatil-e type을 사용해야 한다.

- 여기서 PLL(phase locked loop) 변수들은 *adjtimex* system call을 사용하였을 경우에 timeva-l 구조체의 tv_usec값(microsecond)을 산출하기 위해 사용된다.[81] 여기서 우리는 이 system call이 사용되는 경우를 배제한다.

1) xtime.tv_usec값은 process에서 adjtimex system call이 호출되지 않는 한, 다음 코드에 의해 0, tick, 2*tick, 3*tick로 값이 증가한다. 그러다가 1초에 해당하는 1000000가 되면 다시 0가 되어 증가하기 시작한다.

```
xtime.tv_usec += tick + time_adjust_step(초기치 0);
```

2) 1초가 지나면 second_overflow 수행.
   ① time_adj값은 FINE_TUNE값이다.
   ② 660초마다[82] CMOS RAM의 시각을 갱신한다.
3) jiffies는 timer handler가 수행할때마다 1씩 증가한다.
   즉 1/100초에 해당한다.
4) calc_load
   단위시간당 load되어진 task수를 계산한다.[83]
5) process에 의한 시간(user mode)과 kernel에 의한 시간(system mode)을 계산한다.
   ① user mode
      VM_MASK는 가상 8086 mode를 사용할 경우을 말하는데, 이것은 process에 의해 *vm86* sy-stem call이 호출됨으로써 eflag에 set된다. selector rpl=3 이면 user level을 의미한다.
      가상 timer에 지정된 시간이 끝나면 SIGVTALRM signal을 보낸다.
   ② system mode
      task[0]가 수행중인 시간은 배제한다.
6) task 구조체의 **counter**는 바로 해당 process의 timeslice가 된다.
   timeslice는 schedule함수가 수행되는 주기를 말한다. 리눅스 초기 버젼(version 0.1)의 경우는 timer interrupt handler가 수행될때마다 schedule이 발생하였다. 즉, 1/100초마다 무조건 sche

---

81) 본인은 adjtimex system call에 대해 많이 알지 못한다.
PLL에 대한 기술적인 사항에 대해서는 관련서적을 참고하기 바란다.
82) Why 660?
83) /proc/loadavg 참고.

-duler가 수행되었던 것이다.[84]

counter값은 handler수행시 매번 1씩 감소하여 0이 되면 scheduler가 수행된다.(need_resched =1) 즉,counter값이 바로 timeslice에 해당한다. counter값은 schedule(sched.c) 함수에서 proce -ss의 **priority**(우선권)에 의해 결정된다.

다음은 schedule()함수 코드의 일부이다.

```
c = -1;
next = p = &init_task;
/* TASK_RUNNING 상태인것 중에서 counter값이 가장 큰것을 찾는다. 그러나
  init_task는 대상에서 제외된다. */
for (;;) {
       if ((p=p->next_task) == &init_task)
            goto confuse_gcc2;
       if (p->state = TASK_RUNNING && p->counter > c)
            c = p->counter, next = p;
}
confuse_gcc2:
/* 가장 큰 값이 0이라는 말이 된다. 즉, process들이 자신의 timeslice를 모
 두 사용하였다. 그래서 다시 timeslice를 할당받는다. 이 때 init_task도 할
 당받게 된다. init_task는 아직 덜 사용된 counter값을 가지고 있을수도 있
 다. 아무런 process가 수행되지 않으면 init_task만 counter를 가질 것이다.
 */
if (!c) {
        for_each_task(p)
             p->counter = (p->counter >> 1) + p->priority
}

/* counter값이 가장 큰 process가 next가 되므로, counter값은 process가 얼
마나 자주 수행되느냐를 결정함을 알수있다.
switch_to(next);  /* task 전환 */
```

counter값과 priority의 초기값은 각각 15 이다.(sched.h의 INIT_TASK 초기값 참고) fork시 자식 process의 초기counter는 반으로 준다. 그러나 초기값을 사용후 할당되는 count값은 prio -rity에 의해 15가 된다. 이들의 초기값은 *setpriority*와 *nice*등의 system call에 의해 바뀔수

84) MINIX에서는 1초에 10번 주기적으로 scheduler이 수행된다.

있다.

7) profiling timer에 지정된 시간이 끝나면 SIGPROF signal을 보낸다.
8) mark_bh(TIMER_BH)는 *timer_bh*라는 bottom half routine이 수행되도록 한다.
timer table에 있는 어떤 timer의 정해진 시간이 말소되면, timer_bh함수가 수행된다. timer table에는 다음과 같은 timer들이 들어 갈수 있다.

```
BLANK_TIMER     console screen-saver timer                  blank_screen()
BEEP_TIMER       console beep timer
RS_TIMER           timer for the RS-232 ports                  rs_timer()
HD_TIMER           harddisk timer                              hd_times_out()
HD_TIMER2          (atdisk2 patches)
FLOPPY_TIMER    floppy disk timer (not used right now)      floppy_shutdown()
SCSI_TIMER         scsi.c timeout timer                        scsi_main_timeout()
NET_TIMER          tcp/ip timeout timer
COPRO_TIMER     387 timeout for buggy hardware..             copro_timeout()
TAPE_QIC02_TIMER      timer for QIC-02 tape driver (it's not hardcoded)
MCD_TIMER     Mitsumi     CD-ROM Timer
SBPCD_TIMER        SoundBlaster/Matsushita/Panasonic CD-ROM timer
```

오른쪽에 이탤릭체로 나와있는 함수들은 timer함수들이다.
이들 timer의 말소시간이 정해지면 timer_active변수의 해당 bit를 set하고 timer_struct의 **expire-s**변수에는 정해진 시간이 들어가고, 그리고 fn(void)변수에는 그 시간이 지난후, 수행될 timer함수가 각각 들어가게 된다.

```
struct timer_struct {
        unsigned long expires;
        void (*fn)(void);
};

extern unsigned long timer_active;
extern struct timer_struct timer_table[32];
```

9) timer bottom half routine(timer_bh)
- timer_active에서 해당 bit clear

· 정해진 timer 관련함수 수행.

10) itimer_ticks는 scheduler수행시마다 0로 초기화된다. 즉, 이 값은 scheduler가 수행된 간격을 나타낸다.

다음은 *alarm* system call의 코드(sched.c)이다.

```
asmlinkage int sys_alarm(long seconds)
{
        struct itimerval it_new, it_old;

        it_new.it_interval.tv_sec = it_new.it_interval.tv_usec = 0;
        it_new.it_value.tv_sec = seconds;
        it_new.it_value.tv_usec = 0;
        _setitimer(ITIMER_REAL, &it_new, &it_old);
        return(it_old.it_value.tv_sec + (it_old.it_value.tv_usec / 1000000));
}
```

다음은 _setitimer함수 코드의 일부이다.

```
                case ITIMER_REAL:
                        if (j) {
                                j += 1+itimer_ticks;
                                if (j < itimer_next)
                                        itimer_next = j;
                        }
                        current->it_real_value = j;
        ==>             current->it_real_incr = i;
                        break;
```

alarm system call에 의해 it_real_value값이 정해지는데, 이 값이 0가 되면 scheduler에서 SIG-ALARM signal이 process에게 보내어 진다. it_real_incr는 itimerval 구조체에서 it_interval에 의해 결정되며, 이 값이 있어야 itimer_next값이 정해진다. 이 변수값은 SIGALRM signal을 보낸후, interval시간만큼이 흐르면

에 의해 scheduler를 기동시켜 다시 SIGALRM signal을 발하도록 한다.

```
if (itimer_ticks > itimer_next)   /*  do_timer함수 내에서 */
  need_resched =1;
```

<코드6.1>

```
if (ticks && p->it_real_value) {      /* scheduler 내에서 */
        if (p->it_real_value <= ticks) {
                send_sig(SIGALRM, p, 1);
                if (!p->it_real_incr) {
                        p->it_real_value = 0;
             ==>        goto end_itimer;
                }
                do {
                        p->it_real_value += p->it_real_incr;
                } while (p->it_real_value <= ticks);
        }
        p->it_real_value -= ticks;
        if (p->it_real_value < itimer_next)
                itimer_next = p->it_real_value;
}
end_itimer:
```

위에서 ==> 부분을 주목하자.

다음 signal발생이 정해지지 않으면, itimer_next값은 scheduler에 의해 초기화된 ~0를 유지한다. 이 값은 일반적으로 itimer_ticks보다 매우 큰 값이므로 위의 코드<6.1> 조건이 성립하지 않는다.

11) next_timer변수는 drivers/scsi/ncr5380.c 에서 정해진다. scsi방식에 대해서는 내용의 간결성을 위해 본서에서는 다루지 않는다.

6.3 scheduler(schedule 함수)

먼저 *select* system call에 대해 상기하자.

```
int select(int nfds, fd_set *read_fds, fd_set *write_fds, fd_set *except_fds,
          ,struct timeval *timeout)
```

select함수는 구체화된 file descriptor들의 set에 행위가 가해졌을때(인터럽트가 발생되었을때)까지, 또는 timeout시간이 만료될때까지 이 함수를 호출한 process를 일시 중지시킨다.

```
end_itimer:
                if (p->state != TASK_INTERRUPTIBLE)
                        continue;
                if (p->signal & ~p->blocked) {
                        p->state = TASK_RUNNING;
                        continue;
                }
                if (p->timeout && p->timeout <= jiffies) {
                        p->timeout = 0;
                        p->state = TASK_RUNNING;
                }
```

위의 코드는 select함수를 사용하는 process의 경우처럼 TASK_INTERRUPTIBLE인 체로 sleep하고 있는 process에게만 해당된다. interrupt가 발생하면 signal이 전달될 것이고 이어서 process는 잠에서 깨어나 TASK_RUNNING 상태가 된다. p->timeout은 select에 의해 정해지며, 시간이 말소되면 역시 TASK_RUNNING 상태가 된다.

6.4 TASK 전환

task를 전환(switching)하는 방법에는 call명령을 사용하는 방법과 jmp명령을 사용하는 방법이 있다. 리눅스에서는 jmp명령을 사용하므로, jmp명령으로 task를 전환하는 절차를 알아보자.

```
#define switch_to(tsk) \
__asm__("cmpl %%ecx,_current\n\t" \   /* 전환할려는 task가 current task는 아닌지.. */
        "je 1f\n\t" \                 /* if so, Not switch */
        "cli\n\t" \                   /* switching 하는동안 interrupt 안 걸리게 */
        "xchgl %%ecx,_current\n\t" \  /* current(현재 process를 가리킴)변수값을 바꾼다. */
```

```
"ljmp %0\n\t" \                 /* GDT내의 TSS[해당process]로 jump.
"sti\n\t" \                        &tsk->tss.tr이 TSS descriptor를 가리킴에 주목 */
"cmpl %%ecx,_last_task_used_math\n\t" \
"jne 1f\n\t" \
"clts\n" \
"1:" \
: /* no output */ \
:"m" (*(((char *)&tsk->tss.tr)-4)), \
 "c" (tsk) \
:"cx")
```

윗 코드에서 보다시피 단지 해당 process의 TSS descriptor로 jump함으로써 task switching이 이루어 진다.

▶ call명령어에 의해 task 전환을 할 경우, iret명령어에 의해, 전환을 일으킨 process로 되돌아갈수 있다. 그때는 eflag에 NT bit가 set되어 있어야 한다.(sched_init에서 NT bit를 clear했음을 기억하자.)

task전환시 processor(CPU)는 다음과 같은 과정을 거친다.

① task변경이 유효한지 특권 level을 확인한다.
② 현재 task상태를 해당 TSS(task_struct->tss_struct)에 저장한다.
③ 전환할려는 process의 TSS descriptor의 *selector*(위 코드에서 *&tsk->tss.tr*)을 task register에 넣는다.
④ 새로운 task의 상태를 TSS로부터 load한후 실행한다.
TSS에는 process의 실행이 중단된 지점의 pointer정보도 가지고 있다.

참고로 TSS를 <표6.1>에 나타내었다. task_struct는 kernel이 정한것이지만, TSS(tss_struct)구조는 processor에 의해 정해져 있다. task register에 이 구조체를 위한 selector를 load함으로써 tss_struct의 내용을 process가 관리하게 된다. 그러나, tss_struct에서의 모두가 processor에 의해 관리되는 것은 아니다. 표<6.1>에서 보다시피 tss_struct의 io_bit_map[]까지만이 processor에서 명시한 부분이다.
이상의 일련의 과정이 ljmp명령만으로 이루어진다.[85]
위 코드중에서 다음을 보자.

---

85) ljmp(far jump) 명령어는 보호모드에서 task switch를 위해 사용될수 있다.
80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 460page 명령어 일람.

```
        "cmpl %%ecx,_last_task_used_math\n\t" \
        "jne 1f\n\t" \
        "clts\n" \
```

task전환이 이루어지면 processor는 cr0에서 자동적으로 TS bit를 set한다.[86] *TS bit가 set되면* coprocessor 명령어가 수행될때, **coprocessor not available**예외[87]가 발생한다. 이 곳에서 clts 명령어로 TS bit를 clear시키고, 마지막으로 coprocessor를 사용한 task의 TSS에 현재 coprocessor에 대한 정보를 보존한다. 그리고 새 task의 coprocessor에 대한 정보를 coprocessor의 stack에 load하고 last_task_used_math변수를 current로 한다. (syscall.S의 device_not_available부분 참고)
만약에 새 task가 coprocessor를 마지막으로 사용한 task라면, coprosessor의 stack에 있는 정보는 바뀌지 않고 보존되어 있을 것이므로 clts명령을 수행시켜 예외가 발생하지 않도록 한다.[88]

▶ 286급 이하기종을 위한 TSS16의 구조는 TSS32의 일부분으로 이루어진다.[89]

6.5 debug register

debug register들을 0로 초기화한다.

```
#define loaddebug(register) \
              __asm__("movl %0,%%edx\n\t" \
                      "movl %%edx,%%db" #register "\n\t" \
                      : /* no output */ \
                      :"m" (current->debugreg[register]) \
                      :"dx");
```

debug란 process가 진행되는 중에 원하는 명령어나 data지점의 정보를 얻기위해 그 지점에서 int-errupt(예외) handler를 수행한다. 그러기 위해서는 원하는 지점에서 일시 중지시켜 예외 handle

86) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 의
    3.8.2 Saving processor extension state on a task switch
87) 리눅스에서는 syscall.S의 device not available code
88) 80386 프로그램 입문-교학사 235page 코프로세서 예외

31 23	15 7	0	
I/O 맵 베이스	00000000	0000000	T 64
00000000 00000000	E	D T	60
00000000 00000000	G	S	5C
00000000 00000000	F	S	58
00000000 00000000	D	S	54
00000000 00000000	S	S	50
00000000 00000000	C	S	4C
00000000 00000000	E	S	48
E	D I		44
E	S I		40
E	B P		3C
E	S P		38
E	B X		34
E	D X		30
E	C X		2C
E	A X		28
E F L	A G S		24
E	I P		20
C	R 3		1C
00000000 00000000	S	S 2	18
E S	P 2		14
00000000 00000000	S	S 1	10
E S	P 1		C
00000000 00000000	S	S 0	8
E S	P 0		4
00000000 00000000	이전 TSS	로의 복귀	0

<표6.1> TSS32 구조

-r가 수행되도록 하여야 하는데, 일시 중지되는 지점을 breakpoint(실행정지점) 라고 한다.

많은 debugger(DOS의 debug 포함)들은 다음과 같은 8086계열에서 제공하는 3가지 방법을 제공한다.[90] 이들은 software debugger를 만들수 있도록 processor가 제공하는 기능들이다.

- single step.
- INT 3 을 사용.
- breakpoint(실행정지점) 레지스터 사용.

1) single step

eflag에서 TF(trap flag) bit를 set함으로써, 한 명령어가 수행될때마다 debug예외 handler가 수행된다.

---

89) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 2.11 Task switches

90) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 2.16 Software debug features

그러나, 이 방법의 문제는 debug handler가 아닌 다른 에러 handler가 수행되는 경우, 에러ha-ndler코드도 또한 single step이 걸리는 수가 있다.

2) INT 3 사용.

int 3은 trap을 초기화하는 과정에서 언급했지만 *trap* 명령어를 가리킨다. breakpoint로 하고자 하는 명령어를 *trap* 명령어로 대치시킨다. 이렇게 함으로써 그 지점에서 INT 3 handler를 수행시킬수 있다.

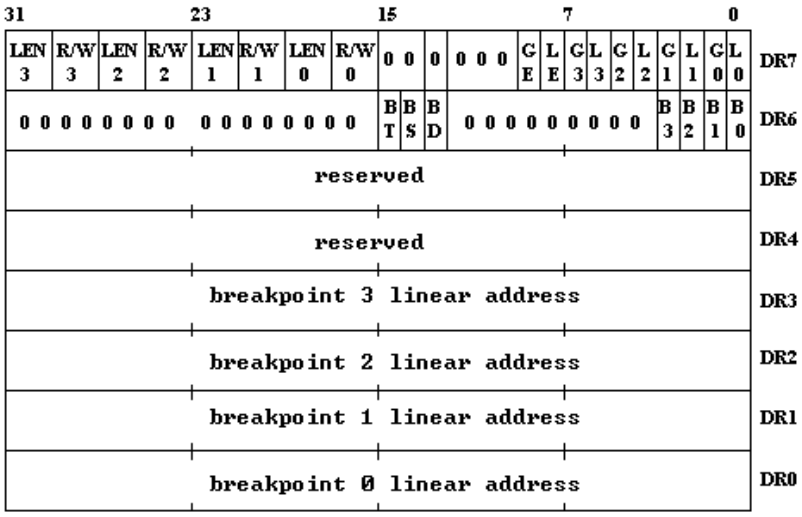
그러나, 이 방법으로 명령어 지점을 breakpoint로 할수는 있지만, data지점에서 중지시킬수는 없다. 그리고, ROM내의 명령어는 바꿀수가 없어, ROM 명령어 수행중에 break를 걸수는 없다.

따라서, 원하는 breakpoint를 임의로 지정할수 있는 방법이 요구된다. 이러한 요구에 부응하여 break pointer지정 register가 사용된다.

3) debug register

debug register에는 breakpoint register, debug status register, debug control register등의 3종류로 이루어져 있다. 이들은 386급 이상에서 사용가능하다.

그림<6.1>에 이들 register들을 나타나 있다.



<그림6.1> debug register

그림<6.1>에서 0 는 INTEL사에 의해 예약된 부분이다.

DR0, DR1, DR2, DR3는 breakpoint register들이다. DR4와 DR5는 예약되어 있으며, DR6은 d-ebug status register이고, DR7은 debug control register이다.
이것으로써 우리는 4개의 breakpoint번지(선형번지)를 지정할수 있으며, 각 breakpoint에 대한 조건을 DR7에 의해 정할수 있다. R/W bit에 의해 명령실행시에 실행정지점이 유효한지, data를 써넣는 것이 가능한지 등이 결정된다.
DR6에 의해서 예외발생시 상태를 알수 있다. B0,B1,B2,B3에 의해 DR0,DR1,DR2,DR3중에서 어느 breakpoint 예외가 발생했는지 알수 있다. BT,BS,BD는 어떤 debug 예외가 발생했는지를 알수있게 한다. TSS의 T(trap) bit가 set되면, task전환시 예외가 발생하고, BT가 set된다. single step 예외가 발생하면 BS가 set되며, debug register를 사용중에 또 하나의 명령이 이 register에 접근하면 BD가 set된다.
표<6.2>에 DR6의 각 bit들이 set되는 조건을 요약하였다.

상태 bit	bit set 조 건
BT	Because of Task switch
BS	Because of Single step
BD	Because of Debug register access
B3	Because of DR3 match
B2	Because of DR2 match
B1	Because of DR1 match
B0	Because of DR0 match

<표6.2> DB6 bit들의 set 조건

6.6 process 상태

task상태는 다음과 같이 6가지로 정의되어 있다.(sched.h)

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             3
#define TASK_STOPPED            4
#define TASK_SWAPPING           5
```

1) TASK_RUNNIG

process가 CPU를 할당받을수 있는 상태이다. task 전환은 이들 process를 대상으로 한다.
즉, TASK_RUNNING상태는 2가지가 있는데, CPU를 할당받은 상태와 할당받기위해 기다리고 있는 상태가 있다.

2) TASK_INTERRUPTIBLE
이것은 수면상태이며, interrupt에 의해 signal을 받아 TASK_RUNNING상태가 될수 있다. interrupt를 기다리는 경우에 주로 사용된다.

3) TASK_UNINTERRUPTIBLE
이것은 수면상태이며, interrupt의 영향을 받지 않는다.
이 상태는 주로 어떤 자원(resource)을 기다릴때, add_wait_queue함수이후에 set되며, 이어서 scheduler를 수행한다. 그리고, *fork* system call에서 자식 process를 생성하는 과정에서 이 상태가 되며, 완전히 생성후 자식 process는 TASK_RUNNING상태가 된다.

4) TASK_ZOMBIE
자식 process를 생성후 부모 process는 *wait* system call에 의해 자식이 종료되기를 기다린다. 자식이 종료할때 *exit* system call(kernel/exit.c내의 do_exit함수 참고)은 자식 process에게 할당된 메모리공간을 해제시키고, error code와 자식의 PID를 부모에게 전달한다. 하지만 이 두가지 정보와 함께 process table에는 여전히 자식 process가 할당되어 있다. 이때 자식 proc-ess는 **좀비(*zombie*)**상태가 된다.
부모 process의 *wait* system call은 좀비 상태인 자식 process가 없는지 확인하고, 자식 proce-ss를 process table에서 제거시킨다. 그런데, 만약에 부모 process가 *wait* system call에 의해 자식 process가 종료되는 것을 기다리지 않는다면, 그리고, 자식 process가 종료된다면 자식 process는 좀비상태가 된다.(여전히 process table을 차지한 체)[91] 그래서, ps로 확인하면

```
# ps
 PID TT STAT  TIME COMMAND
5940 P3 S     0:00 a.out
5941 p3 Z     0:00 <defunct>   <==좀비 상태
5942 P3 S     0:00 sh -c ps
5943 P3 R     0:00 ps
```

위에서 보다시피 process table을 차지하고 있음을 알수 있다.
자식 process가 죽기전에 먼저 부모 process가 죽으면 자식 process의 부모 process는 init da-emon이 된다. init daemon은 waitpid함수를 사용함으로써, 자식 process가 좀비 상태가 되지 않도록 한다.[92] 다음은 init daemon의 코드중에서 SIGCHLD handler이다.

---

91) UNIX 운영체제의 설계 - Maurice J.Bach 7-4. 프로세서의 종료대기
92) Advanced Programming in the Unix Environment 196page

```
void chld_handler()
{
  CHILD *ch;
  int pid, st;

  /* Find out which process(es) this was (were) */
  while((pid = waitpid(-1, &st, WNOHANG)) != 0) {

        ..........................................

  }
}
```

※ zombie를 이해하기 위한 또 한가지 방법.

부모 process는 자식 process가 죽으면 장례를 치루어주어야 할 의무가 있다. 그 방법이 wait과 waitpid함수이다. exit에 의해 자식 process는 죽었지만, 그는 호적에 여전히 등록되어 있는 것이다. 그래서, 부모 process는 wait함수에 의해 자식 process의 유품을 정리하고, 사망신고와 함께 장례식을 치루어야 한다.

만약에 부모 process가 먼저 죽는다면, init daemon이 양육권을 가지게 되고, 그 후 자식 process가 죽으면 그가 대신해서 장례를 치루게 되는 것이다. 자식 process가 죽었을때 부모 process가 생존하지만, 장례를 치루어줄 의사가 없다면 어떻게 될것인가? (즉 부모 process가 전혀 wait(또는 waitpid)함수를 사용하지 않는다면) 그때는 두가지 경우가 발생한다.

첫번째 경우는 이러하다.

fork시 부모 process는 곧 wait함수를 실행시켜 자식 process의 생존여부를 계속 확인하는 방법도 있지만, 어차피 SIGCHLD signal로서 사망통보를 받기때문에 그 때 signal handler에 의해 wait함수를 실행시켜도 된다. 그런데 부모 process가 다음과 같이 사망통보를 무시한다는 의사를 밝히는 경우라면,

```
signal(SIGCHLD, SIG_IGN);
```

정부(kernel)가 나서서 장례를 치루어주게 된다.(do_signal함수에 의해)

```
asmlinkage int do_signal(unsigned long oldmask, struct pt_regs * regs)
{
```

```
.....................................................................

    if (sa->sa_handler == SIG_IGN) {
            if (signr != SIGCHLD)
                    continue;
            /* check for SIGCHLD: it's special */
            while (sys_waitpid(-1,NULL,WNOHANG) > 0)
                    /* nothing */;
            continue;
    }
.....................................................................

}
```

두번째 경우는, 부모가 장례를 치루어줄 의사도 없으면서, 정부에게 그 뜻 또한 밝히지 않는 경우로서 이때는 정부도 어쩔수가 없다. 자식 process는 zombie[93]상태로 부모가 죽을때까지 기다려야 한다. 부모가 죽을때 init 에게 다른 자식 process들처럼 zombie도 같이 넘겨지며 비로소 init에 의해 극락왕생할수있게 되는 것이다.

```
NORET_TYPE void do_exit(long code)
{

.....................................................................................

        while ((p = current->p_cptr) != NULL) {   /* 자식 process들을 찾는다. */
                current->p_cptr = p->p_osptr;      /* 형제 process들을 찾는다. */
                p->p_ysptr = NULL;
                p->flags &= ~(PF_PTRACED|PF_TRACESYS);
                if (task[1] && task[1] != current)
                        p->p_pptr = task[1];      /* init에게 zombie도 넘겨진다. */
                else
                        p->p_pptr = task[0];
                p->p_osptr = p->p_pptr->p_cptr;
                p->p_osptr->p_ysptr = p;
                p->p_pptr->p_cptr = p;
                if (p->state == TASK_ZOMBIE)   /* 현재 zombie의 부모는 init이다. */
                        notify_parent(p);          /* init에게 CHLD signal 보낸다. */
```

93) 죽었다가 살아난 시체를 zombie라고 부른다.

