최상위 디렉터리의 lib , /usr/lib, /usr/local/lib
기본 라이브러리, 응용프로그램이 사용하는 라이브러리

최상위 디렉터리의 dev : 장치 파일이 존재
etc : 환경 파일이 존재 (윈도우의 레지스트리)
home : 개별 사용자의 디렉터리 (로그인 했을 때 디폴트로 작업 디렉터리)
var : 시스템에 대한 파일 (/var/log : 시스템 로그 파일)
var/log/syslog : 시스템 부팅, 커널의 이벤트 발생시 정보를 로그파일에 남김
usr : 리눅스 응용프로그램 설치 파일(윈도우에서의 program files)
usr/include : 리눅스 응용프로그램의 라이브러리

ll : 파일 목록 자세히 보기 (ls와는 다르게 세부적인 내용 확인가능)

echo -e 'hello world₩n₩n₩n' (옵션을 줘야지 적용됨)

cd mkdir; [ -f 'Gone with the Wind.mp3' ] : mkdir로 이동 후 현재 디렉터리에 Gone with the Wind.mp3 파일이 있냐 확인 (대괄호가 명령어임)

wc -w + 파일 : 몇 개의 단어로 구성되어 있는지 (word)
wc -l + 파일 : 라인 수 확인
wc -L + 파일 : 해당 파일의 가장 긴 단어의 글자 수
wc -c : 해당 파일에 몇 개의 글자가 존재하는지

seq 1 200 > num200 : 일련 번호 ( 1 ~ 200)을 만들어서 해당 출력을 num200 파일에 저장함
tail -n 10 num200 : num200 파일의 끝에서 10번째 까지 출력
tail -n +101 num200 | head -n 10 : 101라인부터 10줄을 출력
tail -f /var/log/syslog : 시스템 로그의 변동사항을 실시간으로 출력
(만약에 USB 장치를 넣으면 해당 로그기록이 출력됨) – 시스템 관리자가 주로 사용함(특정 파일 추적)

**alias** mkdir='mkdir -p' : mkdir을 mkdir -p로 재정의
ex) mkdir -p dir123/dir1/dir2/dir3 를 해야되는데, 다음과 같이 재정의를 하면 mkdir dir123/dir1/dir2/dir3만 해도 성공적으로 된다. tree dir123로 확인 가능하다.

또한, alias m='minicom -w -D/dev/ttys0'를 단순히 m을 입력함으로써 실행이 가능한데, ttys1에 대해서 실행시키고 싶은 경우에는, 커멘드 라인에 function mm() { minicom -w D/dev/ttys${1};} 을 입력하면 된다. 그 후에, mm 0을 입력하면 원하는 대로 작동이 가능하다.
(함수를 이용하면 융통성이 있음)

pushd . : 현재 경로를 스택에 저장한다.
그 후에 임의의 경로 예로 들어 cd /etc/samba로 이동하여 popd 명령어를 수행하면 다시 직전 경로로 돌아간다.
(pushd . 는 중첩이 가능하고 popd를 사용하면 직전경로로 다시 간다)
cd – 를 하게 되면 해당 경로를 왔다 갔다 하게 된다.

name=Michael; printf -v legend "%s Jackson" $name; echo $ legend
( -v 는 variable을 상징)
해설) 쉘 스크립트에서는 변수의 선언이 가능한데 name=Michael이 예다.
printf에서도 변수 선언이 가능한데 legend 라는 이름으로 변수를 만들었다. 만들어진 변수는 사용할 때 $를 이용해야 한다.

read num 을 입력하면 입력을 받는다. 입력을 하면, num이라는 변수에 저장이 된다.
echo $num으로 변수의 내용을 확인 가능하다.

read -p "what is your phone number" 변수 -> 입력하면 변수에 저장된다.
read -n 1 -p "Are you over 16?" 변수 -> -n 1 : 사용자가 입력하는 글자수 제한(1글자로)
-s : 사용자의 입력을 보이게 하지 않음
read -s -n 1 -t 3 -p "Are you over 16?" 변수 -> 사용자의 입력을 3초동안 제한함.

환경 변수 추가
PATH=$PATH:경로
->환경변수에 경로를 지정하면 해당 경로+파일명을 지정해 주지않아도 파일명으로만 실행이 됨
./파일명 : 현재 경로에 있는 파일을 실행

쉘 : 운영체제 중심부(커널)과 사용자를 연결시켜준다. 사용자와 상호작용하여, 필요한 커널 서비스를 받을 수 있도록 한다. 모든 사용자의 명령은 bash 쉘에 의하여 해석되어 지고 실행된다.

스크립트 작성법
스크립트 파일 : 뒤에 확장자 .sh

ps 를 해보면 bash 가 백그라운드에서 돌아가고 있음을 알 수 있다
->우리가 입력하는 모든 명령을 항상 bash가 실행한다.

스크립트 파일 실행방법
1) ./helloworld.sh
2) bash helloworld.sh
3) source helloworld.sh
4) . helloworld.sh

~ : 사용자 홈 디렉터리

color=white
echo "tiger's color is $color" -> tiger's color is white

animal=Tiger; color=Red;
echo "$animals $colors"  ->실패 (변수가 뭔지 구분이 안됨)
대신에, echo "${animal}s ${color}s  처럼 중괄호를 사용해야함

스크립트 파일 작성
#!/bin/bash
DIRECTORY=`dirname $0`
echo $DIRECTORY

➔ ` ` : back tick(백틱) : `` 안에 있는 것이 먼저 실행되고 실행결과가 DIRECTORY에 EODLQEHLA
➔ **$0** : 현재 작성하고 있는 스크 립트 파일의 경로명

저장하고 chmod +x 로 실행속성을 부여한다. -> 실행하면 . 이 나옴

```
#1/bin/bash
name=$1
email=$2
all=$*


echo "your name is $name"
echo "your email is $email"
```

**여기서 $1은 첫번째 매개변수, $2는 두번째 매개변수, $*는 전달된 모든 매개변수임**
따라서, 위 파일을 실행시킬 때 ./whois.sh james james@gmail.com 처럼 전달인자를 줘야함

해석) whois.sh : $0 (스크립트의 이름 또는 경로)
james : $1
james@gmail.com : $2

$# : 매개 변수의 총 개수

```
<declare>
declare -a alnum=(a1 b1 c1 d1 e1 f1)
echo ${alnum[1]} -> b1 나옴

declare -i inum=78
inum=inum+1
echo $inum ->79나옴
```