```
        ..........................................................
}
```

5) TASK_STOPPED
   user의 ctrl-Z나 SIGSTOP signal에 의해 process는 이 상태가 된다.
   signal을 받아 TASK_RUNNIG상태가 될수 있다.
6) TASK_SWAPPING
   메모리의 고갈로 process가 swap out(hard disk에 write)된 상태를 나타낸다.

6.7 기타 scheduling 관련함수

6.7.1 sleep 함수

```
void interruptible_sleep_on(struct wait_queue **p)
{
        __sleep_on(p,TASK_INTERRUPTIBLE);
}

void sleep_on(struct wait_queue **p)
{
        __sleep_on(p,TASK_UNINTERRUPTIBLE);
}

static inline void __sleep_on(struct wait_queue **p, int state)
{
        unsigned long flags;
        struct wait_queue wait = { current, NULL };

        if (!p)
                return;
        if (current == task[0])
                panic("task[0] trying to sleep");
        current->state = state;
        add_wait_queue(p, &wait);
        save_flags(flags);
```

```
        sti();
        schedule();
        remove_wait_queue(p, &wait);
        restore_flags(flags);
}
```

- init task는 sleep하지 않으며, stop하지도, 종료되지도 않는다.
- process의 상태를 TASK_UNINTERRUPTIBLE, TASK_INTERRUPTIBLE 상태로 만들고 wait queue에 현재 process를 넣은후 scheduling한다. 그때부터 TASK_RUNNING 상태가 될때까지 이 process로 task전환이 이루어지지 않으므로, 자연히 **수면상태**가 된다.
  이 후 wakeup함수에 의해, process의 상태가 TASK_RUNNING상태가 되면, task 전환에 의해 process가 CPU를 할당받게 되고, wait queue에서 제거된다.

6.7.2 wakeup 함수

```
void wake_up(struct wait_queue **q)
{
        struct wait_queue *tmp;
        struct task_struct * p;

        if (!q || !(tmp = *q))
                return;
        do {
                if ((p = tmp->task) != NULL) {
                        if ((p->state == TASK_UNINTERRUPTIBLE) ||
                            (p->state == TASK_INTERRUPTIBLE)) {
                     ==>        p->state = TASK_RUNNING;
                                if (p->counter > current->counter)
                                        need_resched = 1;
                        }
                }
                 /*  이 곳에서 wait queue 상태 display */
        } while (tmp != *q);
}
```

wake_up함수는 wait queue에서 수면을 취하고 있는 process들을 깨워 CPU를 할당받을수 있도록 해준다.
단지 TASK_RUNNING 상태로 만들어 줌으로써 task전환대상이 되도록 한다. 이 함수는 잠자고

있는 process이면 모두 깨울수 있다. 그러나, wake_up_interruptable함수는 TASK_INTERRUPTI-BLE 상태의 process만 깨운다.
swap된 process(process의 상태가 TASK_SWAPPED)는 깨울수 없음을 주목하자. 즉, TASK_RUNNING상태가 되려면 메모리에 있어야 한다.

7장

메모리 관리

7.1 kmalloc 함수 (mm/kmalloc.c)

7.1.1 secondary_page_list

리눅스는 메모리를 할당할때 page단위(4k)로 한다. 사용가능한 page는 secondary_page_list와 free_page_list에 넣어두었다가 *get_free_page*함수(mm/swap.c)에 의해 필요한 page를 획득한다. get_free_page는 획득한 사용가능page의 pointer를 return한다. page를 다 사용하고 돌려놓을때는 다시 이 두가지 list에 넣게 되는데 먼저 secondary_page_list에 넣고, secondary_page_list가 다 차면, 그 다음에 free_page_list에 넣는다. get_free_page에 의해 획득할때는 먼저 free_page _list에서 구하고, 이 곳이 비어있으면 **경우에 따라** secondary_page_list에서 구하기도 한다. 그러한 *경우*에 대해서는 다음의 priority에 의해 설명된다. kmalloc에 parametor로 전달되는 priority에는

4가지가 있다.(mm.h)

```
#define GFP_BUFFER      0x00
#define GFP_ATOMIC      0x01
#define GFP_USER        0x02
#define GFP_KERNEL      0x03
```

get_free_page에 따르면, GFP_BUFFER로 page를 구할때는(즉 buffer를 할당할려고 할때), first_page_list에서만 구하고, 구할수 없으면 포기한다. GFP_ATOMIC의 경우는 자동적으로 (automati-cally) first_page_list에서 실패하면, secondary_page_list에서 구한다. GFP_USER, GFP_KERNE-L의 경우는 두가지가 구분이 없다.[94] 이들 둘은 free_page_list에서 구하지 못하면, memory에서 다른 process의 일부를 swap-out 시켜 그 process가 사용하는 영역을 빼앗아서 (메모리를 해방시켜서,*stolen*) 원하는 page를 획득할려는 시도를 한다. 그래도 실패하면, secondary_page_list에서 구한다.
interrupt가 수행중에 kmalloc이 호출되었다고 가정하자. 이때 kmalloc에 전달되는 priority가 자동모드(GFP_ATOMIC)가 아니면, 자동mode로 바꾸어서 secondary_page_list에서 빠른 시간내에 page를 구할수 있도록 **특혜**를 받게된다. 만약에 이러한 특혜가 주어지지 않는다면, interrupt handler수행중에 stolen과정에서, 잠들어 버릴수가 있다.
다음 code는 kmalloc의 일부이다. 이 코드는 get_free_page내에도 또한 존재한다.

kmalloc의 맨 앞 코드를 보면,

```
if (intr_count && priority != GFP_ATOMIC) {
        printk("kmalloc called nonatomically from interrupt %08lx\n",
                ((unsigned long *)&size)[-1]);
        priority = GFP_ATOMIC;
}
```

7.1.2 kmalloc code 수행과정

kmalloc 코드는 다음과 같은 순으로 진행된다.

1) interrupt handler수행중 kmalloc호출시 page획득에 대한 자동 mode로 바꾼다.

94) 본인은 구분할수 없었다.

2) get_order
size_descriptor 배열에서 적당한 block size를 구한다. 최대 block크기는 배열의 sizes[7]에 해당하는 4080(kmalloc에 의해 한번에 할당가능한 size최대값)이다. 만약에

```
kmalloc(1000,GFP_KERNEL)
```

로서 메모리를 할당한다고 가정하자. blocksize가 1000인 배열은 없으므로 이것은 sizes[5]에 해당한다. 즉 1020크기를 할당하게 된다. 여기서 return값인 order는 5이다.

3) tries값은 4이다. 메모리를 할당하는데 실패하면, 3번 더 재시도가 가능하다는 것이지만, 실패하면 바로 return하므로써, 코드의 주석에서도 언급한것처럼, 재시도는 없는 것으로 보인다.

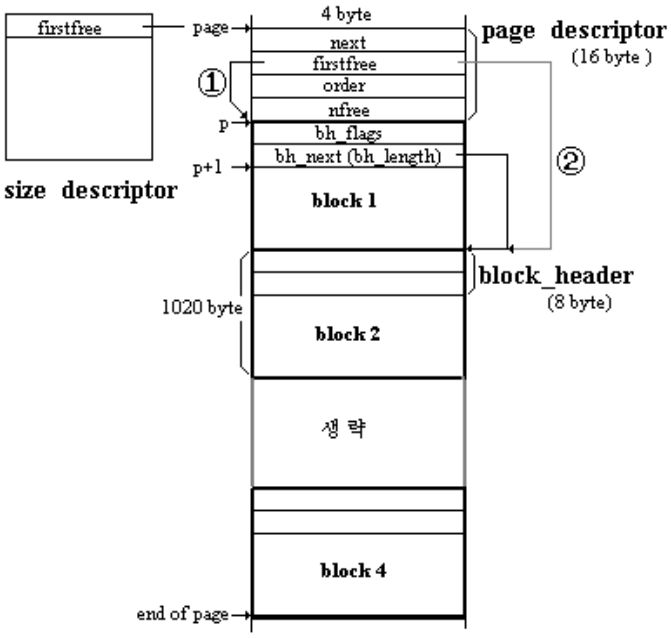
4) 처음에는 sizes[order].firstfree값은 없는 것으로 하자.

5) get_free_page함수로 page획득.

6) 획득한 page를 block으로 구분하고, page맨 앞에 page descriptor를, block맨 앞에는 block he-ader을 써 넣는다. 그림<7.1>에는 blocksize가 1020인 경우를 예로 들었다.
1 page에 의해 4개의 block이 할당된다. 그러나 실제 사용은 처음 1개 block만 사용된다.(1000 byte을 요구하였으므로)
다음 구조체는 kmalloc으로 할당된 물리영역의 list를 유지하기 위해 사용된다.

```
struct page_descriptor {
        struct page_descriptor *next;
        struct block_header *firstfree;
        int order;
        int nfree;
 };
```

7) tries에 의해 while loop는 다시 시작된다. 이때에는 size[order].firstfree에 page시작번지값이 들어있다. block 1의 bh_flag를 사용하고 있다는 표시로 바꾸고(MF_USED), 사용가능 block 수(page->nfree)를 1 줄인 다음, page descriptor에서 firstfree pointer를 다음 block으로 옮긴다.(그림<7.1>에서 ① → ②로)
이로서 첫번째 block이 할당된 것이다. return되는 pointer는 p+1이다.



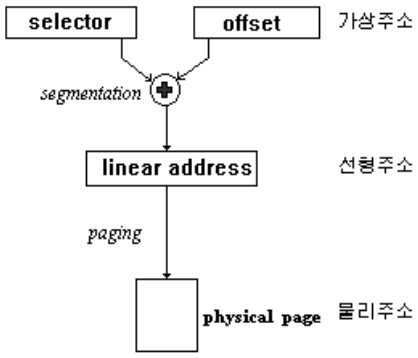
<그림 7.1> order 5인 경우

8) 만약에 이 후에 다시 508<size≤ 1020범위의 할당을 위해 kmalloc이 호출된다면, 이미 앞에서 할당한 page의 다음 block(block 2)을 사용할것이다. 그러나, 그 범위밖의 size로 할당을 요구하면, kmalloc은 다시 page를 획득하여, block을 나누는 작업을 해야 할것이다.

7.2 paging

paging은 386이상기종에서 지원되는 memory 관리체계이다. 이 paging에 의해 적은 양의 실메모리(physical memory)로 마치 매우 많은 양의 memory를 보유하고 있는 것처럼 보이게 한다.
리눅스는 processor가 제공하는 이 paging을 적극적으로 사용하고 있다. 또한 paging은 real mode에서는 사용할수 없으며, 보호모드(protected mode)에서만 사용가능하다.

processor에서 제공되는 보호모드에서의 memory관리체계는 **segmentation**과 **paging**으로 두가지가 제공된다. segmentation은 보호모드에서 필수이지만 paging은 그렇지 않다. 그러나 리눅스를 비롯한 대부분의 system에서는 paging을 사용하는데, 먼저 두가지의 번지지정방식을 간략하게 알아보자. 그림<7.2>는 가상주소, 선형주소, 물리주소의 변천과정을 나타내고 있다.



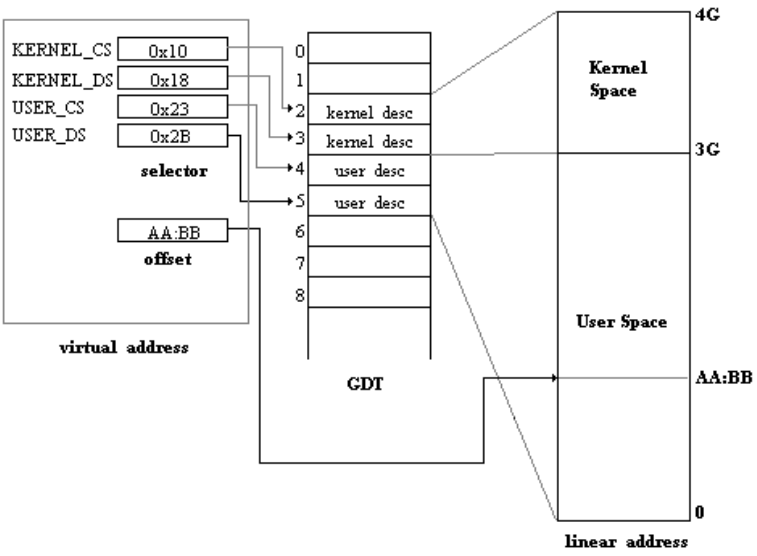
<그림 7.2> 주소 변천과정

7.2.1 segmention

보호모드에서는 DOS에서처럼 사용자 process가 물리 memory를 함부로 건드리지는 못한다. 즉, 보호모드에서는 4가지 특권레벨을 사용하는데 0,1,2,3이 그것이다. 2.4.3절의 15)에서 GDT를 setup한 것을 기억할 것이다.

실 모드에서는 segment와 offset으로 원하는 물리메모리의 한 지점을 지적할수 있었다. 그러나, 보호모드에서는 selector와 offset을 이용하여야 하며, 이것을 논리주소(logical address), 또는 가상주소(virtual address)라고 부른다. selector는 kernel과 user process를 위한 선형공간(linear address)을 만들어내는 GDT내의 descriptor를 가리키고 있다.

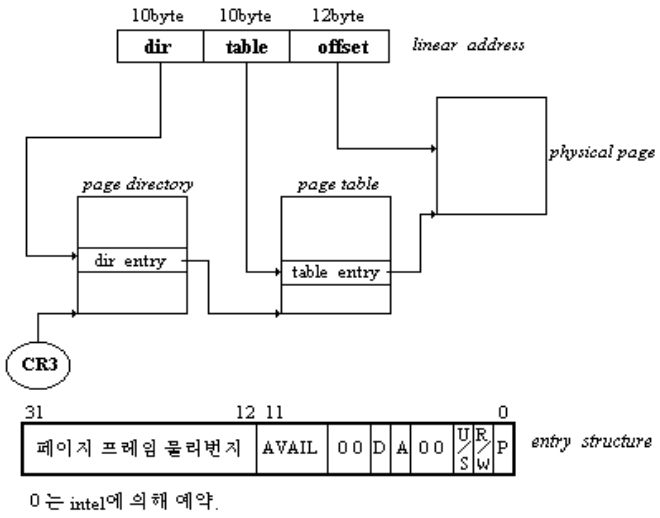
리눅스에서 선형주소는 모두 4 GB(32 bit로 지정할수 있는 최대번지)로 이루어져 있으며 처음 3 GB를 **user 선형주소공간**으로 사용하고, 나중 1 GB 를 **kernel 선형주소공간**으로 사용하고 있다. 그림<7.3>에서 가상주소가 선형주소로 바뀌는 과정을 보여주고 있다. 만약 paging을 사용하지 않는다면 이 선형주소는 물리메모리주소가 된다. 그때에는, 4 GB가 아닌 자신의 system에 장착된 실 memory크기를 선형주소 공간으로 잡아야 할것이다. 4 GB라는 대규모의 선형주소공간은 paging에 의해 이루어진다.



<그림7.3> 가상주소와 선형주소

7.2.2 paging 개요

paging system은 **page directory**와 **page table**이라는 2개의 table에 의해 이루어진다. 이들은 각각 한 page크기로 이루어져 있으며, 그림<7.4>와 같은 방법으로 선형주소를 통한 물리주소에 접근한다.



<그림7.4> 선형주소와 물리주소

그림<7.4>의 entry structure는 page directory와 page table의 entry 모습을 나타내고 있다.
물리번지는 20개 bit에 의해 정해지지만, 실제번지는 아래 12 bit가 0로 채워진 값이 된다. 따라서, page frame 물리번지는 1,2,3,4이지만 0x1000, 0x2000, 0x3000과 같이 4K단위(1page)로 이루어져 있다고 보면된다. 이것은 리눅스가 물리메모리를 사용하는 방법이기도 하다. paging system은 물리메모리를 사용할때는 항상 page단위로 사용하게 된다.

7.2.3 리눅스에서의 paging 사용

리눅스에서의 선형주소는 다음 두개의 매크로(include/linux/page.h)에 의해 이루어진다. 이들 매크로는 선형주소값(매크로에서 address)이 주어지면 page directory entry위치와 page table entr-y위치를 구한다.

```
#define PAGE_DIR_OFFSET(base,address)  ((unsigned long*)((base)+\
  ((unsigned long)(address)>>(PAGE_SHIFT-SIZEOF_PTR_LOG2)*2&PTR_MASK&~PAGE_MASK)))
                        /* to find an entry in a page-table */

#define PAGE_PTR(address)                \
  ((unsigned long)(address)>>(PAGE_SHIFT-SIZEOF_PTR_LOG2)&PTR_MASK&~PAGE_MASK)
                        /* the no. of pointers that fit on a page */
```

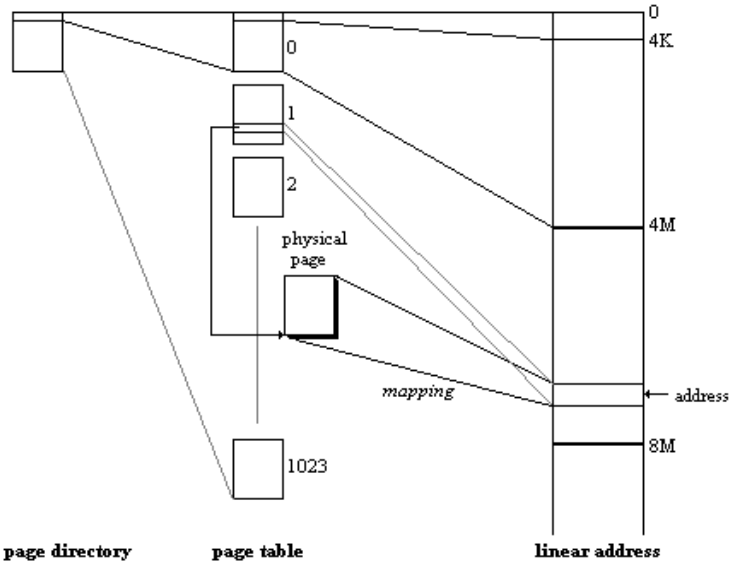
PTRS_PER_PAGE값은 1024로서 page에 들어가는 entry의 수이며, 각 entry의 길이는 4byte이다.

```
#define PTRS_PER_PAGE                    (PAGE_SIZE/sizeof(void*))
```

PAGE_DIR_OFFSET값은 그림<7.4>의 선형주소의 directory에 해당된다. base에는 CR3값이 들어가며, address에 해당하는 page directory entry의 위치를 알수 있다. directory entry 1개는 선형번지 4 MB를 mapping한다.[95]
PAGE_PTR은 그림<7.4>의 선형주소(linear address)의 table에 해당된다. 이 값으로 address에 해당하는 pa -ge table entry의 위치를 알수 있다. table 1개는 선형번지 4 Kbyte를 mapping한다. 이 매크로는 address를 4KB × 4byte로 나눈값의 몫이다. 여기서, 나눈값의 나머지는 선형번지의 offset에 해당한다.
그림<7.5>은 page directory의 첫번째 entry가 선형번지공간 0 ~ 4 Mbyte와, 두번째 entry가 선

95) 이 매크로의 계산법은 간단하다. address를 4Mbyte×4byte(entry길이)로 나눈값의 몫이다.

형번지공간 4 Mbyte ~ 8 Mbyte와 mapping되어 있음을 나타내고 있다.



<그림 7.5> page directory와 선형주소공간의 mapping

이것으로서 우리는 선형주소값이 주어지면, 해당하는 물리 page를 찾아갈수가 있다. 물리 page는 *필요할때마다* page는 free_list에서 가져와 사용하고, 다 사용하면 반납하는 식이므로, 선형번지와 물리번지는 순서대로 mapping되어지지는 않는다. 즉, user process입장에서는 물리번지는 신경을 쓸 필요가 없으며, 물리메모리가 보이지도 않는다. user가 선형번지에 대해 작업을 하면, kernel이 놀고있는 page를 user가 원하는 선형번지에 할당한다.

이 지점에서 우리는 매우 중요한 것을 하나 지적해야 한다. process마다 3 Gbyte의 선형번지공간을 가지고 있다는 것이다. 왜냐하면 fork system call에서 child process를 만들때, 그에게 새로운 page directory를 할당한다. fork는 7.3절에서 논의된다. 즉, process마다 독자적인 자신의 page directory를 가진다는 것이다.[96] 따라서 모든 process는 3 Gbyte의 공간을 쳐다보며, 작업을 하게된다. 만약에 user process가 그 이상을 접근하려고 시도한다면, page fault가 발생하게 되며 SIGKILL signal을 받게된다. page fault는 7.6절에서 논의된다.

---

96) kernel을 위한 page directory는 1장에서 언급되었던 swapper_pg_dir이다.

7.3 fork system call 코드

fork 코드를 살펴보자.

```
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in its entirety.
 */
asmlinkage int sys_fork(struct pt_regs regs)
{
        struct pt_regs * childregs;
        struct task_struct *p;
        int i,nr;
        struct file *f;
        unsigned long clone_flags = COPYVM | SIGCHLD;

        if(!(p = (struct task_struct*)__get_free_page(GFP_KERNEL)))
                goto bad_fork;
1)      nr = find_empty_process();
        if (nr < 0)
                goto bad_fork_free;
        task[nr] = p;
2)      *p = *current;
        p->did_exec = 0;
        p->kernel_stack_page = 0;
3)      p->state = TASK_UNINTERRUPTIBLE;
        p->flags &= ~(PF_PTRACED|PF_TRACESYS);
4)      p->pid = last_pid;
5)      p->swappable = 1;
6)      p->p_pptr = p->p_opptr = current;
        p->p_cptr = NULL;
7)      SET_LINKS(p);
        p->signal = 0;
        p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
```

```
        p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
        p->leader = 0;          /* process leadership doesn't inherit */
        p->utime = p->stime = 0;
        p->cutime = p->cstime = 0;
        p->min_flt = p->maj_flt = 0;
        p->cmin_flt = p->cmaj_flt = 0;
        p->start_time = jiffies;
/*
 * set up new TSS and kernel stack
 */
        if (!(p->kernel_stack_page = __get_free_page(GFP_KERNEL)))
                goto bad_fork_cleanup;
        p->tss.es = KERNEL_DS;
        p->tss.cs = KERNEL_CS;
        p->tss.ss = KERNEL_DS;
        p->tss.ds = KERNEL_DS;
        p->tss.fs = USER_DS;
        p->tss.gs = KERNEL_DS;
        p->tss.ss0 = KERNEL_DS;
        p->tss.esp0 = p->kernel_stack_page + PAGE_SIZE;
8)      p->tss.tr = _TSS(nr);
        childregs = ((struct pt_regs *) (p->kernel_stack_page + PAGE_SIZE)) - 1;
        p->tss.esp = (unsigned long) childregs;
        p->tss.eip = (unsigned long) ret_from_sys_call;
        *childregs = regs;
        childregs->eax = 0;
        p->tss.back_link = 0;
9)      p->tss.eflags = regs.eflags & 0xffffcfff;       /* iopl is always 0 for a new
process */
10)     if (IS_CLONE) {
                if (regs.ebx)
                        childregs->esp = regs.ebx;
                clone_flags = regs.ecx;
                if (childregs->esp == regs.esp)
                        clone_flags |= COPYVM;
        }
        p->exit_signal = clone_flags & CSIGNAL;
11)     p->tss.ldt = _LDT(nr);
12)     if (p->ldt) {
```

```
                p->ldt = (struct desc_struct*) vmalloc(LDT_ENTRIES*LDT_ENTRY_SIZE);
                if (p->ldt != NULL)
                        memcpy(p->ldt, current->ldt, LDT_ENTRIES*LDT_ENTRY_SIZE);
        }
13)     p->tss.bitmap = offsetof(struct tss_struct,io_bitmap);
        for (i = 0; i < IO_BITMAP_SIZE+1 ; i++) /* IO bitmap is actually SIZE+1 */
14)             p->tss.io_bitmap[i] = ~0;
15)     if (last_task_used_math == current)
                __asm__("clts ; fnsave %0 ; frstor %0":"=m" (p->tss.i387));
        p->semun = NULL; p->shm = NULL;
16)     if (copy_vm(p) || shm_fork(current, p))
                goto bad_fork_cleanup;
        if (clone_flags & COPYFD) {
                for (i=0; i<NR_OPEN;i++)
                        if ((f = p->filp[i]) != NULL)
                                p->filp[i] = copy_fd(f);
        } else {
                for (i=0; i<NR_OPEN;i++)
                        if ((f = p->filp[i]) != NULL)
                                f->f_count++;
        }
        if (current->pwd)
                current->pwd->i_count++;
        if (current->root)
                current->root->i_count++;
17)     if (current->executable)
                current->executable->i_count++;
18)     dup_mmap(p);
19)     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
20)     if (p->ldt)
                set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,p->ldt, 512);
        else
                set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&default_ldt, 1);

21)     p->counter = current->counter >> 1;
22)     p->state = TASK_RUNNING;        /* do this last, just in case */
23)     return p->pid;
bad_fork_cleanup:
        task[nr] = NULL;
```

```
        REMOVE_LINKS(p);
        free_page(p->kernel_stack_page);
bad_fork_free:
        free_page((long) p);
bad_fork:
        return -EAGAIN;
}
```

1) task table에서 빈 곳을 찾는다.

2) 일단은 부모 process의 정보를 자식 process에게 그대로 전달한다. 이것은 *exec* system call에 의해 상당부분이 다시 set한다.

3) 자식 process의 state변수를 fork과정이 끝날때까지 취침상태로 둔다.

4) get_empty_process함수에 의해 구한 자식 process의 PID를 set한다.

5) 이것으로써 *init* daemon을 제외한 모든 process는 swap-out시킬수 있다.

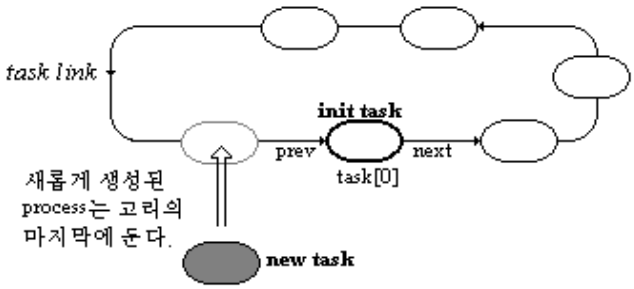
6) p_pptr(parent pointer), p_opptr(*original* parent pointer)은 부모 process의 pointer에 해당한다. 부모 process가 종료되면 child process는 고아(orphan)가 되며, init daemon이 부모 process가 되어, 이들 변수에 들어가게 된다.[97] (do_exit함수<kernel/exit.c>, 6장 프로세스 관리) p_pptr은 *ptrace* system call에 의해 바뀔수 있다. A라는 process를 PTRACE_ATTACH 명령과 함께 ptrace하는 B라는 다른 process가 있다면 B process는 A process의 임시 부모 proce-ss가 된다. 이때 B process는 A process를 자식 process의 대열에 두기위해 아래 7)의*SET_LINK*과정을 수행한다. PTRACE_DETACH명령시에 p_pptr에는 p_opptr이 다시 set된다.

7) SET_LINK
   - 생성된 자식 process를 process들의 고리에 포함시킨다.

   - 자식process들을 **sibling**이라고 한다.
     · **p_cptr(child pointer)**

---

97) Advanced Programming in the Unix Environment에서
9.10 Orphaned Process Groups



<그림 7.6> new task가 끼워지는 위치

가장 최근에 생성시킨 자식 process, 가장 어린 자식 process에 해당한다.

- **p_ysptr(younger sibling pointer)**

  바로 아래 동생 process.

- **p_osptr(older sibling pointer)**

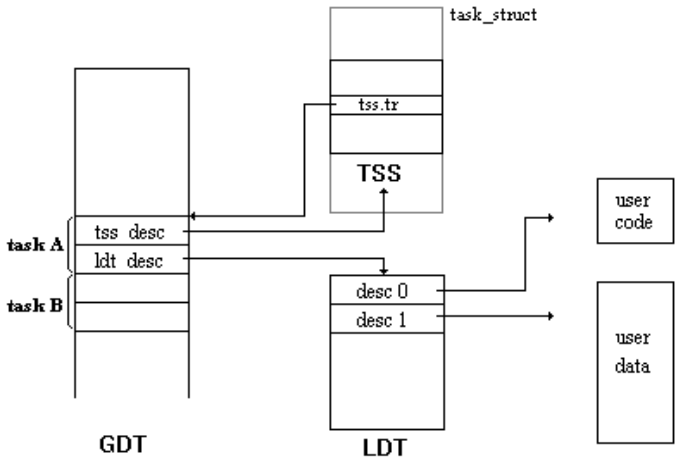
  바로 윗 형 process.

이 pointer들이 자식 process들을 연결시켜 한가족으로 만들어준다.

8) 19) _TSS(nr)은 GDT 에서의 자식 process의 TSS descriptor offset이다. tss.tr은 task 전환시 task register에 들어가야 할 내용이다. 다시 말하면, tss.tr의 내용물이 task register에 load됨으로써 이 task가 CPU를 점유하게 된다.(load_TR정의 <sched.h>)
TSS descriptor를 GDT에 set한다. tss에는 task register외에도 task전환시 control register에 load될 tss.cr3, tss.cr2도 있음을 주목하자. 그러나 tss.tr과 tss.cr2는 processor에 의해 관리되는 것이 아니라, kernel에 의해 관리된다. tss구조체에서 processor에 의해 관리되는 부분은 io_bitmap까지이다.

9) eflag의 IOPL bit에 의해 I/O를 허용하는 특권level을 결정한다. IOPL은 2개의 bit로 이루어지며, 만약 이 값이 1이라면, 특권level(CPL)이 1이상인(즉 1과 0) process들만이 I/O가 허용이 된다. I/O관련 명령어에는 port를 제어하는 in, out 과 hardware interrupt와 관련된 sti, cli 등이 있다.[98] fork시에는 IOPL을 0로 초기화한다.

10) *vfork* system call을 위한 부분이다. vfork에 대해서는 manual page나 C library reference를 참고.



<그림7.7> GDT와 tss내의 tr

11) 12) 20) 부모 process의 ldt를 그대로 복사한다. ldt에 대한 사항은 7.7절에서 논의된다.

13) 14) 어떤 task에 대해 특정 I/O 를 허용하지 않기위해 사용된다. 이것은 io_bitmap[]에 의해 이루어지는데, 해당 bit를 set하면 **허용가능**, no-set이면 **허용불가**가 된다. io_bitmap[]의 각 bit는 port 0x00에서 port 0x3ff까지와 mapping되어있다.

15) 6.4절 task 전환 참고.

16) copy_vm은 fork.c에 정의되어 있다.

```
#define copy_vm(p) ((clone_flags & COPYVM)?copy_page_tables(p):clone_page_tables(p))
```

vfork와 fork를 구분하기 위한 것이다.

fork system call을 위한 copy_page_table은 부모 process의 page directory, page table을 그대로 자식 process의 그것들에게 그대로 복사한다. 물론 이것은 자식 process의 page director-y와 page table을 위한 새로운 page를 획득한후에 이루어진다. 이 때 부모 process의 물리 page(page table entry)는 자식 process와 공유하게 된다. 이 후 어느 한쪽이 page에 쓰기작업을 시도하면 그때서야 비로소 새로운 page가 할당되고, 새로운 page에 기존의 내용을 그대로

98) 80x86 ARCHITECTURE AND PROGRAMMING VOLUME II 2.6.2 I/O access and IOPL

복사한후, 작업을 요구한 process에게 건네진다. 이것은 **Copy-On-Write**라고 부른다. 즉 공유하고 있다가 쓰기작업이 필요할때만 새로운 page복사본이 할당된다는 것이다. copy-on-write를 수행하는 코드는 7.6.2절에서 논의된다.
새롭게 만들어진 자식process의 page directory pointer를 task->tss.cr3에 넣게 된다. task전환이 있으면 processor는 다음 task의 tss구조체에 있는 cr3값으로, 다음 task의 선형주소공간을 인식하게 된다.
vfork를 위한 clone_page_tables는 자식 process를 위한 page directory및 table을 새로 만들지 않고, 부모의 것을 같이 사용한다. 단지 mmap[]을 1증가시켜 같은 page를 공유함을 표시한다.

17) executable은 부모 process의 실행file(binary)[99]에 해당한다. 물론 자식 process도 executable은 같을것이다. 따라서 같은 실행 file을 두 process이상이 사용하고 있으므로 i_count값을 1 증가시킨다. i_count는 해당 inode를 몇개의 process가 사용중인가를 표시한다.

18) 부모 process의 영역(vm_area)를 자식 영역에 그대로 복사한다.
영역에 대해서는 7.4절에서 다룬다.

21) counter는 process의 timeslice를 결정하는데, tick(1/100초)이 있을때마다 1씩 줄어든다.
자세한 사항은 6장 process관리에서 논의된다.

22) 이제 비로소 수행상태로 바꾼다.

23) 부모 process에게 자식 process의 PID를 return.

7.4 영역(vm_area)

7.4.1 mmap system call

mmap system call은 disk의 일부분을 memory와 같은 크기로 mapping시킨다. 그래서 직접 disk를 접근하는 것이 아니라 mapping되어진 memory에 작업을 하는 것은 disk에 대해 작업하는 것과 동일한 결과를 가진다. 즉 mapping된 memory에 쓰기작업을 하면, disk에도 쓰여진다.
그림<7.8>은 이것을 도식화하였다.

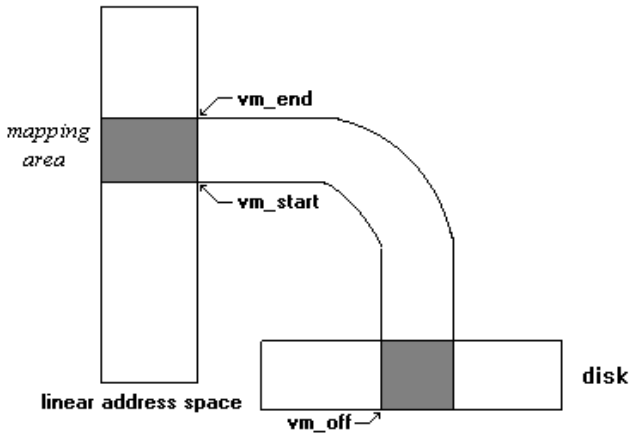
---

99) *bash* process라면 /sbin/bash file.

여기서 mapping된 주소공간은 선형주소공간(physical memory가 아닌)임을 주목하자.

다음은 vm_area구조체(mm.h)를 나타내었다.

```
struct vm_area_struct {
        struct task_struct * vm_task;           /* VM area parameters */
        unsigned long vm_start;
        unsigned long vm_end;


        unsigned short vm_page_prot;
        struct vm_area_struct * vm_next;        /* linked list */
        struct vm_area_struct * vm_share;       /* linked list */
        struct inode * vm_inode;
        unsigned long vm_offset;
        struct vm_operations_struct * vm_ops;
};
```



<그림7.8> memory mapping

영역이란 이렇게 mapping된 memory 부분을 말한다. vm_start는 영역시작을, vm_end는 영역끝을 의미한다. vm_inode는 mapping된 disk상의 file inode이며, vm_offset은 file inode의 offset에 해당한다.

vm_task는 mmap을 호출한 process이며, vm_page_prot는 영역에 대해 쓰기,읽기작업허가를 나타낸다. vm_next는 다음 영역으로서, 한 process에 여러개의 영역이 있을수 있으므로 그들을 연결시킨다. vm_share는 영역을 공유하여 memory를 절약하기 위해 사용된다.

다음은 vm_page_prot에 set되는 것들(mm.h)이다.

```
#define PAGE_PRIVATE    (PAGE_PRESENT | PAGE_RW | PAGE_USER | PAGE_ACCESSED | PAGE_COW)
#define PAGE_SHARED     (PAGE_PRESENT | PAGE_RW | PAGE_USER | PAGE_ACCESSED)
#define PAGE_COPY       (PAGE_PRESENT | PAGE_USER | PAGE_ACCESSED | PAGE_COW)
#define PAGE_READONLY   (PAGE_PRESENT | PAGE_USER | PAGE_ACCESSED)
#define PAGE_TABLE      (PAGE_PRESENT | PAGE_RW | PAGE_USER | PAGE_ACCESSED)
```

이들에 대해서는 mmap의 manual page와 do_mmap(mm/mmap.c)함수 코드를 참고하라.

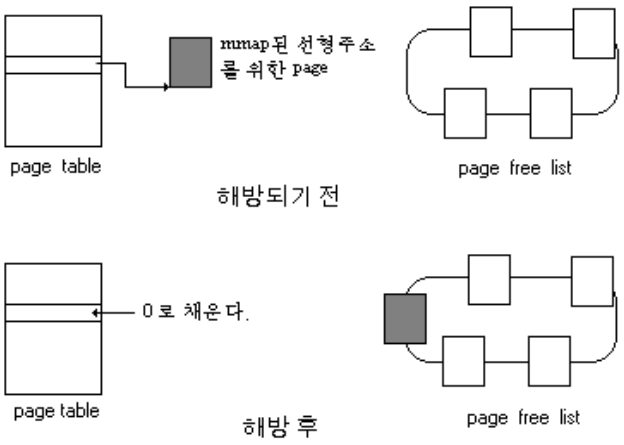
7.4.2 do_mmap 함수코드중에서

file에 대해 mmap을 실시할 경우, generic_mmap함수가 수행된다. 일반적으로 user program은 어떠한 한 file에 대해 mmapping을 시킨다. 그러나 file pointer값을 전달받지 못한 경우에는 ano-n_map함수가 수행되는데, 이러한 경우에 대해서는 7.5절 exec system call에서 다룬다.
generic_mmap함수와 anon_map함수의 수행과정을 비교해보자.

	generic_mmap	anon_map
영역해방 함수	**unmap_page_range(addr, len)**	**zeromap_page_range(addr, len, mask)**
	bread	mpnt->vm_task = current
	mpnt->vm_task = current	mpnt->vm_start = addr
	mpnt->vm_start = addr	mpnt->vm_end = addr + len
	mpnt->vm_end = addr + len	mpnt->vm_page_prot = mask
	mpnt->vm_page_prot = prot	mpnt->vm_share = NULL
	mpnt->vm_share = NULL	mpnt->vm_inode = NULL
file inode	mpnt->vm_inode = inode	
	inode->i_count++	mpnt->vm_offset = 0
file offset	mpnt->vm_offset = off	mpnt->vm_ops = NULL
영역 함수	mpnt->vm_ops = &file_mmap	insert_vm_struct
	insert_vm_struct	merge_segments
	merge_segments	

<표7.1>

· 표<7.1>에서 양쪽의 영역해방함수는 유사한 기능을 한다. 두 함수는 process의 page table에서 선형주소를 위해 할당된 page에 대해 선형번지 addr부터 len크기만큼 해방시킨다.(do_mmap함수에 의해)



<그림7.68> 영역해방

그림<7.9>에는 page를 해방시키는 과정이 나타나 있다. page table과 page의 크기는 같으나 편의상 page를 작게 그렸다.

· bread함수는 mapping된 첫 block을 읽어 해당 file에 대해 읽기작업이 가능한지를 확인한다.
bmap(inode, offset)함수는 file의 offset을 논리 block번호로 바꾸어 return한다.

· anon_map함수에서는 vm_inode와 vm_offset, vm_ops이 NULL로 채워진다.
영역함수들은 file_mmap 구조체에 의해 명시된다.

```
struct vm_operations_struct file_mmap = {
        NULL,                   /* open */
        file_mmap_free,         /* close */
        file_mmap_nopage,       /* nopage */
```

```
        NULL,                   /* wppage */
        file_mmap_share,        /* share */
        NULL,                   /* unmap */
        };
```

- insert_vm_struct함수는 영역의 고리에 이 영역을 끼워넣는다. 영역의 고리는 vm_start가 낮은 영역이 앞에 놓이도록 한다.
- merge_segment함수는 조건만 맞으면 두 영역을 합쳐서 분리된 영역의 수를 줄이는 역활을 한다.
- exeutable은 binary inode로서 binary가 수행중에 있음을 나타낸다.
- return값은 mmap이 시작된 memory상의 주소.

◆ kernel 1.0에서는 일반적인 file에 대해 mmap을 할 경우 page에 대한 priority로서 PAGE_RW는 허용되지 않고 있다.

```
int do_mmap(struct file * file, unsigned long addr, unsigned long len,
        unsigned long prot, unsigned long flags, unsigned long off)
{
        ................................................

        mask = 0;
        if (prot & (PROT_READ | PROT_EXEC))
                mask |= PAGE_READONLY;
        if (prot & PROT_WRITE)
                if ((flags & MAP_TYPE) == MAP_PRIVATE)
                        mask |= PAGE_COPY;
                else
                        mask |= PAGE_SHARED;  /* PAGE_RW가 포함되어 있다. */
        if (!mask)
                return -EINVAL;
```

PAGE_SHARED에는 PAGE_RW bit가 포함되어 있다. 그러나 generic_mmap함수에 의해 PAGE_RW bit가 허용되지 않음을 알수 있다. 그래서 일반 file을 mmap을 하면서 쓰기작업을 원한다면 MAP_PRIVATE flag를 set해 주어야 한다. 결국 **이 부분에서는**[100] PAGE_SHARED도 또한 지원되지 않는다. OMAGIC binary를 mmap하는 경우에는 이러한 제한을 두고있지 않다.(anon_map함수

100) paging_init에서 page table들을 PAGE_SHARED로 초기화한것을 기억하자.

참고)

```
int generic_mmap(struct inode * inode, struct file * file,
        unsigned long addr, size_t len, int prot, unsigned long off)
{
        if (prot & PAGE_RW)     /* only PAGE_COW or read-only supported right now */
              return -EINVAL;

}
```

7.5 exec system call

exec system call은 새로운 program을 실행시키기 위해 사용된다. 다음 구조체(include/linux/binfmts.h)는 file을 실행시킬때 사용되는 정보를 포함하고 있다.

```
linux_binprm{
  char buf[128];
  unsigned long page[MAX_ARG_PAGES];
  unsigned long p;
  int sh_bang;
  struct inode * inode;     /* inode of binary  */
  int e_uid, e_gid;
  int argc, envc;
  char * filename;          /* Name of binary */
};
```

- buf는 실행시킨 file이 shell script인지를 확인하기 위해 위해 실행 file의 머릿부분을 읽어들이는 곳이다.
- page[]는 program의 argument및 환경변수를 담은 page의 pointer를 가지고 있다. 그래서 환경변수가 가질수 있는 최대 크기는 32(MAX_ARG_PAGES) × PAGE_SIZE가 된다. 이 정도 큰 규모의 환경변수는 shell script에 의해 만들어질수 있다. 예를 들어 bash shell script를 실행시킬려고 한다고 가정하자. 이 때 exec는 script를 읽어서 #! 이후에 shell(**sh**나 **bash**등)이 명시되어 있는지 확인하고, 그렇다면 shell script이름 및 script에 전달된 인수들을 bash에 전달되는 인수들의 양식(format)으로 만들어 32개의 page에 담는다. 그리고 그 page들의 pointer는 page[]배열에 들어가게 된다. 이후 page pointer를 전달받은 bash가 수행되면서 shell sc

-ript의 내용을 parsing한다.

▶ bash는 "c"나 "s" option이 주어지지 않으면, 첫번째 인수를 shell script인수로 받아들인다.

- p는 인수가 들어있는 page의 인수시작pointer이다.
- sh_bang는 shell script를 수행시켰을 경우에 1이 set되며, binary를 수행시켰을 경우는 0이 된다.
- inode는 binary의 inode이며, shell script를 수행시켰을 경우는 shell의 inode이다.
- 전달된 인수와 환경변수의 갯수.
- filename은 실행시킨 file의 이름이다.

7.5.1 do_exec함수 코드중에서

1) effective ID와 Set user ID

process에게는 real ID와 effective ID가 있다. real ID는 file을 생성시킨 user에 의해 set된다. 즉 *happy*라는 user가 *foo*라는 실행file을 생성시키면 real ID는 happy에 종속된다. effective ID도 마찬가지다. 하지만 effective ID는 상황에 따라 바뀌게 된다.

```
# ls -l foo
-rwxr-xr-x  1 happy   126 Jul 30 15:17 foo
```

여기서 happy가 foo를 실행시키면 foo의 real ID와 effective ID는 happy의 user ID이다.
그런데 만약에 다른 user process인 *hello*가 exec에 의해 foo를 실행시키면 어떻게 될까? foo의 real ID와 effective ID는 hello의 user ID가 된다.
이것은 Super user(root) program에도 마찬가지로 적용된다. Super user program의 real ID와 effective ID는 *0* 이지만, happy가 Super user의 program인 e2fsck를 실행시킨다면 e2fsck의 real ID와 effective ID는 happy의 user ID가 될것이다.

나아닌 다른 user가 내 program을 수행시킬수는 있지만 그 때도 effective ID는 바뀌지 않기를 바라는 경우가 있다. 예를 들어 root에 소속된 goodguy라는 게임실행file이 있다고 가정하자.

```
# ls -l goodguy
-rwxr-xr-x  1 root   235162 Aug 2 13:50 goodguy
```

만약 happy가 이 게임을 실행시키면, goodguy process의 real ID와 effective ID는 0가 아닌 happy의 user ID가 된다. 그런데, 게임이 *score* file을 관리하고 있다면, 이 file은 goodguy 실행 file에 의해서만 갱신될수 있어야 한다. 즉 effective ID가 0인 process만이 갱신이 가능하도록 set 되어 있어야 한다.

```
# ls -l score
-rwxr--r--  1 root      326 April 7 10:23 score
```

이렇게 set된 *score* file을 goodguy process가 갱신할수는 없다. 왜냐하면 게임이 실행될때 effect -ive ID가 happy ID로 바뀌었기 때문이다. 그래서 goodguy process의 effective ID가 바뀌지 않도록 하기위해 user ID bit를 set한다. 이 bit를 **set-user-ID** flag라고 부른다.

```
# chmod u+s goodguy
# ls -l goodguy
-rwsr-xr-x  1 root   235162 Aug 2 13:50 goodguy
```

s 로 바뀐 부분을 보자.
이제 user가 goodguy를 실행시켜도 goodguy process의 effective ID는 0 이다. 그러나 real ID는 여전히 happy ID로 바뀐다. 다음 코드의 7,8번째 줄을 보자.

```
if (current->flags & PF_PTRACED) {
 /* 현재process가 ptrace되는 경우는 euid를 그대로 받는다. 그런데, 이 euid는
 사실 현재process를 ptrace하고 있는 process의 euid이다.  */
        bprm.e_uid = current->euid;
        bprm.e_gid = current->egid;
} else {
        bprm.e_uid = (i & S_ISUID) ? bprm.inode->i_uid : current->euid;
        bprm.e_gid = (i & S_ISGID) ? bprm.inode->i_gid : current->egid;
}
```

다음 코드에서는 Super user이더라도 mode에서 *execution* bit가 단 한개도 set되어 있지 않으면 binary를 실행시킬수 없음을 나타내고 있다.

```
if (!(i & 1) &&
    !((bprm.inode->i_mode & 0111) && suser())) {
        retval = -EACCES;
```

```
                goto exec_error2;
        }
```

2) user segment에서 kernel segment로의 전환.

```
   old_fs = get_fs();   /* user segment */
   set_fs(get_ds());    /* kernel segment */
   retval = read_exec(bprm.inode,0.bprm.buf,128);
   set_fs(old_fs);      /* restore */
```

bprm.buf가 kernel 선형번지에 있으므로 fs register에 kernel segment selector를 load한다. (만약에 bprm.buf가 user선형번지에 있다면 fs register에는 user segment selctor가 들어가야 한다.) system call이 호출되면 곧 fs에는 user segment selector가 들어간다.

3) verify_area

이 함수는 VERIFY_WRITE명령(argument *type*)으로 addr부터 size만큼의 선형주소공간에 쓰기작업이 가능한지 확인한다. 만약에 address에 해당하는 물리page를 다른 process와 공유하고 있다는 이유등으로 해서 쓰기작업이 가능하지 않다면, address에 새로운 물리 page를 할당함으로써 쓰기작업이 가능하도록 한다. 이것은 do_wp_page함수에 의해 이루어진다. do_wp_page함수에 대해서는 7.6절에서 다룬다.
아래 코드에서 보다시피 VERIFY_READ명령에 의해서는 단지 TASK_SIZE(3 Gbyte)범위를 벗어나지 않는지만을 확인하다.

```
extern inline int verify_area(int type, const void * addr, unsigned long size)
{
        if (TASK_SIZE <= (unsigned long) addr)
                return -EFAULT;
        if (size > TASK_SIZE - (unsigned long) addr)
                return -EFAULT;
        if (wp_works_ok || type == VERIFY_READ || !size)
                return 0;
        return __verify_write((unsigned long) addr,size); /* mm/memory.c  */
}
```

4) read_exec

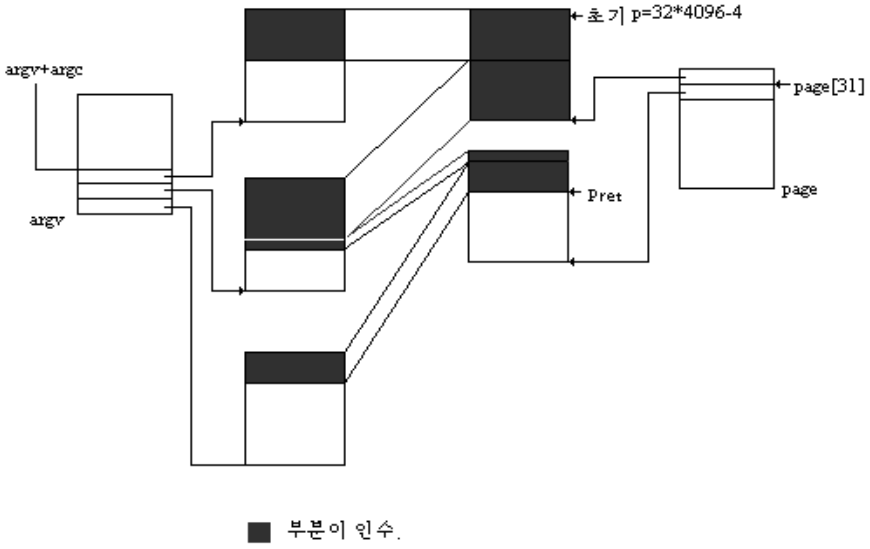
inode인 file의 offset에서 addr로 count만큼 복사한다. 단, 복사하기 전에 addr이 user공간내에 있는지, kernel공간내에 있는지를 확인한다.

```
if( getfs()==USER_DS )
```

addr이 user공간내에 있다면(load_aout_binary함수내의 read_exec), verify_area함수를 통해 쓰기가 가능하도록 확인작업을 한다. kernel공간에 있다면(do_execve함수내의 read_exec), page fault가 발생하는 경우에 쓰기가능확인이 이루어진다.

5) copy_string(int argc,char ** argv,unsigned long *page,unsigned logn p,int from_kmem)

argv의 내용을 page로 argc만큼 복사한다. p는 복사할수 있는 최대크기이다. 초기에 이 값은 MAX_ARG_PAGES * PAGE_SIZE-4 이다. return값은 page에서 인수시작 offset pointer이다. 이 값은 복사할수 있는 남은공간을 의미하기도 한다. 그림<7.10>는 argc가 3인 경우를 예로 들었다. Pret는 return되어지는 p값으로,(32*4096-복사된 인수의 양)이다.



<그림7.10> copy_string 수행

from_kmem은 argv가 *user공간*으로부터 page로 복사하는지, *kernel공간*으로부터 복사하는지를 결정한다.(0이면 user공간으로 부터, 2이면 kernel공간으로부터)

6) fmt->load_binary

binary형태는 A.OUT, ELF, COFF등 여러가지가 있는데, 그에 따라 load_aout_binary, load_elf_binary, load_coff_binary등의 함수들에 의해 binary가 memory에 적재된다.
각 format에 대해서는 관련자료를 찾아보기 바란다.

*fs/exec.c*의

```
struct linux_binfmt formats[]={
       {load_aout_binary, load_aout_libray},
#ifdef CONFIG_BINFMT_ELF
       {load_elf_binary, load_elf_libray},
#ifdef CONFIG_BINFMT_COFF
       {load_coff_binary, load_coff_libray},
#endif
       {NULL,NULL}
};
```

지금까지는 A.OUT fomat이 많이 사용되어 왔다. 근래에 와서는 shared library를 구성하기 용이하다는 이유로 ELF format으로 흘러가는 경향이 있다.
본서에서는 A.OUT format에 대해 다루기로 한다. 물론 load_aout_binary에 의해 실행화일(binary)이 load된다. 만약 file을 실행시켰을때 load_aout_binary가 load에 실패한다면 ,load_elf_binary, load_coff _binary들이 차례로 시도될것이다.

7.5.2 load_aout_binary

A.OUT fomat에는 다시 다음과 같은 몇가지의 실행file로 분류된다.(a.out.h)
편의상 8진수로 정의된 것에 대해 16진수의 주석을 달아두었다. 이들은 다음에 나올 exec구조체의 **a_info**에 들어가게 된다.

```
/* Code indicating object file or impure executable.  */
#define OMAGIC 0407    /* 0x107 */
/* Code indicating pure executable.  */
#define NMAGIC 0410    /* 0x108 */
/* Code indicating demand-paged executable.  */
#define ZMAGIC 0413    /* 0x10B */
/* This indicates a demand-paged executable with the header in the text.
   The first page is unmapped to help trap NULL pointer references */
#define QMAGIC 0314    /* 0x0CC */
```

표<7.2>는 여러가지 binary format의 예를 들었다.

program	format	program	format
bash	ZMAGIC	e2fsck	OMAGIC
init	ZMAGIC	ps	OMAGIC
getty	ZMAGIC	e2dump	QMAGIC
login	OMAGIC		

<표7.2> binary format

그러나, program의 binary format이 항상 표<7.2>와 동일한것은 아니다.
gcc에 -qmagic을 옵션으로 줌으로써 QMAGIC format으로 binary를 만들수 있다. 그리고 마찬가지로 gcc에 -N과 -n을 옵션으로 줌으로써 각각 OMAGIC과 NMAGIC의 binary가 만들어진다. 아무런 옵션도 주지 않으면 ZMAGIC의 binary가 생성된다. -N과 -n은 link 옵션으로서, 그 내용은 다음과 같다.

```
-N      specifies  readable and writable text and data sec-
        tions. If the output  format  supports  Unix  style
        magic numbers, the output is marked as OMAGIC.
        When  you  use the `-N' option, the linker does not
        page-align the data segment.


-n      sets the text segment to be read only,  and  NMAGIC
        is written if possible.
```

다음 구조체는 binary에 대한 정보로서 binary의 선두에 들어간다.

```
struct exec
{
  unsigned long a_info;         /* Use macros N_MAGIC, etc for access */
```

```
  unsigned a_text;              /* length of text, in bytes */
  unsigned a_data;              /* length of data, in bytes */
  unsigned a_bss;               /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;              /* length of symbol table data in file, in bytes */
  unsigned a_entry;             /* start address */
  unsigned a_trsize;            /* length of relocation info for text, in bytes */
  unsigned a_drsize;            /* length of relocation info for data, in bytes */
};
```

이 구조체의 크기는 32 byte이므로 33 byte부터 실제 코드가 시작된다.

load_aout_binary의 뒷 부분의 코드를 보자.

```
경우1)  if (N_MAGIC(ex) == OMAGIC) {
                do_mmap(NULL, 0, ex.a_text+ex.a_data,
                        PROT_READ|PROT_WRITE|PROT_EXEC,
                        MAP_FIXED|MAP_PRIVATE, 0);
                read_exec(bprm->inode, 32, (char *) 0, ex.a_text+ex.a_data);
        } else {
                if (ex.a_text & 0xfff || ex.a_data & 0xfff)
                        printk("%s: executable not page aligned\n", current->comm);

                fd = open_inode(bprm->inode, O_RDONLY);

                if (fd < 0)
                        return fd;
                file = current->filp[fd];
                if (!file->f_op || !file->f_op->mmap) {
                        sys_close(fd);
경우2)                  do_mmap(NULL, 0, ex.a_text+ex.a_data,
                                PROT_READ|PROT_WRITE|PROT_EXEC,
                                MAP_FIXED|MAP_PRIVATE, 0);
                        read_exec(bprm->inode, N_TXTOFF(ex),
                                  (char *) N_TXTADDR(ex), ex.a_text+ex.a_data);
                        goto beyond_if;
                }
경우3)          error = do_mmap(file, N_TXTADDR(ex), ex.a_text,
                                PROT_READ | PROT_EXEC,
                                MAP_FIXED | MAP_SHARED, N_TXTOFF(ex));
```

```
        if (error != N_TXTADDR(ex)) {
                sys_close(fd);
                send_sig(SIGSEGV, current, 0);
                return 0;
        };

        error = do_mmap(file, N_TXTADDR(ex) + ex.a_text, ex.a_data,
                        PROT_READ | PROT_WRITE | PROT_EXEC,
                        MAP_FIXED | MAP_PRIVATE, N_TXTOFF(ex) + ex.a_text);
        sys_close(fd);
        if (error != N_TXTADDR(ex) + ex.a_text) {
                send_sig(SIGSEGV, current, 0);
                return 0;
        };
        current->executable = bprm->inode;
        bprm->inode->i_count++;
}
```

위 코드에서는 binary를 실행시키는데 있어 3가지 경우를 제공한다.
표<7.3>는 format에 따른 disk file내의 binary offset(N_TXTOFF)과 binary를 적재할 선형주소(N_TXTADDR)를 나타내고 있다. <a.out.h>

format	N_TXTOFF	N_TXTADDR
OMAGIC	32	0
NMAGIC	32	0
ZMAGIC	1024	0
QMAGIC	0	4096

<표7.3>

경우1) OMAGIC의 경우

처음에 실행file을 모두 memory로 읽어들이기 때문에 mmap을 할때 binary inode을 사용하지 않는다. 더불어 영역함수들도 없다.<표7.1> page fault가 발생하였을때, map된 disk에서 읽어들이기 위해서는 binary inode가 필요하다. 그래서 OMAGIC format에서는 page fault시

단지 새로운 page를 예외발생 address에 할당한다.(get_empty_page)
disk file의 33번째 byte부터 user 선형번지 0으로 binary크기만큼 읽어들인다.

· MAP_FIXED
이 flag가 제공되지 않으면, 그리고 인수로 제공된 address가 적당하지 않으면, 다른 addr을 *mmap* system call이 스스로 찾아 제공한다. 이때 mmap의 return값은 system call이 제공하는 address가 된다. 그러나 이 flag가 set되어 있으면 무조건 인수로 제공된 address에 binary를 적재해야 하고, 이 address pointer가 return한다.

이하는 OMAGIC format이 아닌경우이다.

· open_inode함수
inode인 file의 file descriptor를 구한다.

경우2) mmap함수가 제공되지 않을 경우[101]
default로 anon_map이 수행되고 실행file을 <표7.3>에 따라 load한다.
QMAGIC format의 경우는 memory의 첫 page를 비워놓는데 이것으로서 process가 NULL pointer를 접근하려고 할때를 알아차릴수 있다. 즉, 4096 byte번지 아래쪽을 접근하려고 하면 page fault가 발생하고, page fault handler(do_page_fault함수)에서 다음과 같은 메세지가 출력되고 process는 정지한다.(이 메세지는 /proc/kmsg 에 출력될것이다.)

```
Unable to handle kernel NULL pointer dereference.
```

경우3) OMAGIC format을 제외한 일반적인 경우
text영역과 data영역을 나누어 mmap만을 한다. 이 후 binary가 수행되려고 하면 page fault가 발생하고 fault handler가 요구되는 한 page만을 메모리에 적재하는 식으로 진행된다.(이후 다른 부분이 필요하면 다시 fault handler발생, 다시 한 page 적재) 이렇게 당장 필요할때 물리 page를 사용하는 것을 **요구시 페이징**(*demand paging*)이라 한다.

· flush_old_exec함수
fork할때, 부모 process로부터 복사된 것들중에 영역(vm_area), ldt, signal, page table등을 clear시킨다.
이 함수내에서 clear_page_tables함수를 주목하자. exec system call이 수행되면 process는 완전히 새로운 process의 내용으로 대치된다. exec함수가 정상적으로 수행이 끝나면 자식 process는 종료된다. 비정상적인 종료시에만 exec함수의 다음 코드들이 수행될수 있는 것이

101) 본인은 아직 이러한 경우를 발견하지 못했다.

다. 이 함수는 부모 process의 흔적이라고 할수 있는 page directory entry들을 clear시키고 해당page table을 해방시킨다. 그러나 process가 종료될때 호출되는 free_page_tables함수가 page directory조차 해방시키는 것과는 달리 clear_page_tables는 page directory의 user page 부분만 clear시켜 새로운 process 코드를 위해 page directory는 그대로 사용된다.

· current->executable
binary inode로서 binary가 수행중임을 표시한다. 그러나 OMAGIC의 경우는 NULL이 된다.

· *brk* system call
data영역을 증가시킨다.

change_ldt, create_table함수들에 의해 stack pointer가 set된다.

7.5.3 stack set

환경변수를 process 선형번지공간에 올리고 stack pointer를 set한다.

1) change_ldt함수

7.5절의 초기에서 bprm.page[]에 인수 및 환경변수를 set한 page의 pointer가 적재되어 있다고 한 것을 기억하자. 이 함수는 그 page들을 process의 선형주소공간에 할당한다.[102] 적재되는 장소는 3 GB (TASK_SIZE) 바로 밑이며, 그림<7.11>의 p는 아래 코드의 ②번에 해당한다.

```
①       p += change_ldt(ex.a_text,bprm->page);
②       p -= MAX_ARG_PAGES*PAGE_SIZE;
```

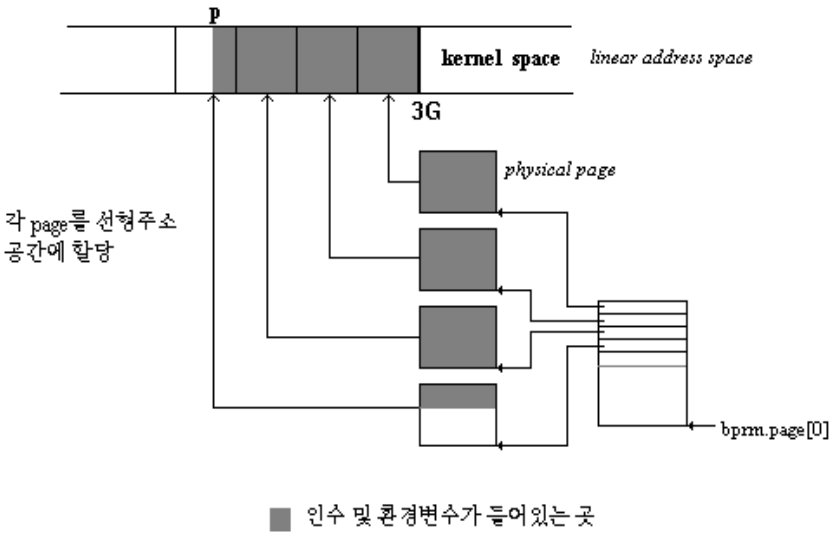
2) create_tables함수

change_ldt함수에 의해 선형공간으로 옮겨진 인수, 환경변수에 대한 pointer table을 생성한다. 그림<7.12>를 참고하라. return값인 p는 그림의 sp에 해당하므로써 stack의 시작번지이기도 하다. binary는 이 stack pointer를 이용하여, execve system call에 의해 전달된 인수 및 환경변수를 얻을수 있다.

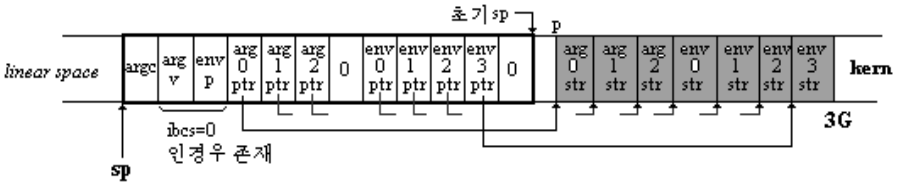
---

102) 왜 이 함수의 이름이 change_ldt인지 본인은 모른다.