반면, num=78 이라고 선언하면 쉘에서는 7이라는 문자와 8이라는 문자로 인식한다.
이것을 정상적으로 동작하게 하려면 declare -i 옵션으로 정수형 변수 선언 방식을 사용해야 한다.

```
declare -r rPi=3.14              // 읽기 전용 변수 사용하기 (다른값으로 재초기화 불가능)
```

declare -x xpath="${HOME}/Desktop/mydir" 와
export XPATH="${HOME}/Desktop/mydir" 은 같은 방법이다.

testString="That that is is that that is not is not"
echo ${#testString}  // 39
echo ${testString:0}  // That that is is that that is not is not (원하는 위치부터 문자열 출력)
echo ${testString:1}  // hat that is is that that is not is not
echo ${testString:3:3} // t t  (3번째 글자부터 3글자만 출력 – 공백 문자 포함)

cd / : 루트 디렉터리로 간다
echo ??? : 디렉터리 이름이 3글자로 된 것만 출력한다.
echo b?? : 디렉터리 이름이 b로 시작하는 3글자로 된 것만 출력한다.

tr abcdefghijklmnopqrstuvwxyz ZABCDEFGHIJKLMNOPQRSTUVWXY <<< "Hello World"
- ➔ HDKKN WNQKC
- ➔ tr 명령어로 Hello World의 글자를 재정의 함 (번역)

tr [:lower:] [:upper:] <<< "Hello World"  // 소문자를 대문자로
tr [:space:] '₩t' <<< "Hello World"  // 공백문자를 tab 문자로 변환

echo *.c :  .c가 들어간 파일만 출력
shopt -s extglob : 확장 globe 사용 명령어  후,
echo !(*.c) : .c가 들어가지 않는 파일 목록만 출력
echo @(*jpg|*bmp) : jpg,bmp가 들어간 파일 모두 출력  // @ : OR

tom="Tom hanks"
deniro="Robert deniro"
[ $tom > $deniro ] 라고 하면 문자열 크기 비교가 아니라 redirect로 알아 들음, 따라서
[[ $tom > $deniro ]] 라고 해야함. (문자열 크기 비교할 때)
echo $? 로 종료상태 출력가능
(마지막으로 종료된 명령어의 종료상태를 알려줌, **정상종료 : 0, 비정상종료 : 0 이외의 값**)

[ $tom = $deniro ] 와 같이하게 되면 Tom hanks와 같이 중간에 공백문자가 있어서 오류가 난다. 다라서 [[ $tom = $deniro ]] 와 같이 이중 대괄호를 사용하던가 아니면 [ "$tom" = "$deniro" ] 와 같이 "" 를 사용해야 한다.

```
if [ ! -f "hello.txt.bak" ]; then
        cp "hello.txt" "hello.txt.bak"
fi
// 만약 hello.txt.bak 이라는 파일이 없다면 hello.txt 파일을 hello.txt.bak 으로 복사한다.
// -f : 파일의 존재 여부를 확인할 때 사용된다. (메타 문자)
// -d : 파일이 디렉터리 인 경우,  -h : 파일이 심볼릭 링크인 경우 등등 많다.

if [[ $( ls -A) ]]; then
        echo "there are files"
else
        echo "no files found"
fi
```

참고)) echo $$ : 처음에 명령어를 호출한 쉘의 PID 값을 출력한다.
eval : 인자를 받아서 현재의 쉘에서 실행한다. 변수들이 명령어가 실행되기 전에 확장되어 변수에 담겨있는 명령어를 실행하는 것이 가능해 진다.

```
COUNT=10
eval echo {0..$COUNT}
// 0 1 2 3 4 5 6 7 8 9 10  가 출력됨
```

**$@ , $*** 은 큰 따옴표 내에서만 사용하지 않으면, 모든 위치 매개변수 라는 의미로 동일하다.

- ➔ "$*" : 매개변수들이 단일 문자로 취급된다
- ➔ "$@" : 매개변수들이 서로 분리된 문자로 취급된다.

```
mystr="Hello"
for((i=0; i<${#mystr}; i++)); do        // 매개 변수의 개수
        c="${mystr:$i:1}"       // ${변수:위치:길이} : 위치 다음부터 지정한 길이만큼의 문자열 추출
        echo "$c"
done
//              H
                e
                l
                l
                o       나옴
```

<date>

date +"%Y₩%m₩%d"  라고 하면 현재 년, 월, 일 나옴

<현재 경로에 있는 파일 삭제>

```
for file in *.c
        do rm "$file"                          //for 루프 , 인용 부호 없음
done                                           // 파일 삭제 시 인용 부호 꼭 사용해야 함
```

seq 0 **2** 10 : 0 ~ 10 까지 2만큼 증가시키면서 화면에 출력
seq 10 -1 0 : 10 ~ 0 까지 1 만큼 감소시키면서 화면에 출력

<case 문>

```
read -p "Enter any string: "
case $REPLY in
        +([[:digit:]]) ) echo "digits" ;;
        *) echo "not digits" ;;                // 그외(default)
esac
```

<case.sh>

```
#!/bin/bash
read -s -n 1 -p "You really want to exit?" response
        case "$response" in
        Y|y) echo YES;;
        N|n) echo NO;;
        *)kill -SIGKILL $$;;
esac
```

```
movies=("Avengers" "Matrix" "Titanic", "None")
        PS3="Please select your favorite movie: "
        select movie in ${movies[@]}
do
        case $movie in
        "None") echo "My favorite movie is not on the list.quit"; break;;
        *) echo "$movie selected";;
esac
```

done

**PS3**를 사용하고 위와 같은 셸 스크립트를 실행하게 되면
1)Avengers
2)Matrix
3)Titanic
4)None
를 먼저 보여주게 된다.

<배열 사용법>

```
declare -a array1=("water" "blue" "super")
declare -a array2=("melon" "mountain" "stars")
for i in "${!array1[@]}" ; do
        printf "%s₩t%s₩t%s₩n" "$i" "${array1[$i]}" "${array2[$i]}"
done
```

<항상 정확하게 파일 검색 후 삭제하는 명령어>

```
find . -iname "*.c" -print0 | xargs -0 rm -rf
```

검색하려는 모든 파일의 마지막에 NULL 문자를 삽입(파일의 시작과 끝 분별 가능)

<find 명령어>

```
find ./ -name "*.c" -exec ls -l {} ₩;
//exec 사용 했을 때 , 이 명령어의 끝을 알 수 있게 ₩; 를 사용해야 함
// 현재 경로에서 부터 확장자 .c 인 파일에 대한 자세한 정보

find ./ ₩( -user hwajun15 -a -perm 644 ₩) -print | xargs ls -l
// 소유자가 hwajun15 이면서 소유권이 644 인 것 . ( -a : AND )
```

png 파일을 jpg로 변환하는 쉘 스크립트

```
#!/bin/bash
for name in *.$1
        do
        mv $name ${name%$1}$2     // 첫번째 인자와 일치하는 부분을 떼어내고, 두번째 인자 붙임
done
```