```
p = (unsigned long) create_tables((char *)p,bprm->argc,bprm->envc,0);
current->start_stack = p;
regs->eip = ex.a_entry;         /* eip, magic happens :-) */
regs->esp = p;                  /* stack pointer */
```



<그림7.11>



<그림7.12>

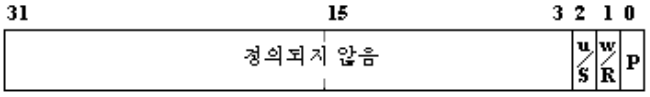
7.6. page fault

task가 access하려는 선형 address에 물리 page가 할당되어 있지 않거나, access할려는 task가 특권 level이 낮아서 access 할수없는 경우에 예외 14(exception 14)인 page fault가 발생한다.
이 때 리눅스에서는 page fault handler인 do_page_fault(mm/memory.c)함수가 수행되는데 그 코드는 다음과 같다.

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
        unsigned long address;
        unsigned long user_esp = 0;
        unsigned int bit;

1)      __asm__("movl %%cr2,%0":"=r" (address));
        if (address < TASK_SIZE) {
2)              if (error_code & 4) {   /* user mode access? */
                        if (regs->eflags & VM_MASK) {
                                bit = (address - 0xA0000) >> PAGE_SHIFT;
                                if (bit < 32)
                                        current->screen_bitmap |= 1 << bit;
                        } else
                                user_esp = regs->esp;
                }
3)              if (error_code & 1)
                        do_wp_page(error_code, address, current, user_esp);
                else
                        do_no_page(error_code, address, current, user_esp);
                return;
        }
        address -= TASK_SIZE;
4)      if (wp_works_ok < 0 && address == 0 && (error_code & PAGE_PRESENT)) {
                wp_works_ok = 1;
                pg0[0] = PAGE_SHARED;
                printk("This processor honours the WP bit even when in supervisor mode.
Good.\n");
                return;
        }
5)      if (address < PAGE_SIZE) {
                printk("Unable to handle kernel NULL pointer dereference");
                pg0[0] = PAGE_SHARED;
        } else
```

```
                printk("Unable to handle kernel paging request");
        printk(" at address %08lx\n",address);
        die_if_kernel("Oops", regs, error_code);
6)      do_exit(SIGKILL);
}
```

1) control register CR2에는 page fault가 발생한 선형 address가 들어있다.
2) 3) error code는 page fault 예외의 경우, 일반적인 예외발생의 경우와 다른 형태의 구조를 갖추고 있다. 그림<7.13>에 그것을 나타내었다.



<그림7.13> page fault예외의 error code구조

- P bit 0 page 존재하지 않음
  1 page level 보호의 위배
- W/R bit 0 fault의 원인이 읽기(read)
  1 fault의 원인이 쓰기(write)
- U/S bit 0 process가 supervisor방식으로 실행되었을때 발생
  1 process가 user방식으로 실행되었을때 발생

page fault의 원인이 page가 존재하지 않기 때문이라면 do_no_page함수가, page level보호가 원인이라면 do_wp_page함수가 수행된다.

4) memory 초기화(1.7.14절 mem_init)시에 WP bit가 지원되는지 확인
5) null pointer접근 확인. (7.5.2절 참고)
6) fault발생한 address가 TASK_SIZE 범위(0 ~ 3GB)안에 있지 않을때 process는 종료된다.

7.6.1 do_no_page 함수

do_no_page함수는 access할려는 address에 page가 할당되어 있지 않을때 page fault예외에 의해 수행된다.

```
void do_no_page(unsigned long error_code, unsigned long address,
        struct task_struct *tsk, unsigned long user_esp)
{
        unsigned long tmp;
        unsigned long page;
        struct vm_area_struct * mpnt;

1)      page = get_empty_pgtable(tsk,address);
        if (!page)
                return;
        page &= PAGE_MASK;
        page += PAGE_PTR(address);
        tmp = *(unsigned long *) page;
2)      if (tmp & PAGE_PRESENT)
                return;
3)      ++tsk->rss;
4)      if (tmp) {
                ++tsk->maj_flt;
                swap_in((unsigned long *) page);
                return;
        }
        address &= 0xfffff000;
        tmp = 0;
        for (mpnt = tsk->mmap; mpnt != NULL; mpnt = mpnt->vm_next) {
                if (address < mpnt->vm_start)
                        break;
                if (address >= mpnt->vm_end) {
                        tmp = mpnt->vm_end;
                        continue;
                }
5)              if (!mpnt->vm_ops || !mpnt->vm_ops->nopage) {
                        ++tsk->min_flt;
                        get_empty_page(tsk,address);
                        return;
                }
6)              mpnt->vm_ops->nopage(error_code, mpnt, address);
                return;
        }
7)      if (tsk != current)
```

```
                goto ok_no_page;
        if (address >= tsk->end_data && address < tsk->brk)
                goto ok_no_page;
        if (mpnt && mpnt == tsk->stk_vma &&
            address - tmp > mpnt->vm_start - address &&
            tsk->rlim[RLIMIT_STACK].rlim_cur > mpnt->vm_end - address) {
                mpnt->vm_start = address;
                goto ok_no_page;
        }
        tsk->tss.cr2 = address;
        current->tss.error_code = error_code;
        current->tss.trap_no = 14;
        send_sig(SIGSEGV,tsk,1);
        if (error_code & 4)     /* user level access? */
                return;
ok_no_page:
        ++tsk->min_flt;
        get_empty_page(tsk,address);
}
```

1) address에 해당하는 page table이 있는지 확인. 없다면 물리page를 address에 해당하는 page table에 할당한다.
2) address에 해당하는 page가 memory상에 존재하는지 확인.
3) rss는 task가 사용중인 물리 page의 수이다. 이 곳에서 address에 한 page를 할당할 것이므로 rss값을 1 증가시킨다.
4) page가 현재 memory상에는 없지만, page table이 page pointer값을 가지고 있다면 해당 page가 disk에 *swap-out*된 상태임을 의미한다. 따라서 swap-in을 실시하여 memory에 해당 page를 load시킨다.

   page fault handler는 page fault가 일어난 횟수를 기록하는데, disk로부터 읽어들여야 하는 경우는 **maj_flt**, 그렇지 않은 경우는 **min_flt**에 기록된다. maj_flt는 지금처럼 swap_in이 수행될때 기록된다.
5) OMAGIC binary format의 경우에 해당된다. 단지 address에 물리 page를 할당한다.
6) file_mmap_nopage함수가 수행된다.

   이 함수는 다른 물리page와 공유할수 있는지 확인하고, 불가능하다면 address에 새로운 page를 할당한다.
7) 다음 세가지 경우를 제외하고는 SIGSEGV signal을 process에게 보낸다. 이 signal을 받은 pr-ocess의 default동작은 process를 종료시키는 것이다.

- 다른 process에 의해 ptrace되어지는 경우.(ptrace.c의 getlong, putlong함수 참고)
- 예외가 발생한 지점이 bss영역내인 경우.
- 예외가 발생한 지점을 영역(vm_area)안에 포함시킬수 있는 경우.

위의 세가지 경우가 성립되면 새로운 page를 address에 할당한다.

7.6.2 do_wp_page 함수

do_wp_page함수는 process가 access(읽기, 쓰기작업)할려는 번지에 access가 보장되지 않는 경우, page fault예외에 의해 수행된다. page fault예외의 경우는 해당 page의 R/W bit와 특권레벨의 확인에 의해 이 함수가 호출되지만, 다른 process를 ptrace하는 경우에도 그 process에 있어서의 page공유에 의해 software상에서 이 함수가 호출되기도 한다.
이때 kernel은 새로운 page를 만들어 address에 할당하고, 기존의 공유했던 page의 내용을 그대로 새로운 page에 복사한다. 이것이 7.3절에서 설명한 Copy-On-Write(COW)방식이다. 즉 쓰기작업을 수행할때 비로소 똑같은 새로운 page를 할당하는 것이다.

```
void do_wp_page(unsigned long error_code, unsigned long address,
{
                    /*   중 략  */

1)              if (!(page & PAGE_COW)) {
                        if (user_esp && tsk == current) {
                                current->tss.cr2 = address;
                                current->tss.error_code = error_code;
                                current->tss.trap_no = 14;
                                send_sig(SIGSEGV, tsk, 1);
                                return;
                        }
                }
2)              if (mem_map[MAP_NR(page)] == 1) {
                        *pg_table |= PAGE_RW | PAGE_DIRTY;
                        invalidate();
                        return;
                }
                        /*  중  략  */
}
```

1) PAGE_RW bit와 PAGE_COW가 모두 set되어있지 않으면 기본적으로 read only상태임을 의미한다. 이들이 set되어있지 않고, 수행중인 process자신의 내부에서 쓰기작업에 대한 page fault가 발생했다면 SIGSEGV signal을 process에게 보낸다. 만약 다른 process를 **ptrace**하는 중에 발생했다면 이것은 ptrace를 당하고 있는 process내에서 page 공유의 이유로 일어난 것이다. 이때 do_wp_page함수는 쓰기작업이 가능한지 다시 확인한다.(아래 2) 참고) 다음은 *put_long*함수의 내용이다.(kernel/ptrace.c)

```
if (!(page & PAGE_RW)) {
        if(!(page & PAGE_COW))
                readonly = 1;
        do_wp_page(PAGE_RW | PAGE_PRESENT,addr,tsk,0);
        goto repeat;
}
```

◆ PAGE_COW bit : 이 bit는 Intel사에 의해 system programmer가 사용할수 있도록 확보된 bit들 중에서 리눅스가 만든 page상태 bit로서, 쓰기작업 error가 발생했을 때 copy-on-write를 시킬수 있는 page인지를 결정한다. 이 bit는 **PAGE_PRIVATE**와 **PAGE_COPY**에 포함된다. 같이 포함된 bit중 **PAGE_USER** bit가 있다. 이것은 user mode에서 access할수 있도록 하기 위해서이다.
PAGE_COPY는 mmap수행시 전달되는 **MAP_PRIVATE** flag에 의해 영역구조체의 vm_page_prot에 set된다.

2) 혼자서 page를 사용하고 있다는 것이 확인되면, 쓰기작업이 가능하도록 한다. 공유하고 있는 상태이면 __do_wp_page함수에 의해 copy-on-write가 수행된다.

```
static void __do_wp_page(unsigned long error_code, unsigned long address,
        struct task_struct * tsk, unsigned long user_esp)
{
                        /* 중  략 */

1)      if (mem_map[MAP_NR(old_page)] != 1) {
                if (new_page) {
                        if (mem_map[MAP_NR(old_page)] & MAP_PAGE_RESERVED)
                                ++tsk->rss;
                        copy_page(old_page,new_page);
                        *(unsigned long *) pte = new_page | prot;
```

```
                        free_page(old_page);
                        invalidate();
                        return;
                }
2)              free_page(old_page);
                oom(tsk);  /*  out of memory  */
                *(unsigned long *) pte = BAD_PAGE | prot;
                invalidate();
                return;
        }
3)      *(unsigned long *) pte |= PAGE_RW;
        invalidate();
        if (new_page)
                free_page(new_page);
        return;
                 /*  중  략  */

}
```

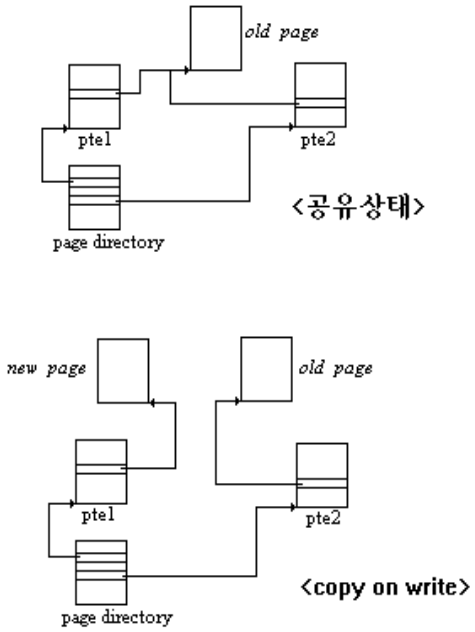
1) page를 공유하고 있는 상태이므로(get_free_page에서 이미 1로 set되었음을 기억하자. 즉 me-m_map이 이 시점에서 0일수는 없다.) 놀고있는 새로운 page에 내용을 그대로 복사하여 add-ress에 할당한다. 그림<7.14>는 copy-on-write과정을 도식화하였다.

2) 복사할 새로운 page가 없다.
3) 혼자서 page를 사용하고 있는데도 page fault가 발생했다면, 쓰기작업이 가능하지 않아서라고 판단하고 쓰기작업이 가능하도록 고친다.

7.7. LDT(local descriptor table)

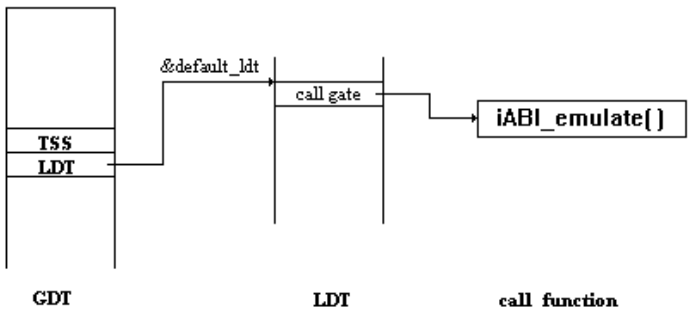
LDT는 task에 종속되며, GDT에서의 내용을 다른 task가 참조할수 있는 것과는 달리 LDT는 다른 task들로 부터 완전히 독립된다. LDT에 TSS descriptor와 LDT descriptor는 상주할수 없다.

current->tss.ldt는 GDT의 LDT descriptor를 가리키며, current->ldt는 LDT 차체를 가리킨다. 그림<7.15>은 default_ldt descriptor를 set하는 경우를 나타내었다.
LDT내의 *default_ldt* call_gate는 2.5.1절에서 set되었다. memeory상의 &default_ldt는 compiler에



<그림7.14> copy-on-write

의해 정해지지만, 결국 이곳이 LDT entry pointer가 된다. default_ldt에 의한 call gate는 리눅스 kernel내에서 ldt를 사용하는 유일한 경우이다.



<그림7.15> LDT 사용.

LDT의 최대크기는 8192(LDT_ENTRIES)×8byte(1개의 LDT entry)이지만, default_entry를 set하는 경우 LDT에는 LDT[0]만이 존재하므로 LDT 크기는 의미가 없다.

리눅스에서는 user가 ldt의 내용을 갱신할수 있도록 *modify_ldt* system call(kernel/ldt.c)을 제공하고 있다.

asmlinkage int sys_modify_ldt(int func, void *ptr, unsigned long bytecount)

func : 0 ldt의 내용을 읽는다.
1 ldt의 내용을 갱신한다.
ptr : func가 0이면 ldt를 가리키는 pointer가 return
func가 1이면 ldt entry정보를 담은 modify_ldt_ldt_s구조체 pointer로서 user가 제공하는 구조체 내용물에 의해 ldt가 갱신된다.

```
struct modify_ldt_ldt_s {
        unsigned int  entry_number;
        unsigned long base_addr;
        unsigned int  limit;
        unsigned int  seg_32bit:1;
        unsigned int  contents:2;
        unsigned int  read_exec_only:1;
        unsigned int  limit_in_pages:1;
};

/* Maximum number of LDT entries supported. */
#define LDT_ENTRIES     8192
/* The size of each LDT entry. */
#define LDT_ENTRY_SIZE  8
```

◆ 예전의 kernel에서는 default LDT로서 3개의 entry를 set한것으로 보인다.(hacker's Guide 5장 Linux Memory Management)
첫번째 entry는 NULL로, 두번째와 세번째 entry는 각각 현재 GDT의 entry 4, 5로 user segment descriptor들(code 및 data)이 GDT가 아닌 LDT에 상주한것으로 여겨진다.

7.8 swap영역

7.8.1 swap영역 생성

swap영역을 만드는데는 swap partition을 취하는 방법과 swap file을 이용하는 두가지 방법이 있다. swap영역은 *mkswap*명령에 의해 만들어지고, *swapon* 명령에 의해 swap 영역으로 사용할수가 있게 된다.
본인의 system의 file system table은 다음과 같다.

```
# cat /etc/fstab
/dev/hda2        swap     swap        defaults
/dev/hda3        /        ext2        defaults
none             /proc    proc        defaults
```

첫번째 disk의 2번째 partition이 swap영역으로 잡혀있다.

이제 /proc/meminfo안을 들여다보자.

```
# cat /proc/meminfo
        total:    used:    free:   shared:  buffers:
Mem:   7303168  3133440  4169728  1675264  1277952
Swap:  8769536        0  8769536
```

Swap영역이 8 MB가량 만들어져 있다고 나와있다.

만약에 여기에 swap file을 사용하여 swap영역을 늘이고 싶다면 다음과 같이 하면 된다.
swap file의 이름을 'happy'라고 하자.

```
# mkswap happy 8192
# swapon happy

# cat /proc/meminfo
        total:    used:    free:   shared:  buffers:
Mem:   7303168  3133440  4169728  1675264  1277952
Swap: 17154048        0 17154048
```

swap영역이 약 8 MB가량 늘어났음을 알수있다.

mkswap은 단지 swap영역의 첫 page의 맨끝에 “SWAP-SPACE”라고 10개의 문자를 적어넣을 뿐이다. 영역의 관리는 물리메모리처럼 page단위로 하기때문에 swap영역에 써넣을때도 page단위로 작업이 이루어진다. swap영역은 물리메모리의 연장이라고 볼수있다.

swapon은 SWAP-SPACE 문자열 바로 다음에 swap_map과 swap_lockmap이라는 두개의 bitmap을 둔다.
다음 swap영역에 대한 자료구조체를 보자.

```
static struct swap_info_struct {
        unsigned long flags;
        struct inode * swap_file;
        unsigned int swap_device;
        unsigned char * swap_map;
        unsigned char * swap_lockmap;
        int pages;
        int lowest_bit;
        int highest_bit;
        unsigned long max;
} swap_info[MAX_SWAPFILES];

#define  MAX_SWAPFILES  8
```

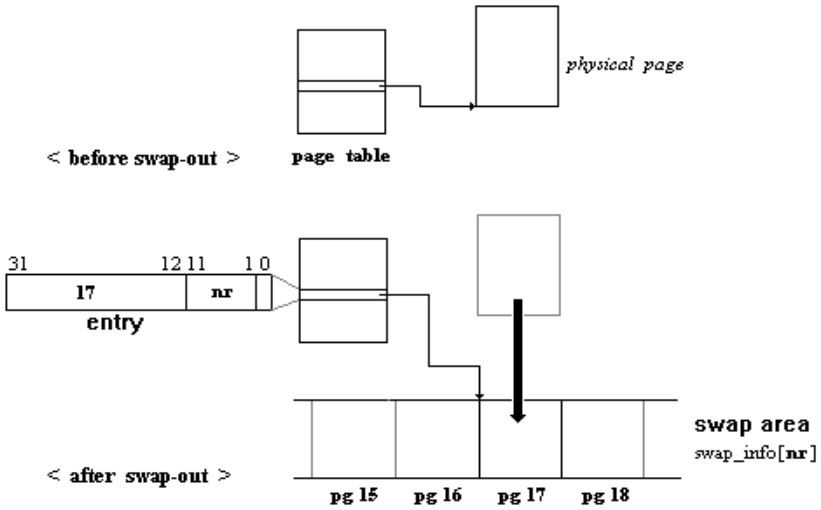
swap영역은 8개까지 지원되는 것을 알수있다. swap_map은 1byte가 swap page 1개를 가리킨다. 그래서 만약 swap page[103]가 100개(4096×100 byte의 swap영역)라면 swap_map의 크기는 100byte가 될것이다. 그래서 swap_map은 bytemap이라고 부르는 것이 적당할것이다. swap_lockmap은 1개의 swap page에 1개의 bit가 할당된다. swap-in을 할때 swap_lockmap에서 swap-in 시킬려는 page의 해당bit는 1로 set된다. swap-in이 끝나면 set된 bit는 clear된다. 1로 set할려는데 이미 set되어 있다면 다른 process가 읽기작업을 하고 있는것을 의미하므로 기다려야 한다. 물리메모리처럼 한개의 swap page를 여러 process가 공유할수 있다. 이 때 그 page의 해당 swap_map byte는 공유하는 process갯수만큼의 값이 된다. 이것이 swap_map이 bytemap인 이유이다. 물론 혼자서 사용하고 있다면 swap_map byte값은 1이 될것이다.

swap page도 또한 물리page와 같은 형태로 공유되기때문에 copy-on-write가 성립한다. fork시에 부모proces의 swap영역과 자식 process의 swap 영역은 공유된다. 즉, swap_map을 1증가시키는 것이 전부이다. 이후 둘중 한 process가 swap영역을 갱신하려고 하면 새로운 swap page를 할당해야 할것이다.

앞에서 page table에는 물리 page주소가 들어간다고 했다. 만약 어떤 page가 swap-out된다면 해당 page table entry에는 swap page주소가 들어가게 된다. swap page주소는 swap영역의 첫번째

103) 이것이 물리 page가 아니라는 것에 주의하자.

free page부터 0,1,2,...번호를 붙여나가는 것이다.
page table entry에 들어가는 swap page주소의 entry구조는 그림<7.16>과 같다.



<그림7.16> swap out

7.8.2 swap-out 전략

1) 최근에 major fault(task->majflt)가 많이 일어난 process일수록, 즉 swap-in을 많이 한 proce-ss일수록 swap-out대상에서 제외된다.
   swap-out시킬 process가 정해지면, process의 물리page들에 대해 다음과 같은 전략이 적용된다.
2) PG_ACCESSED bit가 set된 page는 swap-out시키지 않는다. 조만간 다시 access될 가능성이 높다고 보기 때문이다.
3) RESERVED page는 swap-out시키지 않는다. 이들 page들은 커널 코드를 위한 page들이다.
4) last_free_pages에 포함된 page는 swap-out시키지 않는다. 최소한 확보되어져야 할 물리 page로 본다.

참고로 1)번의 경우는 swap_out함수의 다음 코드를 보라. process에서 swap-out시킬 page의 양(swap_cnt)은 major fault에 의해 결정된다.

```
        /*
         * Determine the number of pages to swap from this process.
         */
        if(! p -> swap_cnt) {
            p->dec_flt = (p->dec_flt * 3) / 4 + p->maj_flt - p->old_maj_flt;
            p->old_maj_flt = p->maj_flt;

            if(p->dec_flt >= SWAP_RATIO / SWAP_MIN) {
                p->dec_flt = SWAP_RATIO / SWAP_MIN;
                p->swap_cnt = SWAP_MIN;
            } else if(p->dec_flt <= SWAP_RATIO / SWAP_MAX)
                p->swap_cnt = SWAP_MAX;
            else
                p->swap_cnt = SWAP_RATIO / p->dec_flt;
        }
```

물론 swap-out후에는 task->rss가 1 감소한다.

7.9 vmalloc 함수 (mm/vmalloc.c)

7.1절에서 논의되었던 kmalloc이 실메모리를 할당하는 것이라면, vmalloc은 kernel선형공간을 할당하기 위해 사용된다.

다음 구조체는 vmalloc으로 할당된 공간의 list를 만들기 위한 것으로, kmalloc의 page_descriptor에 대응하는 것이다.(vm_area영역구조체와 혼돈하지 말것.)

```
struct vm_struct {
        unsigned long flags;
        void * addr;
        unsigned long size;
        struct vm_struct * next;
};
```

다음 매크로는 물리메모리의 상한에서 8M떨어진곳부터 공간을 할당하겠다는 뜻이다. 할당영역과 할당공간사이에도 4K 만큼의 공간을 둔다. 이렇게 함으로써 할당공간을 벗어나서 참조하려고하면 곧 page_fault가 발생하게 된다. 번지지정이 영역을 초과하는지를 확인하기위해 사용된다.

```
#define VMALLOC_OFFSET  (8*1024*1024)

void * vmalloc(unsigned long size)
{
            /*   중   략   */

1)      addr = (void *) ((high_memory + VMALLOC_OFFSET) &
                ~(VMALLOC_OFFSET-1));
2)      area->size = size + PAGE_SIZE;
        area->next = NULL;
3)      for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
                if (size + (unsigned long) addr < (unsigned long) tmp->addr)
                        break;
                addr = (void *) (tmp->size + (unsigned long) tmp->addr);
        }
        area->addr = addr;
4)      area->next = *p;
        *p = area;
5)      if (do_area(addr, size, alloc_area_pages)) {
                vfree(addr);
                return NULL;
        }
6)      return addr;

}
```

1) 물리메모리 상한 8M지점부터 할당을 시도한다.
2) 할당공간과 할당공간사이에 4K 공간을 둔다.
3) size가 vm_area영역을 침범하지 않는지 확인한다.
   침범할 경우 vm_area영역밖으로 시작주소를 다시 할당한다.
4) 할당영역 list에 추가한다.
5) 할당된 가상공간에 물리메모리를 부여한다.
6) 할당공간의 시작주소를 return한다.

```
static int do_area(void * addr, unsigned long size,
        int (*area_fn)(unsigned long,unsigned long,unsigned long))
{
              ..................................
```

```
1)      dindex = (TASK_SIZE + (unsigned long) addr) >> 22;

                ....................................
2)              if (area_fn(dindex, index, i))
                        return -1;

                ....................................
}
```

1) kernel선형공간을 할당하므로 TASK_SIZE을 기준으로 할당된다.
2) alloc_area_pages 함수 수행.

```
static int alloc_area_pages(unsigned long dindex, unsigned long index, unsigned long nr)
{
        unsigned long page, *pte;

1)      page = swapper_pg_dir[dindex];
        if (!page) {
                page = get_free_page(GFP_KERNEL);
                if (!page)
                        return -ENOMEM;
                if (swapper_pg_dir[dindex]) {
                        free_page(page);
                        page = swapper_pg_dir[dindex];
                } else {
                        mem_map[MAP_NR(page)] = MAP_PAGE_RESERVED;
2)                      set_pgdir(dindex, page | PAGE_SHARED);
                }
        }
                  ..................................

}
```

1) swapper_pg_dir은 kernel의 page directory이다. 할당할려는 영역의 page dir entry(page table)이 없으면 page table을 할당한다.
2) page table pointer는 swapper_pg_dir뿐만 아니라, 모든 process page dir의 kernel부분[104]에

104) 모든 page directory의 entry 768 ~ 1023은 kernel공간을 위한 부분이다.

set된다.

8장

시스템 콜

8.1 초기화

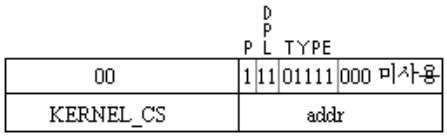
system call은 sched_init(sched.c)에서 초기화 된다.

```
set_system_gate(0x80,&system_call)   /* in sched_init */
```

리눅스에서는 INT 0x80을 system call을 위해 사용한다.
위의 매크로는 0x80 interrupt를 위한 gate descriptor를 만든다. &system_call은 handler의 주소에 해당한다. head.S의 **_idt영역**에서 0x80번째에 이 descriptor를 넣으면 int 0x80 명령에 의해 handler가 수행된다. (물론 이미 앞에서 lidt명령어에 의해 이 영역을 interrupt descripto -r를 위해 사용할 것을 CPU에게 알려주었다. 2장 참고)

```
#define set_system_gate(n,addr) \          /* include/asm/system.h */
        _set_gate(&idt[n],15,3,addr)       /*  descriptor를 만든다. */
```

n은 interrupt번호이며, 15는 type, 3은 dpl, addr은 interrupt handler 주소이다. 따라서 interru-pt descriptor는 그림<8.1>과 같은 모양이 된다.



<그림8.1> interrupt gate descriptor

dpl 3은 user process에 의한 접근이 가능한 것을 의미하며, type 15는 IF flag bit를 clear시키는 것이 불가능함을 나타내고 있다.

다음 매크로는 system call을 위한 함수를 만들어 준다.(include/linux/unistd.h) syscallX에서 X는 함수의 parameter수이다.

```
#define _syscall0(type,name) \     /*  include/linux/unistd.h  */
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \      /* eax에 들어간다.105)  */
if (__res >= 0) \
        return (type) __res; \
errno = -__res; \
return -1; \
}
```

예를 들어, *fork* system call의 경우 이 매크로를 사용한다.

```
        static inline _syscall0(int,fork)    /*  init/main.c의 서두에서  */
```

그래서 위 syscall매크로의 type name(void)이 int fork(void) 가 되는 것이다.

이후 fork system call number인 2와 함께 int 0x80 이 호출되고, syscall.S내의 _system _call: 부분이 수행된다. _system _call: 코드에 대해서는 8.2절에서 논의된다.