<리다이랙션> : 표준 출력 내용을 파일로 저장하는 것과 같은 방법

나머지는 다 아는 내용이고, 만약에 dir3333가 없는 파일이라면
ls dir3333 하게 되면 오류메세지가 출력된다. 이를,
**ls dir3333 2> err.log** 하면 (overwrite or 신규 생성되고)
**ls dir3333 2>>err.log** 하면 기존 파일에 추가되는 효과 (append 한다)

두 개의 파일도 병합 가능

```
echo ABCD > file1
echo 1234 > file2
cat file1 file2 > file
//      ABCD
        1234
```

<파이프> - fifo
mknod /tmp/mypipe p 형태로 만듬

```
echo begin; (for i in {100..110}; do echo $i >> num100; sleep1; done)&
```
//백그라운드로 실행 후, tail로 num100 파일 확인해 보면 실시간으로 변하는걸 알 수 있음

```
function sum() { declare -i sum; START=$1; END=$2; for i in `eval echo {$START..$END}`; do((sum+=i);
done; echo $sum;)
// total=$(sum 1 100); echo $total
```

~/.bashrc 를 실행하는 것과 source ~/.bashrc 를 실행하는 것의 차이는, 전자는 bashrc 에서 포함하고 있는 환경변수가 실행된 후 소멸하게 되고, 후자는 소멸되지 않고 그 값을 현재 쉘에 영향을 미친다.

(source ~/.bashrc == . ~/.bashrc 와 동일한 기능)

**<foreground>**

sleep 1000 &  이라고 하면 sleep 1000 이라는 프로세스가 백그라운드에서 작동하게 되는데, **fg** 명령어를 입력하게 되면 가장 최근에 실행한 명령어가 foreground로 되돌아 오게 된다.

<기타 운영체제>

----------윈도우 ----------

**EPROCESS 구조체** : 커널모드에서의 프로세스 구조체

프로세스가 처음 실행 될 때, 커널 메모리에 해당 프로세스의 정보를 지닌 구조체를 생성한다. 따라서, 모든 프로세스는 각각의 Eprocess 구조체를 지닌다.

또한, 프로세스에서 실행하고 있는 Thread의 개수 만큼 ETHREAD 구조체를 커널 메모리에 갖는다.

-윈도우 관점에서 프로세스란 Eprocess 구조체를 의미하고, 프로세스를 종료한다는 것은 이 구조체에 대한 포인터 값을 관련 링크드 리스트에서 삭제하는 것을 의미한다.

-위 구조체 안에는 시스템이 프로세스를 실행, 관리하기 위한 모든 정보가 있다.

**PEB 구조체** : 유저모드에서의 프로세스 구조체

프로세스 구조체가 커널(EPROCESS)과 사용자 공간(PEB)에 따로 있는 이유?

->프로세스가 자신의 Process id 값을 얻고자 할 때, EPROCESS 까지 가려면 커널의 구조체에 접근을 해야 한다. 그러나, 유저모드에서 접근이 불가능하기 때문에 시간이 오래걸린다. (시스템 콜과 같은 간접적인 방식으로 호출하여 커널에 있는 정보를 가져와야 함) 따라서, 유저 메모리에 PEB 구조체를 만들어 프로세스 정보를 좀 더 효율적으로 가져가려는 것이다.

**EPROCESS**와 ETHREAD 내부에는 **PCB**(Process control block), TCB(Thread control block) 이란 이름의 **KPROCESS**, KTHREAD 구조체가 존재한다

-KPROCESS(PCB) : 가상메모리 CR3레지스터 값, 프로세스 스레드 리스트 헤더, 등등

-KTHREAD : 스레드 DNJTS쑨위, 쓰레뜨 리스트 등 스레드에 관련된

EPROCESS 구조체 필드

-PCB 데이터 타입 : _KPROCESS

PCB, KPROCESS 위치를 가리킴

Dispatcher header, 디렉터리 테이블 주소, ETHREAD 목록, 우선순위, 등등에 대한 정보

-CreateTime 데이터 타입 : _LARGE_INTEGER

프로세스가 시작된 시간을 나타낸다

-ExitTime 데이터 타입 : _LARGE_INTEGER

프로세스가 종료된 시간을 나타낸다

-UniqueProcessID 데이터 타입 : Ptr32Void

프로세스 ID값(PID) 을 의미한다.

-ObjectTable 데이터 타입 : Ptr32-HANDLE-TABLE

오브젝트 핸들 테이블의 위치를 가리키는 포인터 값

**리눅스 부팅 과정**

사각형 캡처(R)

1. Power on
2. ROM-BIOS 프로그램 실행
    - BIOS (Basic Input/Output System): 메모리의 특정 번지로 자동 로드되어 실행됨
    - POST(Power On Self Test) 수행
        - 장착된 H/W가 인식됨. (예: 시리얼 장치, 마우스 장치, 사운드 장치)
        - H/W의 물리적인 손상체크와 초기화가 병행
    - 부팅 매체 (Disk, CD_ROM 등) 검색
    - MBR(Master Boot Record)를 읽어, 부트로더(GRUB)를 로드시킨다.
3. GRUB 실행(부트로더 실행)
    - 부팅메뉴 선택
        - /boot/grub/grub.conf 파일
        - kernel 이미지 로드
        - swapper 프로세스 호출
4. swapper 프로세서 (PID=0)
    - 커널 이미지 압축을 해제
    - 각 장치 드라이브들을 초기화
    - 루트( / ) 파일 시스템을 read-only로 마운트
    - 파일시스템 검사
    - 루트( / ) 파일시스템을 read-write 모드로 마운트
    - init 프로세스 (PID=1)
5. init 프로세스 (PID=1)
    - 리눅스에서 모든 프로세스는 그 실행과 함께 설정파일을 읽어들인다.
    - init 프로세스는 /etc/inittab이라는 설정 파일을 읽어들여 무엇을, 어떻게, 언제 실행할 것인가를 결정하여 커널 서비스 시작

Windows 정품 [설정]으로 이동하여 W

사용자 영역 VS 커널 영역 메모리 덤프 차이

-사용자 영역에서 메모리 덤프를 하게 되면 데이터들이 불완전하게 덤프가 되는 경우가 있다.
반면, 커널 영역에서 메모리 덤프를 하게 되면 데이터들이 온전하게 되는 경우가 있다.

**이는, 가상 메모리에서 사용자 영역에서 데이터가 물리 메모리와 페이지 파일을 오고 가기 때문**

-프로세스가 사용하는 데이터가 오랜기간 사용되지 않을 경우, 페이지 파일에 저장해두고 필요할 경우 다시 물리 메모리에 적재해 사용하는 방식 때문에 불완전하게 덤프가 된다.