---

105) "0" 는 operand 0("=a")와 동일한 레지스터인 eax에 system call number를 넣음을 의미한다.

```
#define __NR_fork                2
```

user 프로그램에서는 단지 프로그램 서두에

```
#include <unistd.h>
```

라고만 해줌으로써 system call routine(kernel내의 sys_fork함수)이 호출된다. 이것은 library에 의해 이루어진다. 그러나, kernel내에서 sytem call호출을 위해서는 위의 inline함수로 함수정의가 있어야한다. inline함수는 매크로처럼, 호출한 fork()의 자리에 들어가게 되며, 함수 cal -l에 따른 부하를 줄여 수행속도를 높여준다.[106]
위의 inline함수는 *int* 명령에 의해 system_call routine을 호출한다.

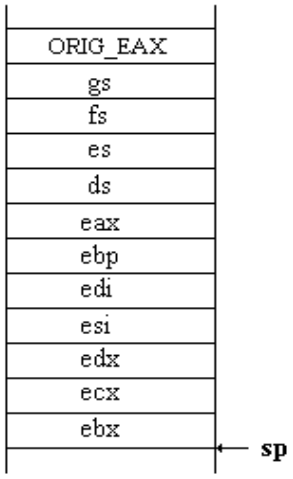
8.2 system call routine

```
.align 4
_system_call:
1)      pushl %eax                      # save orig_eax
        SAVE_ALL
2)      movl $-ENOSYS,EAX(%esp)
        cmpl _NR_syscalls,%eax
        jae ret_from_sys_call
3)      movl _current,%ebx
        andl $~CF_MASK,EFLAGS(%esp)     # clear carry - assume no errors
        movl $0,errno(%ebx)
        movl %db6,%edx
        movl %edx,dbgreg6(%ebx)  # save current hardware debugging status
4)      testb $0x20,flags(%ebx)         # PF_TRACESYS
        jne 1f
5) ==>  call _sys_call_table(,%eax,4)
6)      movl %eax,EAX(%esp)             # save the return value
7)      movl errno(%ebx),%edx
        negl %edx
        je ret_from_sys_call            /* error 없으면 , CF=0  */
```

106) Using and Porting GNU CC for Version 2.5
5.28절 An Inline Fuction is As Fast As a Macro

```
        movl %edx,EAX(%esp)
        orl $(CF_MASK),EFLAGS(%esp)     # set carry to indicate error
        jmp ret_from_sys_call
.align 4
1:      call _syscall_trace
```

1) system call number(ORIG_EAX)와 함께 레지스터값들을 stack에 보관한다.



<그림8.2> 현재 상태를 stack에 저장

- ds,es ← KERNEL_DS
- fs ← USER_DS

2) default error로 Not system call(ENOSYS)을 set한후, system call범위에 sytem call number가 있는지 확인후, 범위를 벗어나면 error값과 함께 그냥 return.

3) carry flag bit가 No set로 초기화. 뒤에서 system call후 error가 발견되면 set되어진다. 현재 process의 task struct에서 error값의 0로 초기화와 debug state register값을 보관.

4) process의 EFLAG값에 PF_TRACESYS(process flag-trace syscall)가 set되어 있으면, *sys-call_trace*함수 호출후 TASK_STOPPED상태, signal받으면 다시 running상태가 된다.

5) sys_call_table(sched.c)을 사용하여 해당 *sys*_함수(fork system call의 경우는 sys_fork함수) 호출.[107]

함수에 전달되는 parameter는 함수정의에 따라 그림<8.2>의 sp에서부터 차례로 전달된다.

fork와 exec system call처럼 stack값의 전체가 전달되는 경우와 sp지점부터 ebx, ecx, edx들만이 전달되는 경우가 있다.

```
sys_fork(struct pt_regs regs)
sys_open(const char * filename,int flags,int mode)
```

fork의 경우 sp에서 시작하여 다음과 같은 구조체[108]에 값이 들어가고 이 구조체가 parametor에 전달된다.

```
/* this struct defines the way the registers are stored on the
   stack during a system call. */

struct pt_regs {
  long ebx;
  long ecx;
  long edx;
  long esi;
  long edi;
  long ebp;
  long eax;
  unsigned short ds, __dsu;
  unsigned short es, __esu;
  unsigned short fs, __fsu;
  unsigned short gs, __gsu;
  long orig_eax;
  long eip;
  unsigned short cs, __csu;
  long eflags;
  long esp;
  unsigned short ss, __ssu;
};
```

한편 open의 경우는 stack에 저장된 ebx, ecx, edx들이 filename, flag, mode에 차례로 값이 전달된다.

---

107) foo(,eax,4)는 foo + eax*4 를 나타낸다.
108) include/linux/ptrace.h

6) return값(fork의 경우 PID)을 stack에 보관.

7) error값이 있는지 확인. carry flag bit가 set이면 error.(neg명령어 참고) eflag를 task stru-ct에 보관.

다음은 sys_함수 수행후 system call로부터 user mode로 복귀하는 과정이다.

```
ret_from_sys_call:
1)      cmpl $0,_intr_count
        jne 2f
        movl _bh_mask,%eax
        andl _bh_active,%eax
2)      jne handle_bottom_half          /* bottom half routine 수행 */
9:      movl EFLAGS(%esp),%eax          # check VM86 flag: CS/SS are
3)      testl $(VM_MASK),%eax           # different then
        jne 1f
4)      cmpw $(KERNEL_CS),CS(%esp)      # was old code segment supervisor ?
        je 2f
1:      sti
5)      orl $(IF_MASK),%eax             # these just try to make sure
        andl $~NT_MASK,%eax             # the program doesn't do anything
        movl %eax,EFLAGS(%esp)          # stupid
6)      cmpl $0,_need_resched
        jne reschedule
7)      movl _current,%eax
        cmpl _task,%eax                 # task[0] cannot have signals
        je 2f
        cmpl $0,state(%eax)             # state
        jne reschedule
        cmpl $0,counter(%eax)           # counter
        je reschedule
        movl blocked(%eax),%ecx
        movl %ecx,%ebx                  # save blocked in %ebx for signal handling
        notl %ecx
        andl signal(%eax),%ecx
        jne signal_return
2:      RESTORE_ALL
```

1) interrupt handler가 수행중에는 bottom half routine을 실행시키거나 scheduling을 하지 않는다.
2) bottom half rountine을 실행시킨다.
3) 가상 8086mode인지 확인.
4) kernel mode에서의 system call호출이면 No scheduling
5) IF flag의 set와 NT flag의 no-set 재확인.(sched_init에서 이미 NT bit clear)
6) need_resched변수가 set되었으면 scheduling.
7) 이후 현재 process가 signal을 받았는지 확인한다.
   받은 signal이 block된 것이 아니면, signal_return 코드로 가서, signal을 처리한다.
   init_task는 signal을 받을수 없으므로 확인작업 없이 복귀한다.
   - process의 state가 TASK_RUNNING이면 scheduling.
   - counter가 0이면 timeslice를 모두 사용한 것이므로 scheduling.
9) RESTORE_ALL
   stack의 register들을 다시 load시키고, iret명령을 사용하여 system call을 호출한 곳(user mode)으로 복귀한다.

8.3 STACK

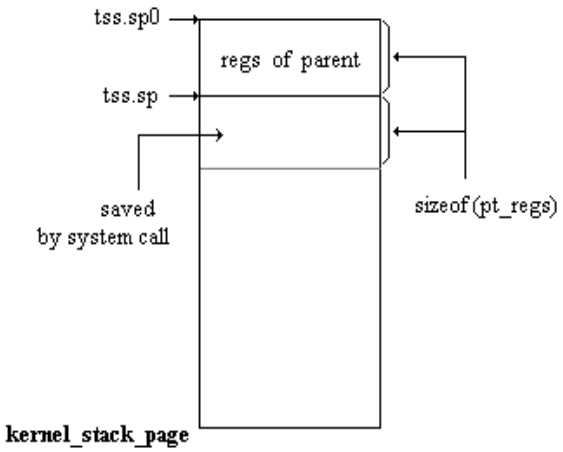
8.3.1 kernel stack page

리눅스에서 stack은 user mode에서는 process의 user선형공간의 끝에 존재하며, kernel mode로 진입하면 fork시 할당받은 1 page의 실메모리가 stack으로 사용된다.
7.5.3절의 그림<7.12>를 보면 환경변수들이 선형번지 3G 바로아래에 놓인다는 것을 알수 있다. 그래서 선형공간에서는 전형적인 stack의 형태를 갖추고 있음을 알수 있다. stack point -er는 다음 두 변수에 들어가게 된다.(load_aout_binary함수 중에서) regs->esp가 user program이 사용하는 stack pointer가 된다.

```
current->start_stack = p;
regs->esp = p;                  /* stack pointer */
```

kernel mode에서의 stack의 크기는 1 page로 고정되어 있다. 그리고 윗쪽에는 fork시 부모

process에게 전달받은 register값들을 저장한다. 이것은 tss.sp0(특권레벨 0의 stack pointer)에 저장된다. sp0는 가상 86모드에서 사용된다. stack page의 1 PAGE - sizeof(pt_regs) 지점부터 각 processs가 사용할 stack이 시작된다.



<그림8.3> kernel mode stack

fork system call의 코드를 보자.

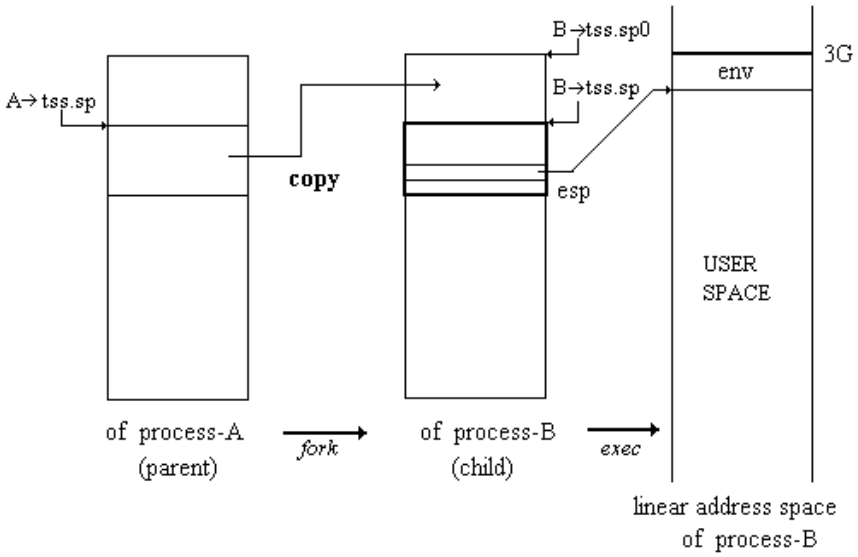
```
if (!(p->kernel_stack_page = __get_free_page(GFP_KERNEL)))
   ......................................
p->tss.ss = KERNEL_DS;
   ......................................
p->tss.ss0 = KERNEL_DS;
p->tss.esp0 = p->kernel_stack_page + PAGE_SIZE;
p->tss.tr = _TSS(nr);
childregs = ((struct pt_regs *) (p->kernel_stack_page + PAGE_SIZE)) - 1;
p->tss.esp = (unsigned long) childregs;
p->tss.eip = (unsigned long) ret_from_sys_call;
*childregs = regs;  /* 부모 process의 register값들 저장 */
```

8.3.2 stack의 사용

그림<8.4>에 fork와 exec에 의해 stack이 사용되는 것을 나타내었다. 우선 fork에서 부모 process A의 register값들은 system call의 도입부에 SAVE_ALL에 의해 A->tss.sp에서 부터 저장된다. 이후 자식 process B를 위한 stack page가 만들어지고 윗부분에 자신의 stack에 있는 register값들을 복사한다. 자식 process에서 exec가 호출되면 역시 SAVE_ALL 에 의해 자식 process의 register값들이 B->tss.sp에 저장되고 kernel stack에는 user 선형공간 stack의 위치가 저장된다.

```
regs->esp = p;                  /* stack pointer */
```

이후 user process가 수행되면서 tss.sp(kernel stack page pointer)에 의해 자신의 선형공간 stack을 사용할수 있다. 그리고 kernel stack의 내용은 해당 process가 종료될때까지 활용될것이다.



<그림8.4> stack의 사용

user page는 swapping이 가능하나 kernel page는 swapping이 되지 않는다. stack의 경우도마찬가지이다.
참고로 init task의 kernel stack page는 init_kernel_stack[1024](sched.c)이다.

```
static unsigned long init_kernel_stack[1024];
```

9장

리눅스 파티션

리눅스 partition이라고 제목을 붙였지만 리눅스 system에 국한된 내용은 아니다. 리눅스 system은 DOS의 fdisk에 의해 분할된 partition을 그대로 사용하기 때문이다.

9.1 물리 sector와 논리 sector

hard disk는 여러장의 floppy diskette을 겹쳐놓은것과 같은 형태로 되어있다. head는 0에서부터, cylinder는 0에서부터, sector는 1에서 부터 물리번호가 붙는다. 따라서 MBR(Master Boot Record)은 head 0, cylinder 0, sector 1에 위치하게 된다. 그런데 대부분의 system(운영체제)은 disk관리를 용이하게 하기위해 논리번호에 의한 논리 sector라는 것을 사용한다. 논리번호란 0에서 부터 disk의 마지막까지 번호를 붙인것이다. 논리번호는 cylinder진행순으로 붙여나간다.
예로서, head 12, track당 sector가 35인 disk의 논리번호를 붙이는 방법은 다음과 같다.

1) head 0, cylinder 0, sector 1은 논리 sector 0
2) head 1, cylinder 0, sector 1은 논리 sector 35
3) head 2, cylinder 0, sector 1은 논리 sector 70

.....

4) head 11, cylinder 0, sector 1은 논리 sector 385 (=11×35)

5) head 0, cylinder 1, sector 1은 논리 sector 420 (=385+35)

이것은 disk를 *formatting*할때와 동일한 순서인데, cylinder순으로 진행되는 것은 disk arm이 이동하는 시간을 줄이기 위해서다.

9.2 setup system call

setup system call은 DOS partition을 포함한 현재 system에 설치된 모든 hard disk partition정보를 구한다. 이것은 MBR에 있는 partion table을 읽어들이는 것에 의해 이루어진다. partition table은 4개의 partition record로 나누어진다. MBR의 0x1BE에 위치한 이들 record의 구조(genhd.c)는 다음과 같다.

```
struct partition {
        unsigned char boot_ind;         /* 0x80 - active */
        unsigned char head;             /* starting head */
        unsigned char sector;           /* starting sector */
        unsigned char cyl;              /* starting cylinder */
        unsigned char sys_ind;          /* What partition type */
        unsigned char end_head;         /* end head */
        unsigned char end_sector;       /* end sector */
        unsigned char end_cyl;          /* end cylinder */
==>     unsigned int start_sect;        /* starting sector counting from 0 */
        unsigned int nr_sects;          /* nr of sectors in partition */
};
```

partition에는 3가지 종류가 있다. primary partition, extended partition, logical partition, primary partition은 한개의 disk에 4개까지 가능하다. 이것은 disk한개에 4개의 운영체제를 깔아서 사용할 수 있음을 의미하기도 한다. 더 많은 partition이 필요하면 primary partition을 여러개의 logical partition을 가진 extended partition으로 사용하면 된다. 이것에 대해서는 LILO패키지에 포함된 LILO USER's guide를 참고하기 바란다.
extended partition은 5번부터 시작한다. 즉 hda5, hdb5..와 같이 말이다.

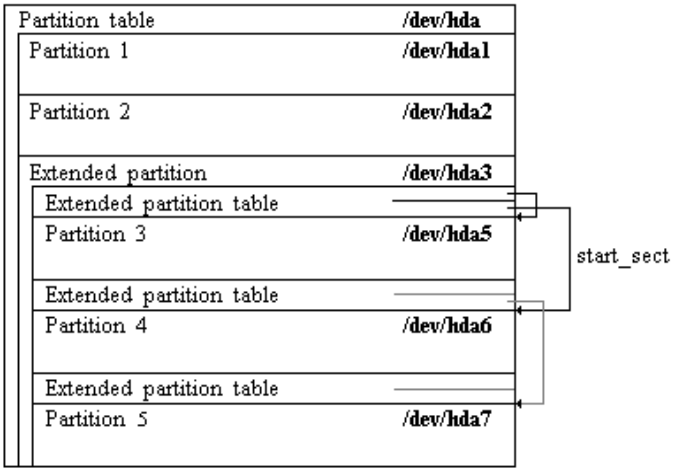
```
#define EXTENDED_PARTITION 5
```

9.3 gendisk 구조체

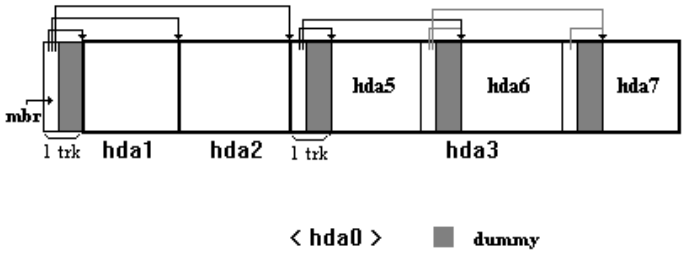
```
struct gendisk {
        int major;                      /* major number of driver */
        char *major_name;               /* name of major driver */
        int minor_shift;                /* number of times minor is shifted to
                                           get real minor */
        int max_p;                      /* maximum partitions per device */
        int max_nr;                     /* maximum number of real devices */

        void (*init)(void);             /* Initialization called before we do our thing */
        struct hd_struct *part;         /* partition table */
        int *sizes;                     /* size of device in blocks */
        int nr_real;                    /* number of real devices */

        void *real_devices;             /* internal use */
        struct gendisk *next;
};

struct hd_struct {
        long start_sect;
        long nr_sects;
};
```

위 구조체는 kernel이 관리하는 각 partition마다 가지고 있는 partition정보구조체이다. part->start_sect에는 partition이 시작하는 *절대 sector번지*가 들어간다. 그런데 disk에 있는 partition table의 *start_sect*는 시작sector의 상대번지값이다. 즉, partition table이 있는 sector를 기준으로 한 값이다. primary partition의 경우는 0을 기준으로 시작하니까 상관이 없지만 extened partition내의 logical partition은 각 partition마다 partition table을 가지고 있다.

그림<9.1>에는 /dev/hda5, /dev/hda6, /dev/hda7 세개의 logical partition이 있다. 각각 partition table을 가지고 있으며, 각 table에는 자신을 가리키는 record와 다음 logical partition을 가리키는 record가 있다.
이들 record들의 start_sect값은 Extended partition table이 위치한 곳을 기준으로 한다. 따라서 절대번지로 환산해서 part->start_sect에 들어가야 한다. 그림<9.1>에 나와있는 hda의 disk상의 실제모양은 그림<9.2>와 같다.



<그림 9.1> partition table



<그림 9.2> disk상의 partition

그림에서 나타나 있듯이 partition은 partition table이 있는 sector를 포함하여 1 track뒤에서 시작된다.

```
static void setup_dev(struct gendisk *dev)
{
          ..............................

1)      dev->init();    /* Executing hd_geninit() <hd.c> */
        for (drive=0 ; drive<dev->nr_real ; drive++) {
```

```
                current_minor = 1+(drive<<dev->minor_shift);
2)              check_partition(dev, major+(drive<<dev->minor_shift));
        }
          ..............................
}
```

1) hd_geninit함수는 system에 설치된 disk의 수를 확인하고 block size를 결정한다.
현재 kernel은 head가 16개가 넘는 ST-506(MFM방식,컴퓨터 구조이야기-이승택 저 675 p-age)은 지원되지 않는다.[109]

2) partition을 확인한다. inode구조체에 들어가는 i_dev의 구조(setup_dev코드내의 dev가 아님)는 다음과 같다.

dev = major << 8 | drive << 6 | partition번호

15~8 bit : major 번호
7~6 bit : minor 번호
5~0 bit : partition번호(0,1,2,...), 2^6= 64 : 최대 partition수

109) 현재 kernel Version 1.2.3 은 MFM방식을 포함한 RLL, ESDI, IDE, EIDE 방식들을 지원한다.

10장

디바이스 드라이버

10.1 서 론

device driver란 monitor(graphic card)를 비롯한 hard disk, floppy diskette, cd-rom, sound card, printer, modem등 device를 제어하는 interrupt handler를 말한다. 2장의 IRQ에 대해 논의 하면서 IRQ 5번이 발생하면, hd_interrupt라는 함수가 수행된다고 하였다. 이 함수가 hard disk device driver에 해당한다.

10.2 hard disk driver

그림<2.31>을 보면 user의 program에서 *read* system call을 호출되었을때부터 device driver가 data를 읽어들이는 과정이 나타나 있다. write system call을 호출되었을때도 같은 과정을 거쳐 disk에 쓰기 작업이 이루어진다.
device에 읽기, 쓰기 작업을 하기 위해서는 request라는 작업이 있어야 한다. 이것은 device controller에게 무슨 작업을 하라는 지시를 하는것을 말한다. 이 후 contoller에 의해 interrupt handler가 수행되어 작업이 이루어진다. 예를 들어 hard disk에서 data를 읽기 위해서는 먼저 disk controller에게 어느 sector를 읽으라는 명령을 해야한다. 이 후 disk controller는 disk에서

data를 읽어 contoller의 *buffer*에 담아두고, *irq신호에 의해* hard disk interrupt handler를 호출한다. handler는 buffer의 data를 kernel내부로 읽어들인다.
만약 쓰기작업이라면 쓰기작업명령을 받은 disk controller는 쓰기위한 준비를 한 후, handler를 호출한다. handler는 data를 disk controller의 buffer에 보내고, controller는 buffer의 data를 disk에 쓴다.
kernel은 *request queue* 라는 것을 관리하고 있다. 여러 process에 의해 같은 device에 request가 있다면, 순서를 만들어 대기시켜야 한다. 만약 해당 device에 대한 request quest가 비어있는 상태에서 한개의 request가 온다면 바로 수행되겠지만, request queue가 비어있지 않다면, 또는 어떤 process가 여러개의 request를 한꺼번에 보낸다면 당장 수행되는 것을 제외한 나머지는 request queue에서 대기하고 있어야 한다.

10.2.1 ll_rw_blk 함수

이 함수는 disk에서 buffer로 data를 읽고, 쓰는 역활을 담당하는 함수이다. 만약 nr이 2 이상이면, bh[0]에 해당하는 disk block를 대상으로 작업을 한후, bh[1], bh[2],.... 순으로 작업이 이루어질것이다.

```
void ll_rw_block(int rw, int nr, struct buffer_head * bh[])
{
                ..............................
1)      while (!*bh) {
                bh++;
                if (--nr <= 0)
                        return;
        };
                ..............................

        plugged = 0;
        cli();
2)      if (!dev->current_request && nr > 1) {
                dev->current_request = &plug;
                plug.dev = -1;
                plug.next = NULL;
                plugged = 1;
        }
        sti();
3)      for (i = 0; i < nr; i++) {
```

```
                if (bh[i]) {
                        bh[i]->b_req = 1;
                        make_request(major, rw, bh[i]);
                        if (rw == READ || rw == READA)
                                kstat.pgpgin++;
                        else
                                kstat.pgpgout++;
                }
        }
4)      if (plugged) {
                cli();
                dev->current_request = plug.next;
                (dev->request_fn)();
                sti();
        }
        return;
                  ..........................

}
```

1) bh->b_data가 있는 것에 대해서만 작업이 가능하다.
2) nr이 2 이상이면 request queue에 일단 모두를 대기시키기 위하여 dummy request를 둔다. current_request가 0이라는 것은 현재 request queue가 비어있음을 나타내는데, 이때는 맨 앞에 dummy request인 *&plug*를 두어 이 후 make_request함수에 의해 request queue에 추가시킬때 곧바로 request가 이루어지는 것을 방지한다.
3) make_request함수는 1개의 request를 받을때는 바로 request를 실시하지만, 여러개의 request를 받을때는, 또는 request queue에 대기중인 request가 있다면 request queue에 대기시킨다. add_request함수(in make_request)에서 다음 코드는 대기중인 request가 없는 경우(request q-ueue가 비어있는 경우) 곧장 request가 이루어지는 것을 나타내고 있다.

```
        if (!(tmp = dev->current_request)) {
                dev->current_request = req;
                (dev->request_fn)();
                sti();
                return;
        }
```

request_fn은 do_hd_request함수(hd.c)이다.

4) 2개이상의 request에 의해 request queue에 적재된 request를 이곳에서 실행시킨다.

10.2.2 make_request 함수

```
static void make_request(int major,int rw, struct buffer_head * bh)
{
            ..............................
        if ((major == HD_MAJOR
             || major == SCSI_DISK_MAJOR
             || major == SCSI_CDROM_MAJOR)
            && (req = blk_dev[major].current_request))
        {
                if (major == HD_MAJOR)
1)                      req = req->next;
                while (req) {
2)                      if (req->dev == bh->b_dev &&
                            !req->waiting &&
                            req->cmd == rw &&
                            req->sector + req->nr_sectors == sector &&
                            req->nr_sectors < 254)
                        {
                                req->bhtail->b_reqnext = bh;
                                req->bhtail = bh;
                                req->nr_sectors += count;
                                bh->b_dirt = 0;
                                sti();
                                return;
                        }

3)                      if (req->dev == bh->b_dev &&
                            !req->waiting &&
                            req->cmd == rw &&
                            req->sector - count == sector &&
                            req->nr_sectors < 254)
                        {
                                req->nr_sectors += count;
                                bh->b_reqnext = req->bh;
```

```
                                req->buffer = bh->b_data;
                                req->current_nr_sectors = count;
                                req->sector = sector;
                                bh->b_dirt = 0;
                                req->bh = bh;
                                sti();
                                return;
                        }

                        req = req->next;
                }
        }

/* find an unused request. */
4)      req = get_request(max_req, bh->b_dev);
            ..............................

/* fill up the request-info, and add it to the queue */
5)      req->cmd = rw;
        req->errors = 0;
        req->sector = sector;
        req->nr_sectors = count;
        req->current_nr_sectors = count;
        req->buffer = bh->b_data;
        req->waiting = NULL;
        req->bh = bh;
        req->bhtail = bh;
        req->next = NULL;
        add_request(major+blk_dev,req);
}
```

1) request queue의 맨 앞에는 dummy request가 있다.

2) 추가시킬려는 request의 작업대상 sector가 request queue에 있는 request의 **작업대상sector** 바로 다음에 위치한다면, 그 queue에 있는 request에 흡수시킨다. 즉, 한번의 request에 의해 두가지 request의 sector가 모두 작업이 이루어진다.
이것은 disk작업의 효율을 높이기 위해서이다. disk를 대상으로 하는 작업에서는 실제로 disk sector에서 data를 읽는 시간보다 원하는 sector로 disk arm이 움직이는데 소요되는 시간이 훨씬 많다. 따라서 '줄서기질서확립'에는 위배되지만 이렇게 같은 방향으로 가는 사람(request)

바로 뒤에 서는 새치기를 하는 것이다.

3) 이 곳도 2)와 같은 내용이다.

4) request queue에서 비어있는 자리를 구한다. request queue의 최대 자리수는 64개이다.

```
#define  NR_REQUEST  64
```

5) request queue에 끼워넣는다.

10.2.3 add_request 함수

request를 request queue에 끼워넣는다. 이것은 **elevator 알고리즘**을 사용하는 IN_ORDER 매크로에 의해 이루어진다. elevator 알고리즘은 앞에서 얘기했든 disk arm의 이동을 최대한 줄여보자는 전략에서 나온것이다.

```
/*
* This is used in the elevator algorithm: Note that
 * reads always go before writes. This is natural: reads
 * are much more time-critical than writes.
 */
#define IN_ORDER(s1,s2) \
((s1)->cmd < (s2)->cmd || ((s1)->cmd == (s2)->cmd && \
((s1)->dev < (s2)->dev || (((s1)->dev == (s2)->dev && \
(s1)->sector < (s2)->sector)))))
```

kernel 주석에는 READ 명령이 WRITE 명령보다 더 시급히(time-critical) 요구된다고 나와있다. elevator 알고리즘에 대한 보다 자세한 내용은 OPERATION SYSTEMS *DESIGN AND IMPLEM ENTATION* - ANDREW S.TANENBAUM 3.6 DISKS 를 참고하기 바란다.

10.2.4 hd_out 함수

do_hd_request함수는 request queue에 request가 있는지 확인하고, 읽기, 쓰기작업등에 따라 hd_out함수를 호출한다.

```
static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
                unsigned int head,unsigned int cyl,unsigned int cmd,
```

```
                void (*intr_addr)(void))
{
            ..............................

        if (!controller_ready(drive, head)) {
1)              reset = 1;
                return;
        }
2)      SET_INTR(intr_addr);
3)      outb_p(hd_info[drive].ctl,HD_CMD);
        port=HD_DATA;
        outb_p(hd_info[drive].wpcom>>2,++port);
        outb_p(nsect,++port);
        outb_p(sect,++port);
        outb_p(cyl,++port);
        outb_p(cyl>>8,++port);
        outb_p(0xA0|(drive<<4)|head,++port);
        outb_p(cmd,++port);
}
```

1) controller가 준비가 되어있지 않으면 reset = 1
   do_hd_reqeust에서 다시 시도할 것을 종용한다.
2) 이 때 set되는 interrupt 함수는 hd_interrupt에서 handler변수값이 된다. 다음 코드를 보자.

```
static void hd_interrupt(int unused)
{
     void (*handler)(void) = DEVICE_INTR;

     DEVICE_INTR = NULL;
     timer_active &= ~(1<<HD_TIMER);
     if (!handler)
             handler = unexpected_hd_interrupt;
     handler();   /* read_intr, write_intr 함수 */
     sti();
}
```

3) 이 부분이 disk controller에게 disk에서 controller buffer로 읽어들이라는 명령을 보내는 곳이다.

10.2.5 hard disk interrupt handler

이 후 IRQ 14번이 발생하고 interrupt handler인 hd_interrupt(hd_init함수)에 의해 **read_intr** 또는 **write_intr**이 수행된다. 이들은 controller buffer에서 kernel영역으로 data를 가져오거나, kernel영역에서 controller buffer로 data를 보낸다. 그리고, 이들은 한번의 request에 응답하고나면 do_hd _request를 호출하여 request queue에 다음 수행될 request가 있는지 확인한다.

10.3 line printer driver

printer에 data를 보내는 방법에는 **poll방식**과 **interrupt방식**이 있다. 일반적으로는 poll방식을 사용하는데 interrupt방식을 지원하는 병렬카드를 사용한다면, 그리고 program에서 ioctl system call의 SETIRQ명령으로 IRQ를 set한다면 interrupt방식을 사용할수도 있다.
다음 table을 보자.(include/linux/lp.h)

```
struct lp_struct lp_table[] = {
        { 0x3bc, 0, 0, LP_INIT_CHAR, LP_INIT_TIME, LP_INIT_WAIT, NULL, NULL, },
        { 0x378, 0, 0, LP_INIT_CHAR, LP_INIT_TIME, LP_INIT_WAIT, NULL, NULL, },
        { 0x278, 0, 0, LP_INIT_CHAR, LP_INIT_TIME, LP_INIT_WAIT, NULL, NULL, },
};
```

```
struct lp_struct {
        int base;
        unsigned int irq;
        int flags;
        unsigned int chars;
        unsigned int time;
        unsigned int wait;
        struct wait_queue *lp_wait_q;
        char *lp_buffer;
};
```

base는 부팅시 ROM내의 POST(자기진단 program)에 의해 확인되는 병렬 port이다.

```
#define LP_B(minor)     lp_table[(minor)].base          /* IO address */
#define LP_CHAR(minor)  lp_table[(minor)].chars         /* busy timeout */
#define LP_TIME(minor)  lp_table[(minor)].time          /* wait time */
#define LP_WAIT(minor)  lp_table[(minor)].wait          /* strobe wait */
#define LP_IRQ(minor)   lp_table[(minor)].irq           /* interrupt< 0 means polled > */
```

chars는 printer가 BUSY상태일 경우 얼마나 기다릴지를 나타낸다.
data를 printer에 보내면 strobe line[110]이 낮아진다. **wait**는 이 line이 다시 high상태가 될때까지 기다리기 위해 소요되는 시간이다. strobe line에 대해서는 printer관련자료를 참고하라.

110) THE UNDOCUMENTED PC-FRANK VAN GULLUWE 709 page

11장

키보드 시스템

keyboard system의 경우 user가 terminal을 통해 보는 것에 비하면 내부에서는 꽤나 복잡한 과정이 이루어진다.

11.1 Terminal Modes

본절에서는 터미널의 속성(attribute)에 대해 설명한다. 터미날 속성은 input과 output이 제어되는 방법을 나타내다. mode에 대한 보다 상세한 내용이나 옵션에 대해서는 **GNU C library reference manual** Chapter 12. Low-Level Termianl Interface 를 참고하기 바란다.

11.1.1 Data type

struct termios의 구조는 다음과 같다.(include/linux/termios.h)

```
tcflag_t c_iflag   : input mode를 위한 flag
tcflag_t c_oflag   : output mode를 위한 flag
tcflag_t c_cflag   : control mode를 위한 flag
tcflag_t c_lflag   : local mode를 위한 flag
```

11.1.2. Input mode

1) IGNBRK : 이 bit가 set되면 break조건이 무시된다.
   break조건은 1 byte보다 긴 일련의 zero값 bit들로서 비동기 직렬 data전송에 사용된다.

2) BRKINT : 이 bit가 set되고 IGNBRK가 set되지 않으면, break조건은 터미날 입력,출력 큐(queue)를 clear시키고, 현재 터미날에 대해 forground인 process group에게 SIGI-NT 시그날을 보낸다.
   BRKINT와 IGNBRK가 둘다 set되어 있지 않으면, break조건은 PARMRK가 set되어 있으면 application에 '\0'한 문자를 보내며, PARMRK가 set되어 있지 않으면 application에 '\377','\0','\0'를 연속적으로 보낸다.

◆ break조건 (BREAK condition)이란?

break조건은 serial interface에 따라 다양한 방법에 의해 keyboard device driver에 전달된다.[111] 일반적으로 터미날에는 Break키가 있다. 비동기 직렬 data전송에서는 1byte이상의 zero bit들을 보내는 것으로서 Break키를 대신한다.

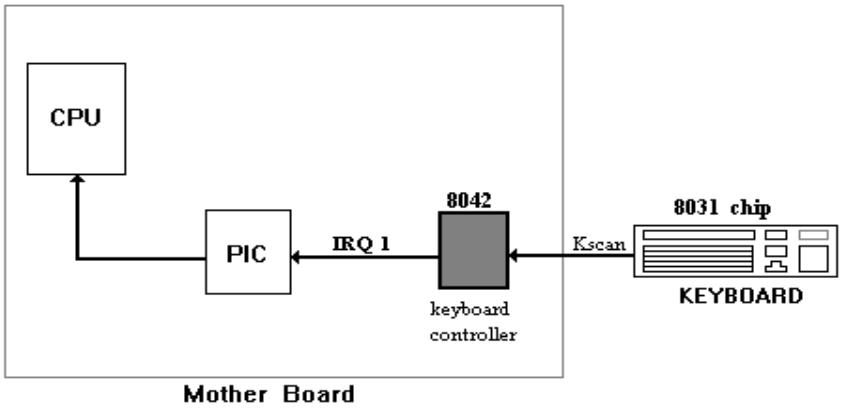
11.2 keyboard에서의 Key 입력과정

키보드는 키보드와 시스템사이를 연결시켜주는 8042 chip을 가지고 있다. 이 microcontroller는 data의 타당성 check와, Kscan code를 scan code로 바꾸어 주는 역활을 한다.[112]
소문자 "o"를 누를경우 code값 전달과정은 다음과 같다.

1) 우리가 "o"키를 누르면 8031 chip이 Kscan code값인 0x44를 mother board상의 8042 chip에 전달한다.[113]
2) 8042 chip은 0x44를 scancode값인 0x18로 바꾸어 output buffer에 둔다.[114]
   PIC에는 IRQ 1신호가 전달된다. keyboard hander가 기동한다.
3) keyboard handler는 drivers/char/keyboard.c내의 **keyboard_interrupt()**로서 다음 코드에 의

---

111) Advanced Programming in the Unix Environment-Stevens 335page참조.



<그림11.1> keyboard controller

해 buffer의 값을 읽어 들인다.

```
scancode = inb(0x60);
```

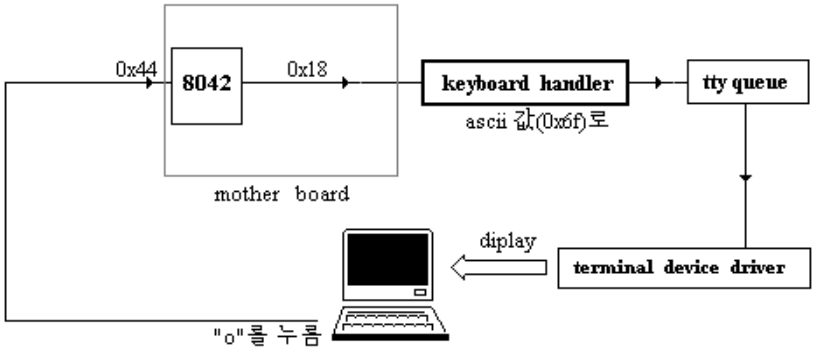
4) 읽혀진 scancode는 drivers/char/defkeymap.c의 keymap[]배열을 이용하여, ascii값으로 바꾼어 tty_queue에 저장된다.
5) 그후 terminal device handler가 이 queue의 내용을 읽어 terminal의 모니터화면에 뿌린다.
6) key를 놓을때도 또한 handler가 수행되며, 그 때 전달되는 scancode의 값에 대해서는 **show-key**라는 utility를 사용하여 알아볼수 있다. key를 한번 눌렀다 뗄때마다 두개의 값이 보일것이다. 두번째 값이 key를 놓을때(release) 전달되는 scancode값이 된다. 예를 들어 key를 누를때 "o"의 scancode는 0x18이다. 그러나 key를 놓을때는 0x98이 된다.

11.3 keyboard handler 수행과정

handler는 keyboard_interrupt()함수이다.

---

112) The Undocumented PC-Wesley 8장 KEYBOARD SYSTEM 참조.
113) kscan code table은 IBM PC AT Technical Reference Scan code set 2,3(163,166page)를 참조.
114) defkeymap.map keycode값 참조.



<그림11.2> code값 전달과정

다음은 handler 수행과정이다.

1) 키보드를 disable시킨다.

2) scancode를 읽어들인다.

3) bh_active에 KEYBOARD_BH bit를 set한다.
handler가 return(ret_from_syscall<syscall.S>)할때 이 bit에 의해 do_bottom_half routine이 수행된다. irq handler의 나머지 반이란 의미인 이 routine은 do_bottom_half()<irq.c>라는 함수로서 bh_active에 set된 bit들에 해당하는 모든 half routine을 수행한다. half routine이 있는 DEVICE는 include/linux/interrupt.h을 참고하라.
자주 발생하는 DEVICE일수록 번호가 작아서 do_bottom_half()에서 가장 먼저 수행되도록 한다. keyboard에 해당하는 half routine는 kbd_bh()함수이다.
예를 들어, console을 change시키는 경우에 KEBORAD_BH bit를 setting시킴으로써 ret_from_syscall이 수행(ret_from_syscall은 system call이 return하거나, 다른 device의 irq handler가 return할때 수행되어진다.)되는 과정에서 kbd_bh()가 호출되고, 결과적으로 consol-e을 change하는 작업이 이루어 지도록 한다.

4) scancode값의 0xfa,fe,ff,등은 keyboard가 내부 시스템에게 scancode의 형식으로 보내는 명령들이다. 다음과 같은 것들이 있다.[115]

115) 보다 자세한 내용은 IBMPC AT Technical Reference-성안당 4장 키보드 참조.

① ACK (0xfa)
키보드는 Echo나 Resend명령 이외의 입력에 대해 ACK(Acknowledge)로 응답한다.
ACK를 보내는 중에 인터럽트가 걸리면 키보드는 ACK를 버리고 새명령을 받아들이고 응답한다.

② Overrun (0x00 or 0xff)
buffer용량을 초과하면 overrun문자가 buffer에 들어가서 마지막 코드를 대체한다.

③ Key Detection Error(0x00 or 0xff)
키보드내의 조건으로 스위치 폐쇄를 확인할 수 없는 상황일때 보낸다.

④ Resend (0xfe)
키보드가 부적합한 입력을 받거나 패리티에러(parity error)가 발생하면 Resend명령을 보낸다.

나머지 내용들이다.

⑤ BAT Failure Code (0xfc)
⑥ Echo (0xee)
⑦ keyboard ID(0x83ab)
⑧ BAT Completion Code (0xaa)

5) 현재 console의 tty_struct pointer획득.

```
tty = TTY_TABLE(x);
```

x에 값을 넣어 특정 console의 tty_struct를 위한 pointer를 구할수 있다. 그러나 0을 넣으면 현재 console에 대한 tty_struct가 획득된다.

6) raw mode인지 cooked mode인지를 확인한다.[116]

raw mode란 user가 입력한 키값을 다루는데 있어 문자지향(character-oriented)방식으로서 입력된 문자를 그대로 application에 보내진다. 그러나 **cooked mode**는 줄지향(line-oriented) 방식으로서 Carrige Return을 비롯한 CTRL-S,CTRL-Q,CTRL-D(end of file)등의 특수한 기능들에 대해 반응하도록 되어있다.

mediumraw mode라는 중간단계가 있다.

7) raw mode이면 곧바로 scancode를 queue에 넣는다.

queue는 tty_struct구조체에서의 read_q로서 이 곳에 들어간 값은 나중에 terminal device driver에의해 읽혀진다.

8) scancode값이 0xe0이면 이는 **기능key**(Prtscr,Insert,화살표등)[117]이며, 0xe1은 **pause** key에 대한 값이다. 이들은 key를 누를때나 놓을때 0xe0 또는 0xe1를 비롯하여 몇가지 scancode값을 같이 보낸다.

표<11.1>에는 Pause와 Print screen key를 예로 다루었다.

KEY	전달되어 오는 scancode(key down/key up)
Pause	e1 1d 45 e1 9d c5 / up시는 없음.
Print Screen	e0 2a e0 37 / e0 b7 e0 aa

<표11.1>

한 값이 전달될때마다 handler가 기동된다. 즉 Pause의 경우는 6번 handler가 수행된다. 다음 값을 받기위해 prev_scancode에 값을 넣고 handler를 마친다.

9) key가 현재 up상태인지 down상태인지를 check한다.

key를 눌렀을때 handler가 발생했다면 down상태일 것이고, key를 놓을때 발생했다면 up상태일것이다.

scancode 8 bit는 up될때(key release) set된다. 예를 들어 key를 누를때 "p"의 scancode는 0x19이다. 그러나 key를 놓을때는 0x99가 된다.

---

116) **Operating System Design and Implementation -TANENBAUM** 30page 참조.
참고로 ANDREW S. TANENBAUM씨는 MINIX를 만든 사람이며, 이 책 또한 MINIX를 중심으로 엮어나갔다.

117) function key는 포함되지 않음.

◆ 숫자앞에 0를 붙이면 이것은 octa(8진수)값이 된다. 즉 0200은 십진수로 200이 아니라 128이다.

10) e0와 e1값의 key를 처리한다. Pause key에 대한 것은 source에 붙은 주석을 참고하고, 여기서는 Print Screen key를 살펴본다. e0값을 전달하는 key들의 가지수에 대해서는 keyboard.c에서 E0_BASE에 대한 define문을 참고 하라.
E0_BASE 값인 96은 user가 정해진 keynumber범위밖에서 임의로 정할수 있다. keynumber는 keyboard에 IBM에 의해 붙여진 번호이며 이 값이 우리가 받는 scancode이기도하다. 그러나 기능 key의 경우는 key 한개가 여러개의 scancode를 발생하므로 리눅스에서 임의로 key-number를 재생(우리가 사용할 scancode를 재생)한다. 재생하는 방법은 e0_keys[128]배열을 이용한다.

여기서 우리는 defkeymap.c와 defkeymap.map 두 화일에 대해 살펴보자.

```
# mktable defkeymap.map > defkeymap.c
```

또는

```
# loadkeys --mktable defkeymap.map > defkeymap.c
```

에 의해 defkeymap.c가 생성된다.[118]
defkeymap.map은 keytable문법에 의해 생성되며,

keycode *keynumber* = *action*

의 형태로 되어있다.
defkeymap.c에는 key_map[16][128]의 배열이 들어있는데, [16]은 [0] ~ [15]까지 각각 표<11.2>와 같은 의미를 가진다.[119]

표<11.2>의 modifier는 interrupt.h에 있는 "KG_" set에 의해 표<11.3>과 같은 bit들의 조합으로 결정된다. 표<11.3>은 key_map[x][128]의 x에 들어갈 1 byte이다.

---

118) **loadkeys,mktable,keytables**의 manual page와, **keystrokes.howto**를 참조하라.

배 열 [x]	modifier
0	none
1	Shift
2	AltGr
3	Shift + AltGr
4	Control
5	Shift + Control
6	AltGr + Control
7	Shift + AltGr + Control
8	Alt
9	Shift + Alt
10	AltGr + Alt
11	Shift + AltGr + Alt
12	Control + Alt
13	Shift + Control + Alt
14	AltGr + Control + Alt
15	Shift + AltGr + Control + Alt

<표11.2>

Ctrl-R	Ctrl-L	Shift-R	Shift-L	Alt	Ctrl	AltGR	Shift
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

<표11.3>

Print Screen key 처리 과정은 다음과 같다.

① 첫 scancode 0xe0를 받으면, prev_scancode에 넣고 return.

② 다음값인 0x2a를 받으면,무시되고 return.

③ 다시 0xe0를 받으면, prev_scancode에 넣고 return.

④ 0x37을 받으면, e0_keys[]에서 EO_PRSCR값인 99를 찾아 scancode에 둔다.

⑤ 이 scancode값으로 key_map[]에서 0x001c값을 찾는다.

⑥ 상위 byte 0x00는 key_handler인 do_self()을 지정하며 이 함수에 의해 0x1c값이 tty qu-eue(read queue)에 저장된다.

⑦ key가 놓여질때도 같은 과정을 통해 do_self()는 수행되나 **up_flag가 set**되어있어, queu-e에 저장되지는 않는다.

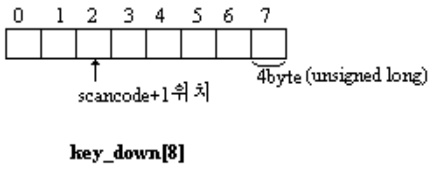
11) raw mode가 아니면서,그리고 e0,e1기능키가 아니면서 scancode가 E0_BASE값을 넘을수는

119) **keytables**의 manual page 참조.

없다. raw mode에서는 어떠한 값이든 queue에 들어갈수 있다.

12) rep == 0 이기 위한 조건.
① key가 놓여질때
② key가 방금 다운되었을때.

◆ key가 계속 down상태(user가 지속적으로 한 key를 누르고 있는 상태)에서는 rep가 0일수 없다.
◆ set_bit와 clear_bit.
key_down배열은 8×4byte size이며 bit수는 8×4×8=256 bits이다.
set_bit 매크로는 key가 down상태에서 scancode값이 24이면 25번째 bit를 set시킨다..
clear_bit 매크로는 key가 up상태에서 scancode값이 24이면 25번째 bit를 set시킨다..



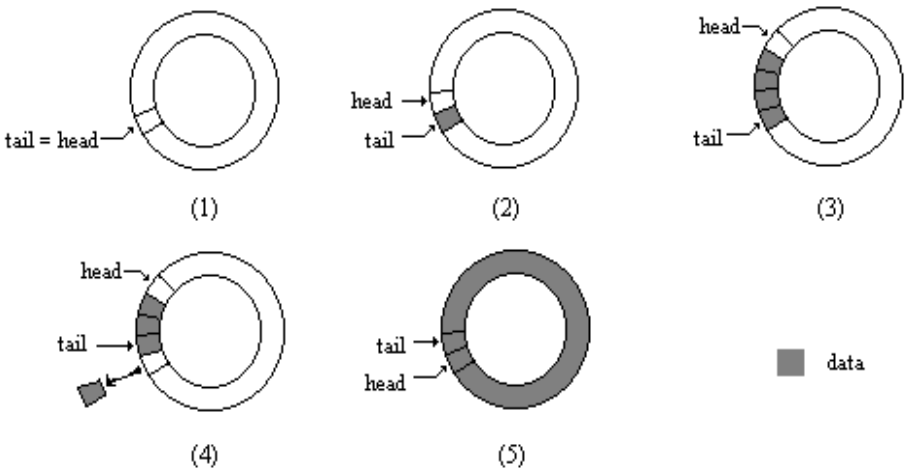
<그림11.3>

set_bit와 clear_bit 매크로는 해당 bit의 이전 값을 return한다.

13) mediumraw mode에서는 cooked mode에서처럼 다양한 기능 key에 대해 동작하지는 않지만, e0, e1기능키에 대해서는 동작한다.

14) cooked mode에서는 rep값이 0이거나 buffer queue가 비어 있으면 tty queue에 scancode를 넣는다. key가 계속 down상태에 있으면, rep는 0가 아니지만, buffer queue가 비어있으면, put_queue함수가 수행된다.

① tty queue는 원형 queue의 형태이다.
그림<11.4>는 원형 queue에 queue가 들어가는 것을 나타내고 있다.



tty Queue

<그림11.85>

(1) queue가 빈 초기상태. tail == head.
(2) queue 1개가 들어감으로써 head pointer가 1증가.
(3) queue 3개 들어감.
(4) queue를 1개 읽어 냄으로써 tail pointer 가 1증가.
(5) queue가 full상태. tail == head - 1

put_queue함수에 의해 tty queue에 keycode값이 들어간다.

② KT_LETTER == 0

15) enable keyboard.

11.4 kbd_bh 함수 수행과정

1) ledstate는 3bit에 의해 결정되며,그들은 kd.h의 표<11.4>과 같다.

현재 console의 ledstate값이 바뀌면 3개중 어느것이 눌려졌다는 것을 의미하므로, 0x60 port에 0xed및 ledstate를 보내어 확인한다.

해당 bit	상수(kd.h)	상수값	해당 key
bit 0	LED_SCR	1	Scroll Lock
bit 1	LED_NUM	2	Num Lock
bit 2	LED_CAP	3	Cap Lock

<표11.4>

2) console change 요구시 change_console()

Alt-F1에서 Alt-F10을 사용하여 console을 바꿀수 있는데, 이때 do_cons()가 수행되고, wanted_console에 0에서 9까지의 값이 들어가게 된다.

3) break조건발생.

termios struct의 input mode flag인 IGNBRK및 BRKINT를 참고하라.

BRKINT가 set되어있지 않으면, TTY_BREAK값이 queue에 들어가며, readq_flag bit가 set된다.

다음은 tty.h에 있는 break및 frame error,parity error에 관한 상수값들이다.

```
TTY_BREAK       1
TTY_FRAME       2
TTY_PARITY      3
TTY_OVERRUN     4
```

queue에 들어간 이러한 값들은 do_keyboard_interrupt() ⇒ tty_read_flush() ⇒ copy_to_cooked()가 수행되면서 queue에서 읽혀져 tty->char_error 변수에 들어가고, readq_flag 값이 확인되어 다음 queue(secondary queue)에 들어갈 적당한 값들을 결정한다.

4) do_keyboard_interrupt() <console.c>

① TTY_READ_FLUSH

readq에 있는 내용을 terminal이 열려있다면(getty라는 daemon에 의해 열린다.) 그 terminal의 모니터에 뿌려주기 전에, write queue에 옮기기 위해 ldisc[120] handler인 copy_to_

120) ldisc란 line disciplne의 약자이며, kernel내부에 존재하는 module로서, device driver와

cooked()[121]가 수행된다.

◆ daemon이란 system booting시 기동되어 background로 수행되는 것을 말한다. rc.local내에서 수행되는 cron, telnetd, rlogind등이 있을수 있다. telnetd와 rlogind는 sleeping하고 있다가 다른 host에서 telnet나 rlogin으로 접속을 시도하면 기동한다.

② console moniter가 일정시간이 지나면 blank(화면보호용으로 화면이 검게 사라짐)되어지는 것을 결정한다.

timer_table[ ].expires 값은 timer handler(do_timer)가 기동할때마다 check되어 0가 되면 blank가 동작한다. console_blanked == 1(blank_screen함수)은 현재 console이 blank상태임을 말한다.

5) fake_keyboard_interrupt()

buffer에서 scancode를 다 읽어들이지 않았으면 keyboard handler를 기동한다.

---

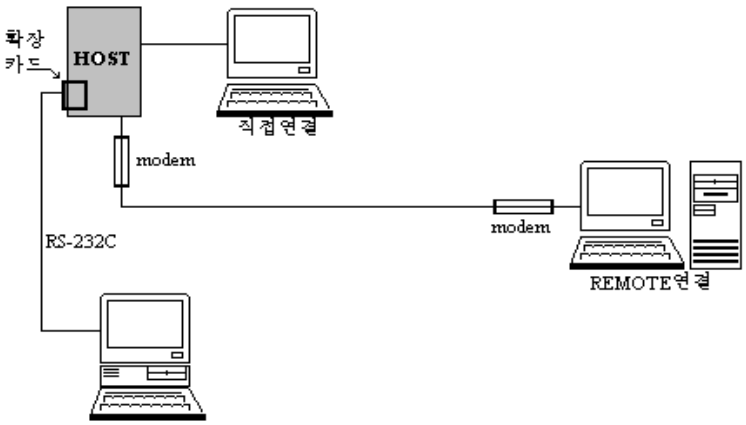
user process를 연결한다. terminal device driver와 함께 설명된다.

Unix Network Programming-Stevens 15.2 Termianl Line Displine참조.

121) tty_init() ⇒ tty_register_ldisc()에 의해 등록된다.

12장

터미날 시스템



<그림12.1> 다양한 터미날 연결.

12.1 서 론

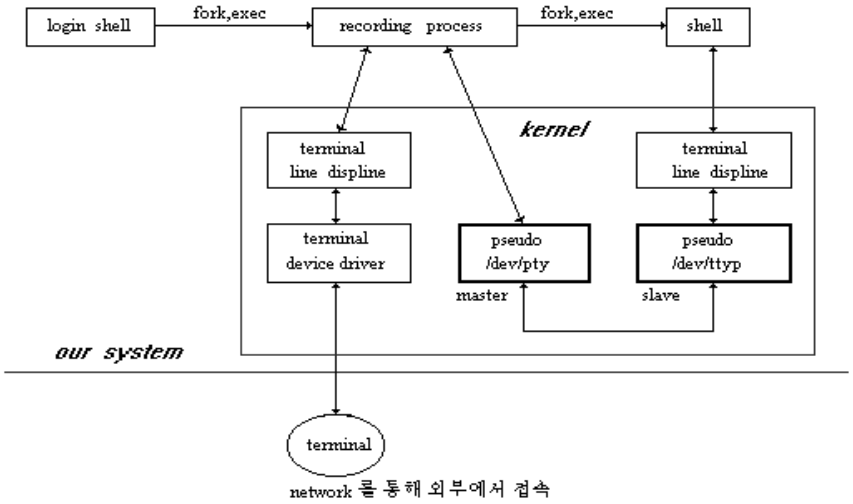
terminal이란 시스템본체와 직접 연결되거나, 입출력 확장카드 및 RS-232C회선을 사용하거나, modem을 통하여 본체와 연결시킴으로써 user가 시스템을 사용할수 있도록 해주는 입출력장치를 말한다.
시스템과 직접 연결된 terminal은 출력에 있어 주메모리에 display화면(더 정확히 말하면 VGA card의 메모리)이 memory map되어 있어 VGA card의 경우 주메모리의 0xB8000부터 data를 써넣으면, 그 값에 해당하는 문자가 monitor에 display된다. 이것은 MS-DOS에서 출력프로그래밍할때 흔히 사용하는 방법이다.
입출력 확장카드와 RS-232C를 사용하는 terminal의 경우는 출력과 입력이 RS-232C회선을 통해 이루어지므로 시스템본체와는 독립되어 있다고 할수있다. 이러한 terminal에는 intelligent terminal과 dumb terminal이 있는데, intelligent terminal이란 자체 메모리 및 processor(CPU)등을 가지고 있어 상당부분을 스스로 처리할수 있도록 되어 있는 terminal을 말하며, online시는 host의 영향을 많이 받지만, off line이 되면(host와의 단절) 독립된 객체로서 작동한다. dumb terminal의 경우는 단지 입출력 장치로서 보다 host에 종속적이다.
console은 일반적으로 시스템본체와 직접 연결되며, 일반적으로 system operator가 시스템 운용을 위해 사용하고 있는 terminal을 말한다.
modem을 이용한 terminal은 먼거리에서 전화선을 통해 terminal을 사용하는 경우이다.
리눅스의 /dev디렉토리에는 console, tty, ttyS, cua, pty등과 같은 device 문자화일들이 있다. /dev/console은 현재 사용중인 console terminal을 가리키며 /dev/tty1가 default이다. /dev/systty 또한 console과 같은 기능을 한다. 아래에 보다시피 세 파일은 major number는 물론이고, minor number가 같다.

```
# ls console systty tty0
crw-------   2 root    root    4,  0 May 13 21:06 console
crw-------   2 root    root    4,  0 May 13 21:06 systty
crw--w--w-   1 root    root    4,  0 jan 26 11:18 tty0
```

/dev/tty와 /dev/ttyS는 외부에서 시스템본체로 입력이 들어오기 위한 것으로, /dev/tty는 리눅스에서는 console을 위해 사용되며, /dev/ttyS는 RS-232C 또는 modem을 통하여 terminal 연결시 사용된다.[122](dial in) /dev/cua는 major numbor가 5로서 본체에서 modem을 이용하여 외부 host에 연결할때 사용된다. 즉, 본체에서 외부로 출력이 나가기 위한것이다.(dial out)
/dev/pty와 /dev/ttyp는 **가상(pseudo) terminal**에 사용된다. 가상 terminal은 실제 터미날이 아니라 외부에서 network를 통한 login이 있을 경우, network device 와 shell을 연결시키기 위한 것이다.[123] 즉, shell은 network device을 직접 terminal로 볼수가 없다. 그래서 *recording*

122) serial.faq 참조

*process*가 network device로부터 data를 받아 pty에 넘겨주게 된다. 이로써 pty는 recording process에게 terminal과 같은 역활을 한다. shell은 ttyp에게 data를 전달하거나 받게된다. 즉 shell에게는 pty가 terminal의 역활을 하는 것이다. 이렇듯 pseudo terminal을 위해서는 1쌍의 pseudo terminal device file을 사용한다. **/dev/pty**는 master, **/dev/ttyp**는 slave라고 한다.
recording process는 rlogind, telnetd server[124]등이 될수있다.
그림<12.2>는 STEVENS의 UNIX NETWORK PROGRAMMING표지를 리눅스 terminal device 이름으로 다소 편집한 것이다.[125]



<그림12.2> 가상 terminal의 구조

12.2 tty_init 함수

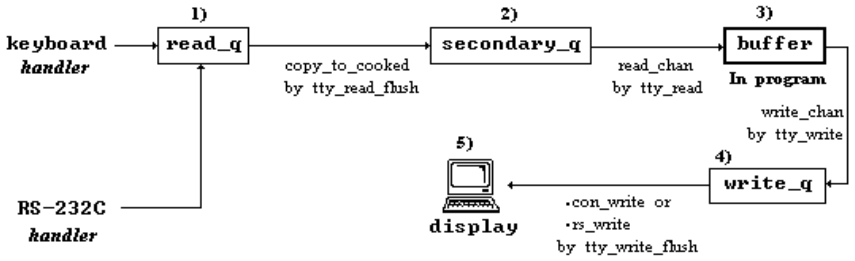
1) major 4번(TTY_MAJOR) device 등록.
2) major 5번(TTYAUX_MAJOR) device 등록.
3) 각각 최대 256개(MAX_TTYS)의 device를 가질수 있다.
4) bottom half routine 등록.