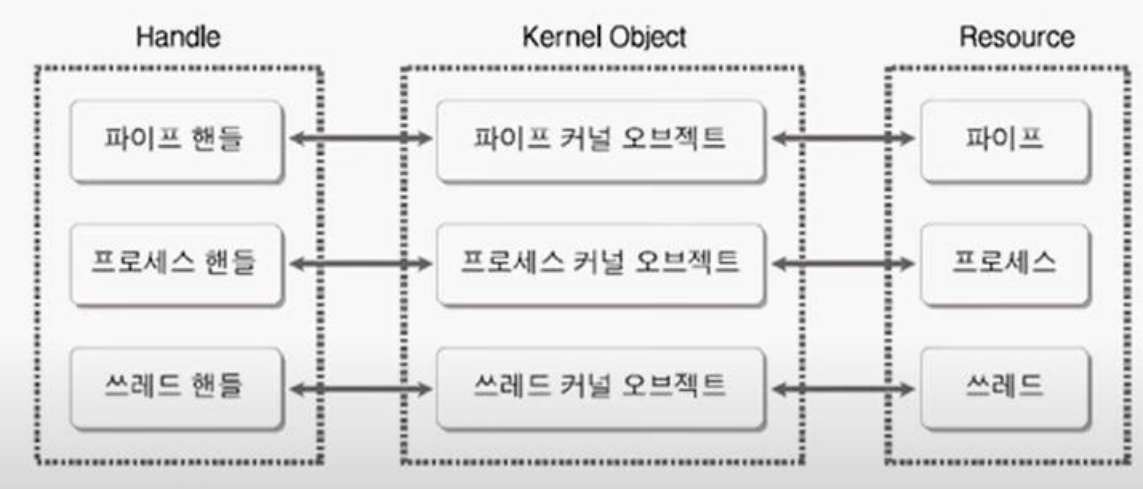
-커널 영역의 경우 시스템에 필요한 데이터들이고 자주 사용되기 때문에 페이지 파일에 저장하지 않고 물리메모리에 저장이 되기 때문에 온전하게 덤프가 되는 것이다.

커널 오브젝트와 오브젝트 핸들

**커널 오브젝트** : 커널에 의해 관리되는 리소스 정보를 담고 있는 데이터 블록
(리소스 : 누구에 의해 관리되고 소멸될 수 있는 것)

-리소스 마다 커널에 의해 관리되어야 하는 데이터가 다르다.

예로 들어, CreateProcess() 함수로 프로세스를 생성요청이 들어오면 커널은 요청대로 프로세스를 생성하고, 해당 프로세스를 관리하기 위한 프로세스 커널 오브젝트 또한 생성된다.




해당 프로세스의 우선순위를 높이기 위하여 **'프로세스 커널 오브젝트' 에 대한 임의적인 접근(reference)은 제한된다**. 따라서, 프로세스가 생성될 때 '프로세스 커널 오브젝트'가 같이 생성되고, '프로세스 커널 오브젝트'의 고유한 번호가 주어진다. 이 고유한 번호(INT)를 '핸들'이라고 하는데 이 정보를 얻을 수 있다. 위와 같은 방법으로 간접적으로 프로세스의 우선순위 정보를 높힐 수 있다.
(리소스 마다 핸들 정보를 얻는 방법이 다름)

이건 별갠데, sleep(10) 하면 10ms 동안 해당 프로세스의 상태가 Running->Blocked 로 바뀐다.

쓰레드 vs 프로세스

-쓰레드가 실행될 수 있는 환경을 제공해주는 것이 '프로세스' 이고, '쓰레드'는 '프로세스'의 환경 내에서 동작한다.

-기본적으로 스레드는 프로세스의 code,data,heap영역을 공유한다

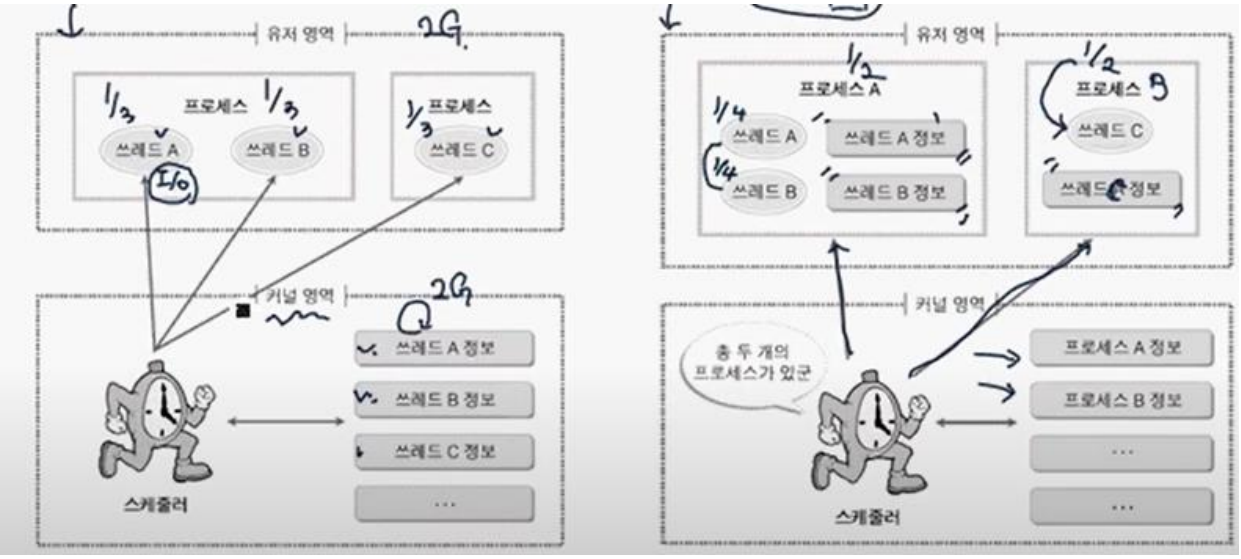
(프로세스의 code 영역에 스레드 자신의 독립적인 code가 있다는 말)

-다만, stack 영역은 별도로 분리한다. (독립된 메모리 공간)

-스레드는 프로세스 내에 존재하는 '실행의 흐름' 이다.

**커널 레벨 스레드 vs 유저 레벨 스레드**

-커널에서 '스레드' 기능을 지원하면(API형태로) '커널 레벨 스레드' 이고, 커널에서 지원하지 않고 개발자가 '라이브러리' 형태로 스레드를 만들면 '유저 레벨 스레드'이다.