---

123) UNIX NetWork Programming-stevens 에서 pseudo terminal 참고.
124) 리눅스에서는 부팅할때 수행되는 inetd(rc.inet2)에 의해 외부에서 rlogin, telnet접속요구시 기동한다.
125) 본인은 이보다 더 잘 그릴 자신이 없다. I am afraid of Copyright!!

5) line displine 함수 등록
6) keyboard 초기화. (kbd_init)
7) console 초기화. (con_init)
8) serial line 초기화 (rs_init)

12.3 terminal입력 전달과정



<그림12.3> terminal 입력에서 출력까지.

1) console에 의한 입력과 rs-232c terminal에 의한 입력은 각각 keyboard handler(keyboard_inte -rrupt함수)와 serial handler(rs_interrupt함수)에 의해 open된 terminal의 read queue buffer에 들어간다.

tty_open은 getty daemon에 의해 가상console을 open하기 위해 처음 수행된다. terminal이 open되지 않으면 put_queue(keyboard.c)함수에서 read queue에 입력 data가 들어가지 못한다.[126]

```
if(!tty)
    return;
```

read queue buffer는 tty_struct구조체 --> tty_queue구조체 --> buf[TTY_BUF_SIZE]에 존재한다.

다음은 tty_queue의 구조체이다.(tty.h)

126) 부팅될때 getty가 수행되기 전에는 keyboard의 key를 눌러도 화면에 출력되지 않는다.

```
#define TTY_BUF_SIZE 1024  /* Must be a power of 2 */

struct tty_queue {
          unsigned long head;
          unsigned long tail;
          struct wait_queue * proc_list;
          unsigned char buf[TTY_BUF_SIZE];
         };
```

2) 각 handler는 bottom half routine이 있고 이들은 handler수행시 동작되며, 이들에 의해 TTY_READ_FLUSH매크로가 수행되고, 연이어서 copy_to_cooked에 의해 terminal의 secondary qu-eue에 read queue의 data가 전달된다. 이때 read queue는 flush(clear)된다.
3) process에 의해 read(fd,buf,.....) system call이 수행되면, file operation함수인 tty_read가 수행되고, line discipline함수인 read_chan에 의해 secondary queue의 내용이 read함수의 *buf*에 저장된다. 이때 secondary queue의 내용은 flush된다.
다음은 tty.c에서 tty_init에 의해 등록되는 line discipline 함수들의 구조체이다. read system call수행을 위해서는 terminal device가 open되어 있어야한다. terminal을 open하는 대표적인 process로는 getty daemon이 있다. getty는 /etc/inittab내에서 defalut로 몇개의 가상 console을 open한다.

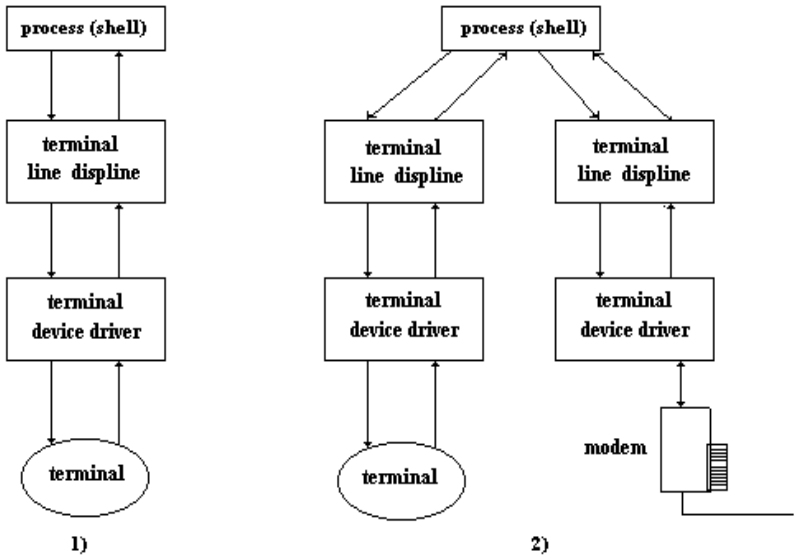
```
static struct tty_ldisc tty_ldisc_N_TTY = {
        0,                      /* flags */
        NULL,                   /* open */
        NULL,                   /* close */
        read_chan,              /* read */
        write_chan,             /* write */
        NULL,                   /* ioctl */
        normal_select,          /* select */
        copy_to_cooked          /* handler */
};
```

등록은

```
(void) tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY);
```

에 의해 이루어진다.

이로써 line discipline module이 놓이는 위치는 그림<12.4>과 같이 process와 device termi -nal사이임을 알수 있다.



<그림12.4> line discipline 위치

- 1)번 그림은 console과 rs-232c접속의 경우를 나타낸다.
- 2)번 그림은 modem을 통한 다른 host로의 연결을 나타낸다.

device driver는 device마다 다르지만, line discipline은 보다 high level에 있으면서, 일정한 함수에 의해(copy_to_cooked) 특정 device driver와의 연결을 결정한다.

4) user process에 의해 write(fd,buf.....) system call를 수행함으로써 *buf*의 내용이 tty_write ⇒ write_chan에 의해 write queue에 data가 전달된다.
5) write queue에 data 저장후, 연이어 TTY_WRITE_FLUSH가 수행되며, write queue의 내용이 con_write, rs_write, pty_write등의 함수에 의해 terminal의 출력장치에 전달된다. 이때 write queue의 내용은 flush된다.

12.4 terminal device driver

terminal device의 각 함수들에 대해 살펴보자.

```
static struct file_operations tty_fops = {
        tty_lseek,
        tty_read,
        tty_write,
        NULL,           /* tty_readdir */
        tty_select,
        tty_ioctl,
        NULL,           /* tty_mmap */
        tty_open,
        tty_release
};
```

이 구조체는 /drivers/char/tty_io.c 에 명시되어 있으며, terminal을 제어하는 함수들로 이루어져 있다. 다른 device driver들과 마찬가지로 먼저 tty_open, tty_read, tty_write함수들을 사용하여 /dev/tty, /dev/pty, /dev/cua등을 대상으로 terminal을 제어한다.

12.4.1 tty_open 함수

1) device file을 open할때 flag로서 O_NOCTTY를 set했는지를 check한다.
   O_NOCTTY는 open을 할려는 process에 대해 device를 controlling terminal로 할당하지 말라는 것이다. controlling terminal이란 process가 foreground이며, device가 현재 terminal을 제어함을 의미한다.

2) /dev/tty와 /dev/console

① major number가 5, minor numbor가 0 이면 /dev/tty를 가르킨다.
   다른 tty의 이름을 가진 device file들은 모두 major numbor가 4이나 이 file만 5이다. 이러한 경우 major를 4로, minor는 현재 process가 가지는 controlling terminal이 된다. /dev/tty는 프로그램의 표준입출력을 redirect하는 것에 상관없이 terminal과의 입출력을 원할때 사용될수 있다.[127]

---

127) Advanced Programming in the UNIX Environment-stevens 247 page

② major numbor가 4, minor number가 0이면 /dev/console을 가르킨다.
/dev/console은 가상 console의 현재 controlling terminal로서, 자동으로 O_NOCTTY flag가 set된다.

③ /dev/tty를 open한 함수가 controlling terminal을 가지고 있지 않으면, open은 실패한다.

```
return -ENXIO;
```

④ device가 PTY_MASTER인 경우 noctty=1.

⑤ init_dev

- tty_table 초기화

```
/* line disciplines */  (termios.h)
#define N_TTY        0
#define N_SLIP       1
#define N_MOUSE      2
#define N_PPP        3
```

특정 terminal에 대한 open함수를 결정한다.

· console	: con_open
· serial	: rs_open
· 가상 terminal	: pty_open

- tty_termios 초기화
  - · IXON :
    STOP문자와 START문자를 제어문자로서 terminal에 출력할때 STOP문자를 만나면 출력을 중지하고, START문자를 만나면 다시 출력이 시작된다. 이 bit가 set되지 않으면, 이 두 문자는 보통문자로 처리(processing)한다.
  - · OPOST :
    terminal에 적당히 display되어지도록 write queue의 내용을 처리(processing)한다. 예를 들어 '/n'은 carrage return과 line feed로 이루어져 출력된다.(opost함수) 이 bi-t가 set되지 않으면 write queue의 내용이 그대로 출력된다.

- termios_locked 초기화
  단지, vt_ioctl함수에 의해 locked배열이 사용된다.

⑥ **결정된 open함수 수행**
수행 실패시 재시도(signal이 없으면).

XENIX V 매뉴얼 User's reference -영진출판사 913page

⑦ O_NOCTTY flag가 set되어있지 않고 session leader의 경우이면, controlling terminal을 할당한다.
대표적인 session leader로는 login shell이 있다.

12.4.2 con_open 함수 (console.c)

```
tty->write = con_write
tty->ioctl = vt_ioctl
```

12.4.3 copy_to_cooked 함수 (tty_io.c)

copy_to_cooked는 특정 terminal device가 open상태이어야 이 함수가 수행가능하다.
다음은 TTY_READ_FLUSH 매크로에 의해 수행되는 함수이다.

```
void tty_read_flush(struct tty_struct * tty)
{
        if (!tty || EMPTY(&tty->read_q))
                return;
        if (set_bit(TTY_READ_BUSY, &tty->flags))
                return;
        ldiscs[tty->disc].handler(tty); /*  copy_to_cooked   */
        if (!clear_bit(TTY_READ_BUSY, &tty->flags))
                printk("tty_read_flush: bit already cleared\n");
}
```

셋째줄에서 tty가 open상태가 아니거나, read queue가 비어 있으면 return한다. copy_to_cooked는 **line displine handler**이다.

1) secondary queue에 빈공간이 있는지 check.
serial terminal일 경우(rs_throttle함수 수행), 남아있는 양이 SQ_THRESHOLD_LW(16)보다 작으면 terminal에서 입력받기를 지연시킨다.(struct termios에서 input mode flag인 IXOFF 참조)
2) 빈공간이 없으면 secondary queue에 data옮기기를 포기한다.

```
if(c ==0)
      break;
```

3) read queue값을 읽어들인다.

4) break및 frame error,parity error가 발생되어 readq_flags가 set되어 있다면 error종류에 따라 secondary queue에 들어가는 값을 정한다.
5) __DISABLED_CHAR == '\0'
ERASE(^H),SUSP(^Z),KILL(^U)와 같은 입력특수문자중에서 원하는 특수문자의 기능을 제거하고자 할때, termios구조체의 c_cc[]에 이 값을 넣기위해 사용한다.
6) read queue에서 읽은 값을 secondary queue에 복사한다.
7) write queue가 비어있지 않으면 TTY_WRITE_FLUSH.
8) terminal에서 읽기를 수행하는 process가 sleep하고 있을때, secondary queue에 값이 들어 있다면 그 process는 깨어나서 읽기를 속행한다.
read_chan, write_chan함수들내의 다음 함수에 의해 wait queue에 읽기, 쓰기수행 process가 queue된다.

```
add_wait_queue(&secondary.proc_list,&wait);
```

9) read queue의 빈 공간이 SQ_THRESHOLD_HW(768)보다 크면 serial line에서 data받기를 속행한다. (request to send)[128]

12.4.4 tty_read 함수

read_chan함수 호출.

12.4.5 read_chan 함수

1) 현재 process가 속해 있는 group이 controlling terminal에 속하지 않았을 경우를 다룬다.
즉, background에 있는 process가 terminal에서 읽기작업을 시도하면 background의 모든 job (process)들에게 SIGTTIN signal을 보내게 된다. 이것의 default결과는 background의 모든 job들을 중지시키는 것이다.

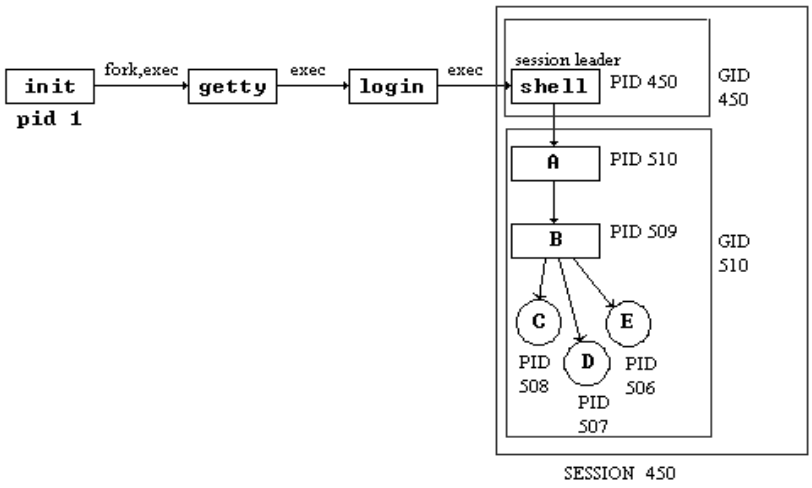
◆ orphaned process group 에 대해.

부모 process가 죽은 자식 process를 orphaned process라 한다. 다른 group에 부모 process를 두고있는 process는 group leader밖에 없다.(그림<12.5>의 process A) 이 group leader가 종료되었을 경우, 그 group은 **orphaned process group**이 된다. group leader의 부모 process가 종료되었을 경우도 마찬가지이다.
일반적으로 이러한 group은 session leader(shell)가 종료(logout)되면서 발생하는데, 이때 bac-kground에 있던 child process들은 orphaned process group에 속하게 된다.[129]

---

128) terminal에게 data를 보내라는 신호.

그림<12.5>는 session과 group의 일반적인 예다. shell에서 fork,exec로 새로운 그룹 GID 510에 process A를 낳는다. 그리고 process A는 process B를, B는 C, D ,E를 차례로 낳아 그룹 510을 이룬다.



<그림12.5> 부모 process와 자식 process의 관계

다음은 pgrp가 orphaned process group인지를 판단하는 함수이다.

```
int is_orphaned_pgrp(int pgrp)
{
        struct task_struct *p;

        for_each_task(p) {
                if ((p->pgrp != pgrp) ||            ①
                    (p->state == TASK_ZOMBIE) ||    ②
                    (p->p_pptr->pid == 1))          ③
                        continue;
                if ((p->p_pptr->pgrp != pgrp) &&        ④
                    (p->p_pptr->session == p->session))  ⑤
                        return 0;               /* Not orphaned */
        }
```

129) GNU C Library Referance Manual의 terminal관련부분 참조.

```
        return(1);      /* orphaned group,  (sighing) "Often!" */
}
```

조건 ①은 다른 group에 속한 process는 대상이 아니기 때문에, 그리고 조건 ②,③은 현재 gr-oup에 속하지만 부모 process가 init daemon인 경우이므로 판단조건에서 제외된다. 조건 ④, ⑤는 POSIX.1에서 지정한 orphaned process group의 정의라고 볼수 있다.[130]

즉, orphaned process group가 되지 않을려면 부모 process가 같은 session에 있으면서 다른 group에 속하는 경우에 해당하는 자식 process가 group내에 한개라도 있어야 한다.[131]

그림<12.5>를 보자. GID 510 group은 현재 orphand process group이 아니다. 그러나, shell이나 process A가 종료되면 위의 고리는 깨어진다. 그때, group ID 510의 process들중에서 부모 process가 다른 그룹에 속하는 것은 존재하지 않는다. 부모 process를 잃은 process의 새로운 부모 process는 init daemon이 된다(조건 3참조). 결과적으로 GID 510 group은 orphand process group이 된다.
session leader가 종료될때 SIGHUP signal과 SIGCONT signal이 child process들에게 전달되는데 SIGHUP signal은 default로 child들을 종료시키지만 child는 handler에 의해 signal을 처리할수 있다. SIGCONT signal은 부모 process가 종료될 당시 stop상태인 child process가 있다면 계속 수행되도록 하는 역활을 한다.
orphand group는 terminal과 대화할수는 없다. 부모 process가 종료될 당시 shell에 의해 forg-round였으므로, 남은 child들은 여전히 background에서만 수행될수 있을뿐이다.[132] 만약 orphand group이 terminal에서 읽을려고 하면 EIO가 error가 return된다.

2) kernel이 user process에게 한번에 얼마만큼의 data를 보내는가를 결정한다.
kernel이 terminal과의 data입출력을 처리하는 방법에 있어 canonical mode(cooked)와 noncan-onical mode(raw)로 나뉜다. 이것은 termios struct의 cflag값인 ICANON에 의해 결정된다.
canonical의 경우는 특수문자를 다룰수 있으며, kernel이 line단위로 data를 읽어들이지만 nonc-anonical은 kernel이 read system call에 의해 한번에 얼마만큼의 양을 읽어야 할지를 따로 결정해 주어야 한다. 한번에 1byte씩을 읽어들일 경우 system call이 여러번 수행되어야 하므로 효율적이지 못하며 system의 속도를 떨어뜨리게 된다. 따라서 최소 몇byte를 읽고난후 read system call이 process로 return한다든지, 일정기간동안 read call내에서 읽기 loop를 지

130) 보다 엄밀히 말하면 orphaned process group이 아니기 위한 조건이다.
131) Advanced programming in the UNIX Environment 256page 참조.
132) Advanced programming in the UNIX Environment 256page 참조.

속하도록 결정하게 된다.[133]
다음은 tty.h의 정의 내용이다.

```
#define TIME_CHAR(tty) ((tty)->termios->c_cc[VTIME])
#define MIN_CHAR(tty) ((tty)->termios->c_cc[VMIN])
```

c_cc[VTIME]의 시간단위는 0.1 초이다.
이 후 코드에서 minimum과 current->timeout이 만족할때까지 loop함을 알수있다.

3) 현재 process(terminal에서 읽기를 수행중임)를 wait queue에 둠으로써 copy_to_cooked함수에 의해(더 정확히 말하면 wake_up함수에 의해) process가 sleep상태이더라도 깨어날수 있다.

```
add_wait_queue(&secondary.proc_list,&wait);
```

4) input_available_p함수에 의해 queue내에 이용가능 data가 있는지 check한다.
current->state = TASK_INTERRUPTIBLE;
이렇게 함으로써 이용가능 data가 없으면 keyboard interrupt를 받아 data이용이 가능할때까지 scheduling 및 looping(continue)을 한다.
이용가능 data가 있다면 다시 TASK_RUNNING상태로 바꾼다.

5) secondary queue에서 data를 읽어낸다.

6) 읽어낸 data를 user buffer에 둔다.

7) 읽기작업이 끝났으므로, wait queue에서 현 process를 제거한다.

12.4.6 tty_write 함수

1) ioctl system call ⇒ vt_ioctl(tty,TIOCCONS,.....)에 의해 출력을 재지정하는지 check.
2) write_chan 호출

12.4.7 write_chan 함수

---

133) Advanced programming in the UNIX Environment 352page 참조.

1) background에 있으면서, terminal에 출력을 할려고 할때 background에 있는 process들은 SIG-TTOU를 받게된다.(check_change)
   ▶ lflag의 TOSTOP :
      background에서 출력하려고 할때, SIGTTOU를 보낸다.
2) wait queue에 현재 process를 queue한다.
   add_wait_queue(&tty->write_q,proc_list,&wait);
3) line이 hangup상태(끊어짐)인지 check.(tty_hung_up_p)
   line이 끊어진 상태에서 write할려고 하면, I/O error.
4) buf의 내용을 write queue에 옮김.
5) TTY_WRITE_FLUSH
   con_write, rs_write, pty_write등의 terminal에 따른 출력함수가 수행된다.
6) wait queue에서 현재 process 제거.

12.5 console driver

12.5.1 con_init 함수

1) setup.S에서 획득한 screen정보를 get.

2) blank timer set.
   화면이 blank 되어지는 시간으로서, 초기치는 600초(second)
   `int  blankinterval = 10*60*HZ  (console.c)`

3) graphic card에 따라 vedio변수값들 지정.

VGA의 경우,

```
can_do_color = 1
video_mem_base = 0xb8000 (map 시작)
video_mem_term = 0xc0000 (map 끝)
  (화면출력을 위해 memory map 된 부분)
video_port_reg = 0x3d4 (screen start)
video_port_val = 0x3d5 (     "      )
video_type = VIDEO_TYPE_EGAC
video_mem_term = 0xc0000
display_desc = "EGA+"
```

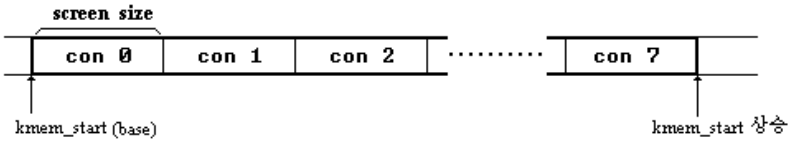
4) 가상 console초기화

여기서 pos, origin, video_mem_start등은 다음과 같이 정의됨을 기억하자.(console.c)

```
#define origin              (vc_cons[currcons].vc_origin)
#define pos                 (vc_cons[currcons].vc_pos)
#define video_mem_start     (vc_cons[currcons].vc_video_mem_start)
#define video_mem_end       (vc_cons[currcons].vc_video_mem_end)
```

즉, 해당 console의 pos, origin, video_mem_start가 된다는 것이다.

① 각 가상 console의 kmem_start에서부터 screen buffer들을 할당한다.



<그림12.6> console screen 위한 메모리 할당.

어떤 가상 console상에서 process가 출력이 있으면 위의 해당 console memeory에 출력된다.

② · console 입력 mode는 KT_TEXT.

key 입력 mode에는 graphic mode와 text mode 두가지가 있다.

cursor를 set하는 등의 kernel내의 동작은 text mode에서만 가능하다.

```
#define         KD_TEXT         0x00
#define         KD_GRAPHICS     0x01
```

- terminal switching mode가 VT_AUTO.

```
#define  VT_AUTO         0x00    /* auto vt switching */
#define  VT_PROCESS      0x01    /* process controls switching */
#define  VT_ACKACQ       0x02    /* acknowledge switch */
```

AUTO mode에서는 Alt-Function key에 의해 무조건 console switching이 이루어진다.
PROCESS mode에서는 바꾸기 전(old) 가상 console의 forground process와 바꿀려는(new) 가상 console의 forground process에 대해 허락을 받아야 한다.
즉, 양쪽 console의 forground proces들이 controlling terminal을 release하겠다는 응답이 필요하다. 그래서 약속된 signal을 forground process들에게 보내면, 그리고 process가 VT_RELDISP ioctl에 의해 응답하게 되고,비로소 완전한 console switching이 이루어 진다.[134]
change_console함수(tty_io.c)를 참고하기 바란다. 그 곳에서 PROCESS mode인 경우, old console의 forground와 new console의 forground process가 각각 check된다.

3) cook mode가 default임을 명시.
Not raw mode (clr_kbd)

```
#define decarm          VC_REPEAT
#define decckm          VC_CKMODE
#define kbdapplic       VC_APPLIC
#define kbdraw          VC_RAW
#define lnm             VC_CRLF
```

4) reset terminal
terminal 속성을 초기화한다.

---

134) I have not enough information about *VT_ACKAQA*
단지, PROCESS MODE가 아닐 경우, process의 RELDISP ioctl과의 대화를 위해 있는 것이라 추정된다. 그 때, process에게 보낸 signal에 대해 무조건 성공적이라는 응답을 할것이며 console switching이 속행될 것이다.(return 0) vt_ioctl함수 참고.

5) 초기 console을 위한 currons와 fg_console이 0(Not 1)[135]임을 기억하자.

6) terminal출력을 위한 memory위치(pos변수값) 지정.
앞에서도 말했듯이 VGA card의 경우, video_mem_base(0xB8000)에서 screen size크기만큼은 screen display를 위한 memory 공간이다. terminal출력함수는 해당 console의 pos위치에 값을 출력한다.
초기 console의 pos는 video_mem_start = video_mem_base이다. 따라서, pos에 값을 출력하면 terminal display화면의 해당위치에 출력이 된다.

7) update_screen
currons = fg_console = 0 이므로 여기서는 그냥 return.
그러나 user가 console switching을 원하면, change_console함수가 수행되는데, 이 때의 과정을 다음 절에서 살펴보자.

8) register_console
console_print라는 함수를 화면출력을 위해 등록한다.
printk함수가 수행되면 write queue에 data가 적재되고, 이들 data는 console_print함수에 의해 terminal에 display된다.
그래서 console_print가 등록되기전인 지금까지는(부팅시작부터) printk가 수행되면, 모두 queue에 적재되어 있다가, 등록된 지금에 와서 적재된 것들이 한꺼번에 출력된다.
이 후에는 정상적으로 출력이 이루어진다.

12.5.2 change_console 함수

1) PROCESS mode인지 확인하고, 그렇다면 현재(old) forground process signal을 보내고 process가 답할때까지 기다린다. 답이 오면 return.(process는 signal을 받으면, complete_change_console을 수행. vt_ioctl참조) 만약에 수행중인 forground process가 없으면, X server가 기동중이 아닌것으로 생각되므로 console mode를 KD_TEXT로 하고, complete_change_console함수가 수행된다.
AUTO mode일 경우 old forground process가 GRAPHIC mode이면 X server가 기동중인것으로 보이므로, console switching이 이루어지지 않는다. TEXT mode이면 complete_change_console을 수행된다.

---

135) 초기 console device file은 /dev/tty1이다.

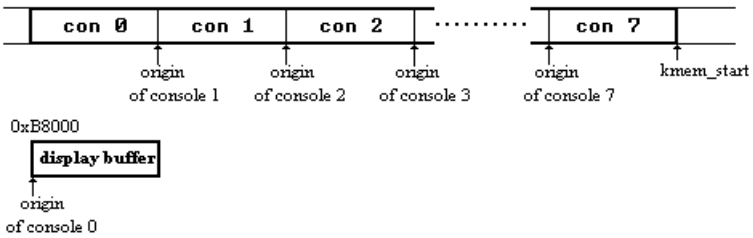
2) update_screen이 수행된다.

complete_chang_console함수에서는 새로운(new) console의 forground process를 check한다. 없으면 앞에서처럼 console mode를 TEXT mode로 한다.

PROCESS mode에서는 GRAPHIC mode에서 TEXT mode로의 console switching이 있을수 있다. 이때는 unblank_screen이 수행된다. 그 반대일 경우는 blank_screen이 수행된다.(TEXT ⇒ GRAPHIC)

3) **update_screen**

처음 부팅되었을 당시에는 0xB8000에 console 0의 origin이 위치해 있다.



<그림12.6> display buffer 초기상태

pos값은 다음 공식에 의해 origin에 의해 결정된다.

```
pos = origin + y*video_size_row + (x<<1)
```

▶ 마지막 항목은 영문 1byte 출력을 위해서는 속성을 포함하여 2바이트 값이 들어가야하므로 1byte출력에 pos는 2byte가 이동한다.

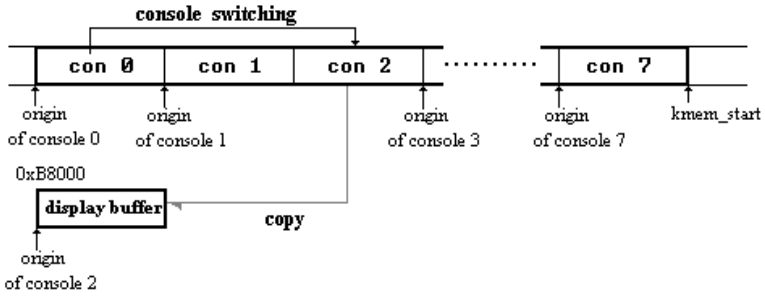
그래서 pos에 값을 써 넣으면 화면에 출력이 된다. 이 후에 console switching이 생기면, 해당 console의 origin은 0xB8000을 가르키고, 나머지들의 origin은 그림<12.7>에서 자신의 screen buffer위치를 가르키고 있다. 그래서 forground가 아닐때는 이곳에 출력을 하다가, 자신이 forground가 되면서 이곳의 내용을 0xB8000에 복사하고, origin도 0xB8000로 옮아간다.

① get_scrmem

0xB8000의 내용을 기존의(old) console screen buf에 복사하여 그가 background로 계속 수행될수 있도록 한다. 이때 그의 origin은 자신의 console screen buffer를 가르키게 된다.
이 console의 origin에 따라 재지정된다.

② set_scrmem

new console의 screen buffer내용을 0xB8000에 복사하고 그의 origin을 0xB8000가 되도록 한다.



<그림12.7> console switching후 display buffer

③ set_origin

이제 0xB8000을 display buffer가 되도록 재지정한다.

관련 port에 대해서는..

```
3d4h index  Ch (r/W):  CRTC: Start Address High Register
bit 0-7  Upper 8 bits of the start address of the display buffer

3d4h index  Dh (r/W):  CRTC: Start Address Low Register
bit 0-7  Lower 8 bits of the start address of the display buffer

3d4h index  Eh (r/W):  CRTC: Cursor Location High Register
bit 0-7  Upper 8 bits of the address of the cursor

3d4h index  Fh (r/W):  CRTC: Cursor Location Low Register
bit 0-7  Lower 8 bits of the address of the cursor
```

④ set_cursor

display buffer에 copy후 다음 위치에 cursor를 갖다 놓는다. 이 위치는 pos에 의해 결정된다.