커널 레벨 스레드와 유저 레벨 스레드의 가장 큰 차이점은 커널이 '스레드'의 존재를 알고 관리를 하느냐 이다. 커널레벨 스레드인 경우 커널이 '스레드'의 존재를 인지하고 있어서, 스레드A가 I/O 상태에 빠진다고 하면 다른 스레드에게 실행시간을 분배할 수가 있다. 하지만, 유저레벨 스레드는 커널이 스레드의 존재를 알지 못하여 프로세스 단위로 관리를 하기 때문에 스레드A가 I/O 상태에 빠진다고 하면, 곧바로 스레드C로 실행시간이 넘어가 비록 스레드B가 온전히 실행할 준비가 되있다고 하더라도 실행이 불가능해진다.

커널 레벨 스레드인 경우, 스레드 관련 변경된 작업이 일어날 때마다 커널모드로 전환이 되어야 하기 때문에 유저 레벨 스레드 보다 느리다.

윈도우의 스케줄러가 스케쥴링 하는 대상은 '프로세스'가 아니라 기본적으로 '스레드' 이다.

지금까지의 main함수에서 실행 되는 것은, main 스레드가 main 함수를 호출해서 실행하는 스레드가 1개인 형태이다.

직접 커널 객체 조작(DKOM – Direct Kernel Object Manupulation)
-DKOM을 사용하는 루트킷은 핵심부에서 자신을 객체 관리자 또는 작업 관리자로 부터 숨는다.
-모든 활성화된 스레드 및 프로세스의 연결리스트를 수정한다.
-시스템 커널이 실행 중인 모든 프로세스의 리스트를 찾고자 할 때는 EPROCESS에 의존하지만, 윈도우 커널은 프로세스 기반이 아니라 스레드 기반이기 때문에 연결 리스트의 포인터는 수정 가능하다.

---

<KOCW 금오공대 – 커널 및 프로그래밍>
PCB (프로세스 컨트롤 블록)

이런 복잡한 프로세스들을 관리하기 위하여 PCB(task_struct 구조체 )를 만들어야 한다.
(task는 리눅스에서 process를 의미함) - <linux/sched.h>

User space 입장에서는 시스템 콜을 사용하는데, 대부분이 프로세스의 작업에 관련되어 있다
프로세스에 대한 작업을 하기 위해서는 task_struct 구조체를 알아야 한다.
유저는 thread_info 의 구조체를 통해서 task_struct 구조체의 위치를 알 수 있다.
**커널 스택의 위치는 시스템 콜이 걸렸을 때 esp 가 가리키고 있다.**

```
struct thread_info {
        struct task_struct *task;
        struct exec_domain *exec_domain;
        ....
}
```

커널스택은 유저스페이스의 스택처럼 파라미터 넘기고 코드(함수)를 쌓아둘 수 있는 별도의 스택.
(커널 스택의 바닥에 thread_info 구조체가 있다, 커널 스택은 대채로 8KB=8*1024byte임)
따라서, 유저 입장에서 thread_info 위치인(struct thread_info *)(%esp-8*1024)을 알 수 있다.
커널 스택은 프로세스 개수 만큼 존재한다.
커널 스택의 위치는 프로세스가 실행시킬 때 %esp의 위치이다.

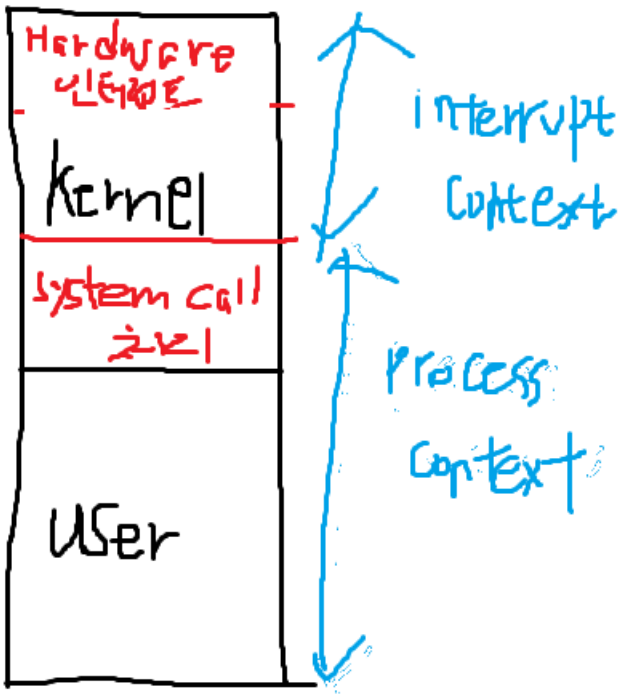
<프로세스 인식 번호 – PID>
-int형이지만 최대 32,768개
-<linux/thread.h> 에서 4백만으로 조정 가능

**-리눅스에서는 ready 상태가 없고 ready queue에서만 있으면 ready, run 상태로 규정한다.**

**프로세스 상태**
-TASK_RUNNING
-TASK_INTERRUPTIBLE
-TASK_UNINTERRUPTIBLE
-_TASK_TRACED : 디버깅을 당하고 있는 프로세스의 상태
-_TASK_STOPPED : signal을 받은 경우 잠시 정지된 상태
**현재 프로세스 상태 조작**
set_task_state(task, state) 매크로 == task->state = state;

process context

원래는 User, Kernel 2개의 부분으로 구성되어 있는데 커널 입장에서 재정의 할 수 있다.




1) process context에 있으면서 User 일 때 : User stack 사용
2) process context에 있으면서 system call 일 때 : Kernel stack 사용
3) interrupt context에 있으면서 interrupt일 때 : 별도의 stack 사용 (CPU갯수가 stack 개수임)

task_struct {
    pid
    state
    (**parent** 1개 : 그냥 부모 task_struct에 연결, **children** 복잡 : 더미노드를 둔 원형연결리스트로 구성되어 있음)
}

리눅스가 처음 부팅되면 맨 처음 init 프로세스가 실행된다. 이 것을 통해서 다른 모든 프로세스가 만들어지고 실행된다.

**list for_each( list , &current->children)** {

**task = list entry(list, struct task_struct, sibling);**

}

-list : task_struct 내의 sibling 필드 가리킴

-task : task_struct 가리키는 포인터

init_task 까지 프로세스는 부모를 찾으러 떠날 수 있다.