4) 자신의 console이 기동하기를 기다리고 있는 process를 깨운다.
이러한 process들은 VT_WAITACTIVE ioctl에 의해 수면에 들어간다.

5) 여기서 화면 update는 이루어졌는데, 입출력은 무엇에 의해 바뀌는지(redirect) 궁금할수 있다.
이것은 입출력을 하는 함수인

```
TTY_READ_FLUSH(TTY_TABLE[0])    /* do_keyboard_interrupt 함수 */
TTY_WRITE_FLUSH(TTY_TABLE[0])
```

에서 TTY_TABLE[0]가 fg_console에 해당된다.
그래서 fg_console값의 변화는 자연적으로 입출력을 변화시킨다.

12.5.3 con_write 함수

1) write queue의 내용이 ESnormal(default by reset_terminal in coninit함수)이고, 출력가능 ascc -ii문자이면 현재 console에 출력한다.

```
pos = (attr << 8) + c;
```

```
▶ console.c의 앞부분에 현재 console의 cursor위치 정의
#define  pos  (vc_cons[currcons].vc_pos)
```

2) 그렇지 않은 data들에 대해서는 translation[][136] 배열에 따라 처리된다.

3) *setterm*이라는 유틸리티를 사용하면 terminal을 제어할수 있다. 예를 들어

```
# setterm -inversescreen on
```

은 console화면의 배경색과 글자색이 반전되어 나타나게 한다.

136) 이 key값 변환부분에 대한 정보를 가지고 있지 않다.

이것은 setterm 코드내의 다음 부분에 의해 이루어진다.

```
#define ESC  "\033"

if (opt_inversescreen) {
 if (vcterm)     /* 이 옵션은 console에서만 동작한다.  */
        if (opt_invsc_on)
                printf("\033[?5h");  /* 반전 */
        else
                printf("\033[?5l");  /* 원래대로 복구 */
}
```

"\033"은 ESC문자이다.[137] 단지 한개의 문자열을 print함으로써 화면의 제어가 이루어지는 것이 다.

con_write는 printf함수에 의해 호출된다. 그리고 **set_mode**()함수가 수행된다.

```
 case ESgotpars:
        state = ESnormal;
        switch(c) {   /* 문자열의 마지막 문자  */
                case 'h':
                        set_mode(currcons,1);
                        continue;
                case 'l':
                        set_mode(currcons,0);
                        continue;
                case 'n':
```

12.5.4 library에서의 getchar 함수

user program에서 getchar()이 수행되었을때 libc[138]내에서 동작하는 과정을 살펴보자. getchar()은 libio/stdio/getchar.c에 정의 되어있다.

---

137) DOS ANSI(American National Standards Institute)를 사용해 본 사람은 이 문자열이 낯설지 않을 것이다. 유감스럽게도 본인은 ANSI code에 대한 자료를 많이 가지고 있지 않다.
138) libc version 4.5.21의 경우이다.

```
int  getchar ()
   {  return _IO_getc (stdin);}
```

1) libio.h에서

```
#define _IO_getc(_fp) \
       ((_fp)->_IO_read_ptr >= (_fp)->_IO_read_end \
          && __underflow(_fp) == EOF ? EOF \
         : *(unsigned char*)(_fp)->_IO_read_ptr++)
```

2) 다음은 libio/stdfiles.c의 코드중에서 standard input,output,error 를 정의하는 부분이다.

```
#define FILEBUF_LITERAL(CHAIN, FLAGS, FD) \
       { _IO_MAGIC+_IO_LINKED+_IO_IS_FILEBUF+FLAGS, \
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, CHAIN, &_IO_file_jumps, FD}

#define DEF_STDFILE(NAME, FD, CHAIN, FLAGS) \
  struct _IO_FILE_plus NAME = {FILEBUF_LITERAL(CHAIN, FLAGS, FD), 0}

DEF_STDFILE(_IO_stdin_, 0, 0, _IO_NO_WRITES);
DEF_STDFILE(_IO_stdout_, 1, &_IO_stdin_._file, _IO_NO_READS);
DEF_STDFILE(_IO_stderr_, 2, &_IO_stdout_._file,
            _IO_NO_READS+_IO_UNBUFFERED);
```

3) read_ptr과 write_ptr이 0임을 알수 있다.
underflow(getops.c)는 IO_file_underflow(fileops.c), IO_file_read(fileops.c)를 호출하고 마침내 read system call을 호출한다. 그때 system call parameter인 buf pointer는 바로 _IO_read_ptr을 가르킨다.
그래서, _IO_getc(stdin) 매크로 내의

```
*(unsigned char*)(_fp)->_IO_read_ptr++
```

에 의해 한 문자를 읽어들인다.

4) 다음은 관련 구조체들이다.
대부분의 symbol들이 '_IO_'가 붙어있음을 알수 있다.

```
        #define  stdin  _IO_stdin
```

특히

```
        struct _IO_jump_t _IO_file_jumps
```

구조체를 주목하자.

```
struct _IO_FILE {
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

/* These names are temporary aliases. TODO */
#define _other_gbase _IO_save_base
#define _aux_limit _IO_backup_base
#define _other_egptr _IO_save_end

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  struct _IO_jump_t *_jumps; /* Jump table */

  int _fileno;
  int _blksize;
  _IO_off_t _offset;

#define __HAVE_COLUMN /* temporary */
```

```
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  char _unused;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */
};

struct _IO_FILE_plus {
  _IO_FILE _file;
  __const void *_vtable;
};

extern struct _IO_FILE_plus _IO_stdin_, _IO_stdout_, _IO_stderr_;
#define _IO_stdin (&_IO_stdin_._file)
#define _IO_stdout (&_IO_stdout_._file)
#define _IO_stderr (&_IO_stderr_._file)

struct _IO_jump_t {
    _IO_overflow_t __overflow;
    _IO_underflow_t __underflow;
    _IO_xsputn_t __xsputn;
    _IO_xsgetn_t __xsgetn;

    _IO_read_t __read;
    _IO_write_t __write;
    _IO_doallocate_t __doallocate;
    _IO_pbackfail_t __pbackfail;
    _IO_setbuf_t __setbuf;
    _IO_sync_t __sync;
    _IO_finish_t __finish;
    _IO_close_t __close;
    _IO_stat_t __stat;
    _IO_seek_t __seek;
    _IO_seekoff_t __seekoff;
    _IO_seekpos_t __seekpos;
#if 0
    get_column;
    set_column;
```

```
#endif
};

struct _IO_jump_t _IO_file_jumps = {
  _IO_file_overflow,
  _IO_file_underflow,
  _IO_file_xsputn,
  _IO_default_xsgetn,
  _IO_file_read,
  _IO_file_write,
  _IO_file_doallocate,
  _IO_default_pbackfail,
  _IO_file_setbuf,
  _IO_file_sync,
  _IO_file_finish,
  _IO_file_close,
  _IO_file_stat,
  _IO_file_seek,
  _IO_file_seekoff,
  _IO_default_seekpos,
};
```

12.6 serial line 초기화

부팅중에 RS-232C와 관련된 terminal의 초기화는 rs_init함수에 의해 이루어진다.
직렬통신은 UART라는 chip을 통해 가능한데, UART chip에는 흔히 알려진 8250 외에도, 16540, 16550, 16550A 등이 있다. rs_init은 PC가 제공하는 UART chip을 확인한다. rs_table(serial.c)은 직렬통신을 위한 port와 IRQ를 다음과 같이 제공한다.

```
struct async_struct rs_table[] = {
        /* UART CLK   PORT IRQ     FLAGS        */
        { BASE_BAUD, 0x3F8, 4, STD_COM_FLAGS },         /* ttyS0 */
        { BASE_BAUD, 0x2F8, 3, STD_COM_FLAGS },         /* ttyS1 */
        { BASE_BAUD, 0x3E8, 4, STD_COM_FLAGS },         /* ttyS2 */
        { BASE_BAUD, 0x2E8, 3, STD_COM4_FLAGS },        /* ttyS3 */
```

modem이나 RS-232C card를 통해 data를 외부로 보내거나, 외부에서 오는 data를 받을때 수행되는 interrupt handler(rs_interrupt함수)는 terminal을 개방하는 rs_open함수에 의해 set된다.

port로 출력을 보내는 rs_write는 정해진 memory로 출력을 보내는 con_write와 비교된다.

13장

시그널 처리

13.1 signal 발생

process에게 signal을 보낸다는 것은 매우 간단한 절차에 의해 이루어진다. 다음 코드(kernel/exit.c)를 보자.

```
int send_sig(unsigned long sig,struct task_struct * p,int priv)
{
        ....................................
        /* Actually generate the signal */
        generate(sig,p);
        ....................................
}
```

generate함수는 process에게 signal을 보내는 역활을 하는데, 이것은 단지 task->signal에 해당 singal bit를 set하는 것에 의해 이루어진다.

```
static int generate(unsigned long sig, struct task_struct * p)
{
        unsigned long mask = 1 << (sig-1);
        ..............................
```

```
        p->signal |= mask;
        ..............................
}
```

특정 signal을 block시키는 것은 task->blocked bitmap 에 의해 이루어진다.

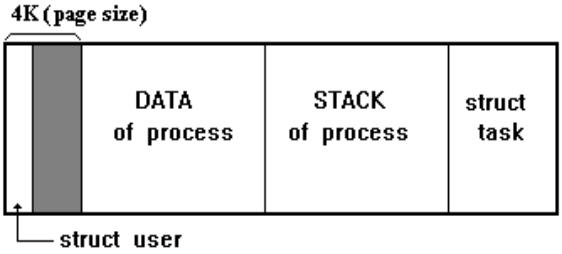
13.2 core. file

SIGIOT, SIGFPE, SIGSEGV signal에 의해 process가 종료될 경우, kernel은 process의 image를 현재 diretory에 *core.*라는 이름의 file로 담아둔다. 이 core file은 대부분의 Unix debugger(예를 들어 gdb등)에 의해 process가 종료 되었을때의 상태를 debugging하는데 사용된다.[139] process의 image를 흔히 *core image* 라고 하는데, 이것은 초창기 memory가 magnetic core memory이었던 것에서 비롯되었다고 한다. [140]

```
case SIGIOT: case SIGFPE: case SIGSEGV:
        if (core_dump(signr,regs)) /* fs/exec.c */
                signr |= 0x80;
        /* fall through */
```

SIGIOT signal은 hardware error에 의한 경우이다.
signr은 parent process에게 전달된다.

core file의 구조는 그림<13.1>과 같다.



<그림 13.1> core. file 구조

139) Advanced Programming in the Unix Environment 265 page
140) OPERATION SYSTEMS DESIGN AND IMPLEMENTATION - ANDREW S. TANENBAUM 1.3.1 Processes

core file의 헤더(header)에 해당하는 user구조체(include/linux/user.h)는 다음과 같다.

```
struct user{
/* We start with the registers, to mimic the way that "memory" is returned
   from the ptrace(3,...) function.  */
  struct pt_regs regs;              /* Where the registers are actually stored */
/* ptrace does not yet supply these.  Someday.... */
  int u_fpvalid;                /* True if math co-processor being used. */
                                /* for this mess. Not yet used. */
  struct user_i387_struct i387;  /* Math Co-processor registers. */
/* The rest of this junk is to help gdb figure out what goes where */
  unsigned long int u_tsize;   /* Text segment size (pages). */
  unsigned long int u_dsize;   /* Data segment size (pages). */
  unsigned long int u_ssize;   /* Stack segment size (pages). */
  unsigned long start_code;    /* Starting virtual address of text. */
  unsigned long start_stack;   /* Starting virtual address of stack area.
                                  This is actually the bottom of the stack,
                                  the top of the stack is always found in the
                                  esp register.  */
  long int signal;             /* Signal that caused the core dump. */
  int reserved;                /* No longer used */
  struct pt_regs * u_ar0;      /* Used by gdb to help find the values for */
                               /* the registers. */
 struct user_i387_struct* u_fpstate; /* Math Co-processor pointer. */
  unsigned long magic;         /* To uniquely identify a core file */
  char u_comm[32];             /* User command that was responsible */
 int u_debugreg[8];
};
```

header 이후에는 종료된 process의 data영역, stack영역, task정보가 차례로 들어간다.

부록 A DLL에 대한 소개

A.1 DLL의 생성

실행파일(binary)이 생성될때 library를 link시키는 방법에는 정적인 것과 동적인 것이 있다. 정적으로 link시킨다는 것은 linker에 의해 library 코드가 binary에 포함되는 것을 의미한다. 동적으로 link할 경우 binary에는 library 코드가 포함되어 있지않지만, binary가 실행될때 library 코드가 user메모리 공간으로 load된다. 이렇게 함으로써 binary크기를 현저하게 줄일수가 있다. 동적으로 link되어지는 library를 공유 library 또는 DLL(Dynamically Linked Library)이라고 부른다. 리눅스에는 다음과 같은 공유 library가 있으며 이들은 리눅스 시스템의 /lib에 존재한다.

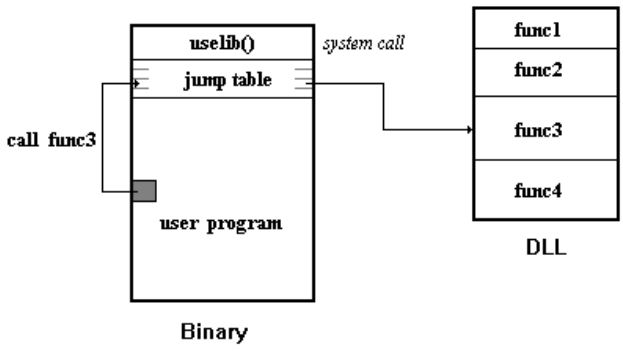
```
Here are the locations of currently registered DLL libraries for Linux.
=======================================================================

libc.so         tsx-11.mit.edu:/pub/linux/packages/GCC/image-4.5.26.tar.gz
libm.so                 included in above tar.gz file.

libX11.so       tsx-11.mit.edu:pub/linux/packages/X11/XFree86-2.1/XF86-2.1-lib.tar.gz
libXt.so                included in above tar.gz file.
libXaw.so               included in above tar.gz file.

librl.so        sunsite.unc.edu:/pub/Linux/libs/librl-1.2.tar.gz
libgr.so        sunsite.unc.edu:/pub/Linux/libs/graphics/libgr-1.3.tar.gz
libf2c.so       sunsite.unc.edu:/pub/Linux/devel/fortran/libf2c-0.9.2.tar.gz
libF77.so               use libf2c.so shown above instead.
libI77.so               use libf2c.so shown above instead.

libXpm.so       sunsite.unc.edu:/pub/Linux/libs/X/libXpm-3.4a.tar.gz

libnsl.so       ftp.lysator.liu.se:/pub/NYS/nys-0.xx.tar.gz (frequent updates)
```

```
libolgx.so      sunsite.unc.edu:/pub/Linux/libs/xview3L5.1.tar.gz
libxview.so             included in above tar.gz file.
libsspkg.so             included in above tar.gz file.
libUIT.so               included in above tar.gz file.

libPEX.so       tsx-11.mit.edu:pub/linux/packages/X11/XFree86-2.1/XF86-2.1-pex.tar.gz

libtcl.so       sunsite.unc.edu:/pub/Linux/devel/tcl/*
libtk.so                various related tcl/tk stuff included in above tar.gz files.

libWc.so                Unknown
libXp.so                Unknown

libIV.so        sunsite.unc.edu:/pub/Linux/X11/devel/iv-3.1l.2.tar.gz
libUnidraw.so           included in above .tgz files.

libXm.so        The Motif library is *not* free. See note below.

libsrgp.so      sunsite.unc.edu:/pub/Linux/X11/devel/suit.tpz
libsuit.so              included in above tpz file.  (reported not shared)

libOI.so        tsx-11.mit.edu:/pub/linux/packages/OI/oi40.tar
libOIrg.so              included in above tar file.

libld.so        tsx-11.mit.edu:/pub/linux/packages/GCC/ld.so-1.4.3.tar.gz
                        (required for libc 4.4.4 and above.)

libarma.so      ftp.atnf.csiro.au:/pub/karma
libkarmaX11.so          see above site
libkarmaXt.so           see above site
libkarmagraphics.so     see above site
libkarmawidgets.so      see above site
libkarmaxview.so        see above site

libwxwin.so     sunsite.unc.edu:/pub/Linux/X11/devel/wxWin_linux.tgz

libandrew.so    sunsite.unc.edu:/pub/Linux/X11/andrew/andrew61.prog.tar.gz
```

```
libUil.so               Commercial library.

libBLT.so       sunsite.unc.edu:/pub/Linux/devel/tcl/blt1.6-bin.tar.gz

libvga.so       sunsite.unc.edu:/pub/Linux/libs/graphics/svgalib111.tgz

libitcl.so      sunsite.unc.edu:/pub/Linux/devel/tcl/itcl1.3-bin.tar.z

------------
Note 1:-

Drop in DLL libraries for Xaw to get a 3d effect (libXaw3d-0.6) and a Mac(TM)
like scroll bar on Xaw clients are available respectively at

        sunsite.unc.edu:/pub/Linux/libs/Xaw3d-0.6B.3.1.1.bin.tar.gz
and
        sunsite.unc.edu:/pub/Linux/libs/libXaw.Scrollbar.taz
```

공유 library명은 *name.so.major.minor*와 같은 형태를 하고 있는데 major.monor는 libray의 버전에 해당하며, so는 shared object를 의미한다.

공유 library와 binary의 관계는 그림<A.1>과 같다.



그림<A.1>

그림에서 보다시피 공유 library를 사용하는 binary는 앞부분에 *uselib* system call을 포함한

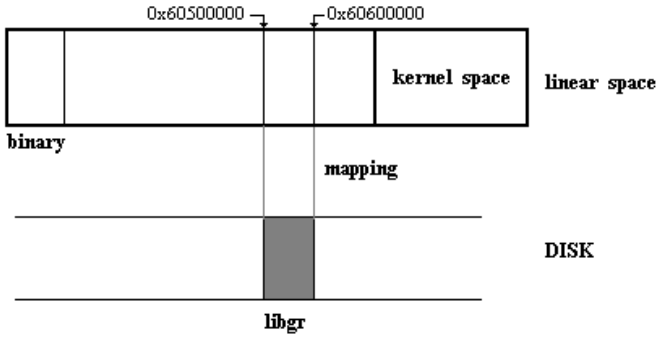
startup 코드와 jump table을 가지고 있다. binary가 수행될때 uselib system call은 DLL을 user 선형공간의 정해진 위치에 mapping시킨다.(그림<A.2> 참고) kernel의 sys_uselib(*library)에서 library parameter에는 library명 pointer가 전달된다.
jump table은 다음과 같은 형태로 되어 있다.

```
func3 :
    jmp  DLL내의 함수 pointer
func 4 :
    jmp .............
```

user program(binary)에 startup 코드와 jump table을 포함시키는 역활은 **mkstubs** 유틸리티에 의해, DLL내의 기계어 코드는 **mkimage** 유틸리티에 의해 이루어진다. 이들 유틸리티들은 DLL을 생성시키는 도구로서 internet을 통해 구할수 있다.

tsx-11.mit.edu:/pub/linux/packages/GCC/src/tools-2.11.tar.gz

이 패키지에 포함된 문서인 README.ps는 실제로 DLL을 생성시키는 예를 보여준다.



그림<A.2>

graphic관련 공유 library인 **libgr**은 0x60500000에서 0x605fffff까지에 mapping이 된다. 이렇든 모든 공유 library는 다른 공유 library와 겹쳐쓰지 않도록 정해진 공간안에 mapping된다. 이것은 초기에 공유 library가 만들어질때 정해지는데, 시작주소와 크기가 mkstubs와 mkimage의 옵션으로 제공된다.[141]

141) 공유 library들이 mapping되는 주소는 DLL 도구패키지내의 REAME.ps에 나와있다.

A.2 ldconfig와 ld.so

DLL과 관련된 명령어 두가지가 ldconfig와 ld.so이다.
ldconfig는 /etc/rc 또는 /etc/rc.d/rcS에 존재하며, 시스템이 부팅될때 /lib 또는 /usr/lib내에 있는 공유 library에 대해 최신 version으로 link시키는 역활을 한다. 즉, /lib directory에 존재하는 C Library에 대한 공유 library 최신 version이 4.5.21이라면 다음과 같은 link가 이루어진다.

```
# ln -sf libc.4.5.21 libc.4
```

ld.so는 /lib 또는 /usr/lib에 상주하며, process의 생성을 위한 마지막 object link작업을 한다. 리눅스에서는 A.OUT format의 경우 컴파일시 -static이라는 option을 주지 않으면, 수행가능한 완전한 binary가 만들어지지 않는다. 따라서 process생성시 공유 library를 load하는 작업이 요구되고, 이것을 ld.so가 하는 것이다. 즉, ld.so는 컴파일할때 완성되지 않은 link작업을, binary가 수행될때 완성시키는 역활을 하는 것이다.

부록 B 커널함수 요약

- void add_timer(long jiffies, void (*fn)(void)) <kernel/sched.c>
  jiffies(1/100초)만큼의 시간이 흐른뒤에 kernel내의 fn함수가 수행된다.

- struct buffer_head * bread(dev_t dev, int block, int size) <fs/buffer.c>
  dev인 disk의 block에서 size만큼 읽은 값을 담은 buffer를 되돌린다. size는 일반적으로 block size인 1024 byte이다.
  return값은 읽은 값을 담은 buffer pointer이다.

- unsigned long bread_page(unsigned long address, dev_t dev, int b[], int size, int prot) <fs/buffer.c>
  b[0]~b[3]의 값인 disk block(모두 1 page에 해당)들을 읽어 원하는 address에 둔다. 이 함수는 buffer와 code page를 공유할수 있도록 하기 위해 try_to_share_buffers함수(fs/buffer.c)를 사용한다.
  try_to_share_buffers는 prot가 read only인 경우만 가능하며, check_aligned함수에 의해 disk block의 buffer들이 일렬로 전부 붙어 있으며 동시에 한개의 page로 이루어져 있다면, buffer의 내용을 address에 따로 copy하지않고, 이 buffer를 그대로 사용함으로써 page를 절약한다.
  return값은 try_to_share_buffers가 성공했을 경우는 첫번째 buffer pointer. 실패했을 경우는 address.

- void brelse(struct buffer_head * buf) <fs/buffer.c>
  할당되어 있는 buffer인 buf를 해제시킨다.

- static struct buffer_head * create_buffers(unsigned long page, unsigned long size) <fs/buffer.c>
  get_free_page함수에 의해 획득한 page를 size크기의 buffer로 분할한다.
  return값은 마지막에 만들어진 buffer pointer를 가리킨다.

· static int dir_namei(const char * pathname, int * namelen, const char **
name, struct inode * base, struct inode ** res_inode) <fs/namei.c>
pathname에서 directory부분까지만(file은 제외한)의 inode pointer를 res_inode에 되돌린다. 그리고 file이름 pointer를 name에 되돌리며, file이름의 길이는 namelen에 되돌린다. base는 pathname을 찾는 directory로 사용된다. 그러나 pathname이 root directory를 포함하고 있다면, 즉 "/usr/src/linux1_0" 와 같은 식이라면 base는 함수내에서 root directory로 바뀌기 때문에 의미가 없다.
return값이 0이면 성공

· int follow_link(struct inode * dir, struct inode * inode, int flag, int mode, struct **
res_inode) <fs/namei.c>
inode가 다른 file에 symbolic link되어 있는지(target file인지) 확인하고, 그렇다면 source file의 inode pointer를 res_inode에 되돌린다.
dir은 source file을 찾는 directory로 사용된다. flag와 mode는 찾은 source file을 개방하기 위해 사용된다. 그러나 source file명이 root directory를 포함하고 있다면, 즉"/usr/src/linux1_0" 와 같은 식이라면 dir은 함수내에서 root directory로 바뀌기 때문에 의미가 없다.
return값이 0이면 성공
함수내에서 ext2_follow_link함수(fs/ext2/symlink.c)가 호출된다.

· struct buffer_head * getblk(dev_t dev, int block, int size) <fs/buffer.c>
dev인 disk의 block에 할당되어 있는 buffer를 구한다. 먼저 cache에 있는지 확인후, 없으면 새로운 buffer를 할당하고 disk block에 할당시킨다.
return값은 할당된 buffer pointer

· int getname(const char * filename, char **result) <fs/namei.c>
user space에 있는 filename이라는 문자열을 받아 kernel space로 옮긴후, 옮겨진 문자열의 pointer를 result로 되돌린다.
return값이 0 이면 성공

· inline unsigned char get_fs_byte(const char * addr)
inline unsigned short get_fs_word(const unsigned short *addr)
inline unsigned long get_fs_long(const unsigned long *addr)

<include/asm/segment.h>
user data공간으로부터 읽어들이기 위해 사용된다.
addr은 선형번지이다.

· struct buffer_head * get_hash_table(dev_t dev, int block, int size) <fs/buffer.c>
cache(hash table)에서 dev인 disk의 block에 할당된 size 크기의 buffer block이 있으면 되돌린다.
return값은 cache에 있는 buffer pointer. pointer 값이 NULL 이면 cache에 없다는 의미.

· void * kmalloc(unsigned int len, int priority) <mm/kmalloc.c>
void * vmalloc(unsigned long size) <mm/vmalloc.c>
kmalloc은 물리메모리 공간을 할당하기 위한것이며, vmalloc은 kernel선형번지공간을 할당하기 위한 것이다.
자세한 것은 7장 메모리관리 를 참조할것.
return값이 0이면 실패, 성공이면 할당시작주소 return.

· int lnamei(const char * pathname, inode ** res_inode) <fs/namei.c>
namei처럼 pathname의 inode를 구해 res_inode에 되돌린다. 그러나, symbloc link확인은 하지 않는다. 찾은 inode를 link확인없이 그냥 되돌린다.
return값이 0이면 성공.

· int lookup(struct inode * dir,const char * name, int len, struct inode ** result)
<fs/namei.c>
dir이라는 directory에서 name인 file또는 sub-directory를 찾는다. 찾은 것의 inode pointer를 result에 되돌린다. 그러기 위해 disk에서 directory block을 읽어 들인다.
len은 name의 길이이다.
return값이 0이면 성공.
함수내에서 ext2_lookup함수(fs/ext2/namei.c)가 호출된다.

· int namei(const char * pathname, inode ** res_inode) <fs/namei.c>
pathname의 inode를 구해 res_inode에 되돌린다. pathname이 root directory를 포함하지 않으면, 즉 "/usr/src/linux1_0" 와 같은 식이 아니면, 현재 directory에서 file을 찾는다.(current->pwd)
return값이 0이면 성공.

· init open_namei(const char * pathname, int flag, int mode, struct inode ** res_inode, inode * base) <fs/namei.c>

open system call의 대부분을 차지하고 있다. 하지만 user가 사용하는 open함수와는 flag에서 다소 차이가 있다. do_open함수(fs/open.c)에 붙은 주석을 보자. open함수의 flag는 *fcntl.h*에 나타나 있다.

```
/*
 * Note that while the flag value (low two bits) for sys_open means:
 *   00 - read-only
 *   01 - write-only
 *   10 - read-write
 *   11 - special
 * it is changed into
 *   00 - no permissions needed
 *   01 - read-permission
 *   10 - write-permission
 *   11 - read-write
 * for the internal routines (ie open_namei()/follow_link() etc). 00 is
 * used by symlinks.
 */
```

· int permission(struct inode * inode, int mask) <fs/namei.c>

어떤 file에 대해 읽기/쓰기/수행이 허가되는지를 확인한다.

다음은 mask로 들어가는 값들(fs.h)이며, OR조합이 가능하다.

```
#define MAY_EXEC  1  : 수행 허가
#define MAY_WRITE 2  : 쓰기 허가
#define MAY_READ  4  : 읽기 허가
```

return값이 1 이면 허가.

함수내에서 ext2_permission함수(fs/ext2/acl.c)가 호출된다.

· int printk(const char * fmt, ...) <kernel/printk.c>

kernel에서 문자출력함수. user mode에서의 printf함수에 대응한다.

return값은 print된 문자 수.

· void putname(char * name) <fs/namei.c>
name을 위해 할당된 memory를 해제.

· asmlinkage void schedule(void) <kernel/sched.c>
task전환이 있어야 하는지 확인후, task전환이 이루어지는 곳이다. 주로 need_resched변수가 set되면, system call에 의해 수행된다.

· void show_buffers(void) <fs/buffer.c>
buffer의 상태 display.

· int shrink_buffers(unsigned int priority) <fs/buffer.c>
memory가 고갈되었을때, buffer수를 줄여 page를 확보한다. priority는 0,1,2,3중 하나가 될 수 있으며, 수가 작을수록 강력한 요구가 이루어진다.
return값이 1이면 성공. 실패면 0.

· static int sync_buffers(dev_t dev, int wait) <fs/buffer.c>
dev인 disk에 해당하는 buffer에 대해 sync작업을 한다. dev값이 0이면 모든 disk의 buffer를 대상으로 한다.
wait값이 0인 경우는 sync할려는 buffer가 lock상태이면 바로 return한다.
그러나 1 이면 lock이 풀릴때까지 기다린후 sync작업을 완성시킨다.

· static inline unsigned long try_to_share_buffers(unsigned long address, dev_t dev, int *b, int size) <fs/buffer.c>
bread_page(fs/buffer.c) 를 참고하라.

· void * vmalloc(unsigned long size) <mm/vmalloc.c>
본장의 kmalloc 참조.

· void wait_on_buffer(strut buffer_head * bh) <fs/buffer.c>
bh인 buffer에 대해 lock이 걸려있지 않은지, 걸려있으면 풀릴때까지 기다릴 것인지를 결정한다. 기다리는것으로 결정되면 scheduling을 한다.

찾 아 보 기

(r)

(s)

(t)

(u)

(v)