(init_task는 전역변수)

for(task = current; task != &init_task; task = task->parent)

결과 : task는 init_task를 가리키게 됨

list_entry 매크로

**task_struct** {

parent, child

sibling ( prev, next) 가 있음 : 자신의 형제 연결

tasks (prev, next)가 있음 : 전체 프로세스 연결 필드

pid

comm : task 이름

}

여기서 , list 라는 포인터는 특정 task_struct의 sibling 필드를 가리킨다. 다음과 같이,

struct task_struct *task;

list_entry(task->tasks.next, struct task_struct, tasks) 이면 **다음 task 시작위치를 반환하는 매크로** 이다.

list_entry(task->tasks.prev, struct task_struct, tasks) 이면 이전 task 시작위치를 반환하는 매크로 이다.

fork() : 현재 task를 복제해 자식 태스크 생성

-막 생성된 자식 task는 부모와 같은 주소공간 공유

-부모의 페이지테이블 복제

-자식의 file descriptor table 생성

따라서, fork() 와 exec() 사이에 gap이 존재해서 file descriptor 를 조작가능 (출력,입력 바꾸기 알잖아)

(stdin, stdout, stderr 조작이라는 말 – **dup2(), dup 사용** )

exec() : 새로운 실행파일(프로그램)을 주소 공간에 불러옴

-write가 발생하면 사본이 만들어지고 변경된 내용이 저장됨

-리눅스에서는 부모 task가 자식 task를 만들면 부모 task는 ready queue로 가고 자식 task가 cpu를 얻어서 수행한다.

**copy_process()**
-dup_task_struct() 함수 호출
(커널 스택 생성, thread_info, task_struct 생성) –> 부모와 자식의 모든 내용은 동일
-프로세스 개수 제한 확인
-부모 / 자식 구별 (task_struct 의 parent, child 필드 )
-자식 상태 : TASK_UNINTERRUPTIBLE
-alloc_pid() 함수 수행 : 자식에 새로운 PID를 할당해서 부모에게 리턴한다
-여러 자원들을 공유하거나 복제

리눅스의 스레드 구현 : 공유 할 것을 지정할 수 있다.
**clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);**
주소 공간 공유, 파일 디스크립터 공유, 파일, 시그널 같이 받겠다
clone(SIGCHLD, 0) 이라고 하면 fork()와 유사하다. (시그널만 공유)

**커널 스레드** : 커널 공간에서 백그라운드로 수행되는 스레드
-커널 스레드는 유저 fork()와는 다르게 **커널 주소공간을 공유한다**.
-커널 스레드간 context switching이 유저공간에서의 프로세스간 context switching 보다 훨씬 light함
ex**) kthreadd** 와 같은 커널 스레드를 만드는 커널 프로세스가 ksoftirqd와 같은 커널 스레드를 만듬
-새로운 커널 스레드 생성은 <linux/kthread.h> 에 정의된다

일반 스레드를 만들 때는 clone, fork를 쓰는데, 커널은 struct *task_struct kthread_create() 를 사용
위와 같이 커널 스레드를 만들면 block 된 상태이다.
struct task_struct *kthread_run()을 사용하면 내부에서 kthread_create()를 사용하여 바로 ready queue 에 들어간다

내가 exit()를 집어넣지 않아도 컴파일러가 맨 마지막에 묵시적으로 exit()를 집어 넣어준다

*exit()를 하면 열려지는 fd 와 같은 모든것이 닫히지만 task_sturct 구조체의 exit_coid 필드는 생존
task_struct 의 exit_code 에 종료코드를 삽입하고 wait(&status)한 부모가 가지러올때까지 살려준다
task_struct {
**exit_code**
}
왜나하면, 부모 입장에서 자식의 exit(종료코드)를 wait(&status)로 받을 수 있기 때문에 자식이 종료가 되더라도 task_struct의 exit_code는 살아남는다.

exit_mm() : 메모리 반환, del_timer_sync() : 커널 타이머 제거, exit_fs() : fd 참조 횟수 감소

release_task() :
-_exit_signal() : _unhash_process() 호출, detach_pid()로 해당 프로세스는 pidhash와 task list에서 제거
(남은 자원 반환)

고아 프로세스가 계속 고아가 되길 유지한다면 놔두고 아니라면 새로운 부모를 정해줘야 한다
(init 프로세스 or 디버거 )

멀티 태스킹 : 하나 이상의 프로세스를 동시에 중첩 형태로 실행

**우선순위**
-일반 우선순위 : 기본값에 nice ( -20 ~ 19)를 더하여 일반 프로세스의 우선순위 결정
(낮은 nice 값이 높은 우선순위를 의미함)
-실시간 우선순위 : 0 ~ 99 사이의 값은 우선순위가 유효하고, 100이상은 거의 다 같다고 본다
(**우선순위가 비슷하면 cpu 사용량을 맞추기 위해 cpu 덜 쓴 프로세스를 선택한다)**
(높은 값이 높은 우선순위를 의미함)

SCHED_FIFO
SCHED_RR
SCHED_NORMAL : cpu 사용량에 따라서 선점 여부를 가림

**CFS**
**기본 개념** : n개의 프로세스에게 각각 1/n의 process time 제공
현실적 문제 : 스케쥴 간격이 짧으면 context switch overhead가 높아짐
대안 : 순차적으로 일정시간 수행하고, 가장 실행이 덜 된 프로세스를 다음 대상으로 스케쥴 한다.
-targeted latency : CFS가 설정한 가장 작은 스케쥴링 단위
-minimum granularity : context switch ocst를 감당할 수 있는 최소한의 time slot

<nice 값이 1씩 차이나는 경우>

| Process | Nice | Time slice | Ratio |
|---|---|---|---|
| p1 | 0 | 100ms | 100% |
| p2 | 1 | 95ms | 95% |

<nice 값이 1씩 차이나는 또 다른 경우>

| Process | Nice | Time slice | Ratio |
|---|---|---|---|
| p1 | 18 | 10ms | 100% |
| p2 | 19 | 5ms | 50% |

->순수 nice 값으로 time slice를 결정하게 되면 위와 같은 문제점이 생김

->**별도의 CFS 방식으로 사용하게 되면 Nice 값이 다르더라도 비율이 같게 되는 장점이 있음(공평성)**

**(순수 nice 값으로만 time slice 를 결정하지 않는다)**

vruntime : 가상 수행 시간

-nice 값에 따라 자신이 수행하는 속도가 다르게 적용된다 (마치 상대성 이론마냥)

-ex. nice 값이 달라서 본인 프로세스가 cpu를 수행한 시간이 각각 다른 것이 일반적이다. 그러나, vruntime 이라는 개념을 사용하여 비록 실제로 cpu를 수행한 시간이 다르더라도 같은 시간만큼 사용한 것처럼 속이기 위해서 사용한다. (즉, **그냥 실제 cpu 수행시간보다 더 사용한 것 처럼 속이는 것**)

(vruntime을 계산하는 식이 따로 있음 – 어렵지 않음)

-각 TASK가 쫓겨날 때 vruntime 값을 갱신한다.

-ready queue는 vruntime이 작은 값을 기준으로 정렬이 되어 있다.

->이 때문에 **I/O를 계속 하는 task는 다른 task보다 vruntime이 작을 확률이 높기 때문에 먼저 시행될 확률이 높다.**

Time accounting

**-스케쥴러는 각 프로세스의 실행시간을 기록함**

**-시스템 클록이 한 tick 지날 때 마다 수행 중인 프로세스의 time slice 를 감소**

**-time slice 가 0이 되면 다른 프로세스를 스케쥴**

```
struct task_struct {
        ...
        struct sched_entity se;
}
struct sched_entity {          //ready queue에 실제로 들어가는 구조체 (vruntime으로 정렬, 대기)
        ...
        u64 vruntime;
}
```

**스케쥴러가 호출될 때마다 수행 중인 task의 vruntime을 바꿔줌**

**sched_entity 자료구조**

**=> 커널 상에서 스케쥴링의 기본 단위, 그룹 스케쥴링의 구현을 위한 핵심(내부에 cfs 실행큐 有)**

**=> 기본단위 : task_struct를 대신해 스케쥴러의 선택 대상이 됨,**

```
static void update_curr(struct cfs_rq *cfs_rq) {
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_of(cfs_rq)->clock;
        unsigned long delta exec;
        if(unlikely(!curr)) return;
        delta_exec = (unsigned long)(now – curr->exec_start);
        if(!delta_exec) return;
        __update_curr(cfs, rq, curr, delta_exec);
        ....
}
static unline void __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr, unsigned long delta_exec) {
        unsigned long delta_exec_weighted;
        schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));
        curr->sum_exec_runtime += delta_exec;
        ...
        curr->vruntime += delta_exec_weighted;
        update_min_vruntime(cfs_rq);
}
```

**새로 수행될 task 고름(ready_queue에서 제일 왼쪽에 있는 = vruntime이 제일 적은 task 고름)**

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq) {
        struct rb_node *left = cfs_rq->rb_leftmost;
        if(!left) return NULL;
        return rb entry(left, struct sched entity, run node);
}
```

time slice : 실행의 최소 단위 시간, 이를 기준으로 정해진 시간마다 cpu 할당
cpu burst time : cpu가 일을 수행하는 시간
->kocw 커널 12차시 8분 그림보면 바로 이해됨

프로세스마다 task_struct가 있는데 ready queue에 task_struct 자체가 들어가지 않고 task_struct 안의 struct sched_entity 가 ready_queue에서 기다린다.

struct sched_entity 안에 vruntime이 있고, vruntime을 key 값으로 ready queue가 정렬된다.

**리눅스에는 ready queue가 3개가 있다.**
(우선순위가 100보다 작은 – **시스템 프로세스 queue가 2개, 100보다 큰–유저 프로세스 queue가 1개**)

리눅스의 **CFS(Ready queue)에서는 red_black_tree 를 사용**한다. **AVL 트리보다 덜 엄격하고, 삽입 삭제가 많이 일어나는 리눅스 CFS 특성에 최적화됨.**

<디바이스 드라이버>
-리눅스에서 많은 객체들은 파일의 형태로 접근 가능 (모니터, 키보드, 마우스, 정규파일 등)
-device를 가리키는 파일을 장치파일(device file)이라고 한다.
- H/W 형태의 센서나, 네트워크 카드들은 모두 device file에 의해서 통제된다.
- **TASK가 VFS의 각 명령을 호출했을 때 이에 적합한 함수를 제공하는 역할을 담당한다.**
ex. 키보드 장치에 read()를 요청하면 키보드 입력 값을 넘겨주는 함수를 제공해야 함
이 함수를 file_operations 구조체의 함수에 매핑

**디바이스 드라이버 정보**
1) **고유 번호(major number)** : **다양한 디바이스 드라이버들을 구분**하기 위한 번호
-0~255 사이의 값을 가짐
-**inode 객체의 i_rdev 필드에 존재** (include/linux/fs.h)
**i_rdev : 드라이스 드라이버의 주 번호와 부 번호로 구성**
2) **부 번호(minor number)** : **동일한 디바이스가 여러 개가 있을 때 각 디바이스를 구분**
-콘솔, 모니터인 경우 : /dev/tty0 4,0 : 장치 파일 이름은 tty0 , 주번호는 4, 부 번호는 0
-하드디스크인 경우 : /dev/hda1 3, **1** , 두번째 하드디스크 : /dev/hdb1 3, 65
(1 : 첫번째 파티션, 65 : 65번째 파티션)
open("/dev/hda1", "r") 와 같이 디바이스 파일을 오픈할 수가 있다.

**mknod** /dev/mydrv [c][b] 주번호 부번호 : 장치파일 생성 예
inode 안의
-i_rdev 에는 주번호와 부번호 저장
-i_mode 에는 S_IFCHR (Character타입 디바이스 파일), S_IFBLK(Block타입 디바이스 파일) 저장

디바이스 드라이버 종류

1) **character device driver**

-순차접근

-터미널 드라이버

-struct file_operation 구조체에 지정된 함수를 통해 접근

2) **block device driver**

-임의 접근

-디스크 드라이버

-큐를 통해 '버퍼 캐시'와 통신

-struct block_device_operations 에 지정된 함수를 통해 접근

3) Network device driver

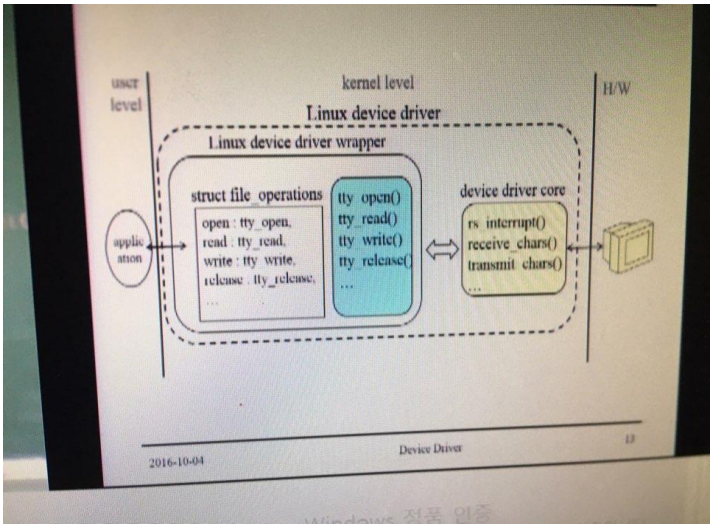
효율적인 관리를 위해 디바이스 드라이버는 다음의 두 부분을 나누어 설계함

1) **하드웨어와 밀접한 코드**

->**디바이스 드라이버 코어** : 하드웨어 메뉴얼을 바탕으로 만드는 **디바이스 제어 함수** (전자공학과)

2) **운영체제와 밀접한 코드**

->**래퍼(wrapper) : 사용자에게 제공할 함수와 코어를 연결시키는 부분**




디바이스 드라이버에 대한 입장

1) 사용자 : 커널에 기본적으로 제공하는 system call을 통해 디바이스를 접근할 수 있어야 한다

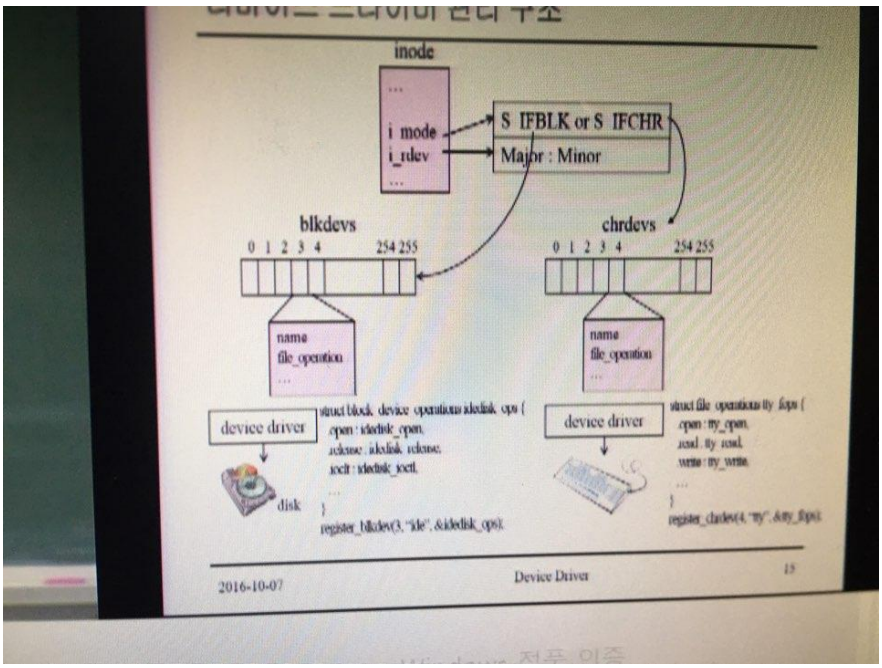
2) 디바이스 드라이버 개발자 : 리눅스가 정의한 file_operation 구조체에 정의된 함수를 구현

3) 커널 :

**-문자형 디바이스 드라이버를 위한 배열 유지 : chrdevs**

**-블록형 디바이스 드라이버를 위한 배열 유지 : blkdevs**

-각 배열의 크기는 256 (디바이스 주번호가 0 ~ 255 사이이므로)




**디바이스 드라이버 등록**

**1) chrdevs 배열인 경우** : register_chrdev() 함수로 등록

**int register_chrdev** {

```
        unsigned int major,          //major number 등록, 모를 경우 0으로 하면 알아서 해줌
        const char *name,            // 디바이스 이름 : ex. /dev/mydev
        struct file_operation *fops   // open,read... 등이 들어있는 file_operation 구조체
```

}

// **major number에 0을 넣어주면 커널이 쓸 수 있는 번호를 리턴해 준다**

// 리눅스에서 각 디바이스가 사용하는 주 번호는 따로 테이블로 정의되어 있다.

-> cat /proc/devices 로 확인가능

**2) blkdevs 배열인 경우** : register_blkdev() 함수로 등록

**디바이스 드라이버를 추가하는 과정**

1) 드바이스 드라이버 **코어 함수 구현** – 하드웨어 메뉴얼 기반으로 작성

2) **래퍼 작성** – 리눅스 file_operation 구조체 기반 접근 방식을 제공하기 위해

3) 드라이버를 커널에 **등록** – register_XXXdev() 같은 커널 내부 함수 이용

4) **장치 파일 생성** – mknod 이용

문자 디바이스 드라이버 구현

```
static ssize_t mydrv_read(struct file *file, char *buf, size_t count, left_t *ppoa) {
        if((buf == NULL) || (cout < 0) ) return -EINVAL;
        if((mydrv_write_offset – mydrv_read_offset) <= 0) return 0;
        count = min(( mydrv_write_offset – mydrv_read_offset), count);
        if(copy_to_user(buf, mydrv_data + mydrv_read_offset, count)) return -EFAULT;
        mydrv_read_offset += count;
        return count;
} // mydrv_data : 할당된 드라이버 크기
unsigned long copy_to_user (void _user *to, void * from, unsigned long n);
```

to : user space에 있는 목적지 주소 , from : kernel space에 있는 소스 주소

n : copy 되는 데이터 byte 수

->리턴값 : 성공시 0

-> 커널 영역에서 유저 영역으로 데이터를 copy

copy_to_user 함수를 쓰는 경우

1) buf가 사용자가 접근 가능한 영역인지?

2) buf가 page out 처리 되지 않았는지 처리

```
static ssize_t mydrv_write(struct file *file, const char *buf, size_t count, left_t *ppoa) {
        if((buf == NULL) || (count < 0)) return -EINVAL;
        if(count + mydrv_write_offset >= MYDRV_MAX_LENGTH) return 0;
        // 할당된 드라이버 크기가 너무 작으면
        if(copy_from_user(mydrv_data+mydrv_write_offset, buf, count)) return -EFAULT;
        mydrv_write_offset += count;
        return count;
}
// 유저 영역에 있는 데이터를 커널 영역으로 copy 하는 함수
struct file_operations mydrv_fops = {
        .owner = THIS_MODULE,
//THIS_MODULE macro: 코드가 적재되는 메모리 상의 위치
Extern struct module_this_module;
#define THIS_MODULE   (&_this_module) -> include/linux/module.h
-Module이 적재되는 메모리의 시작 주소(insmod시에 값이 결정되어 기록됨)
        .read = mydrv_read,
```

```
        .write = mydrv_write,
        .compat_ioctl = mydrv_ioctl,
        .open = mydrv_open,
        .release = mydrv_release,
}
int mydrv_init() {
// 모듈을 집어넣을 때, 커널에서 꺼낼 때 수행해야 하는 작업들(register 등록 작업)
        if( (MYDRV_MAJOR = register_chrdev(0, DEVICE_NAME, &mydrv_fops) ) <0) {
                                        // major 번호, name, file_operation
                printk(KERN_INFO "can't be registered ₩n");
                //printk : 커널에서 쓰는 printf (커널 특정 영역에 저장 – dmegs로 출력가능)
                return MYDRV_MAJOR;
        }
        printk(KERN INFO "major NO - %d₩n", MYDRV_MAJOR);
        if( (mydrv_data=(char *)kmalloc(MYDRV_MAX_LENGTH * sizeof(char), GFP_KERNEL)) == NULL){
        //kmalloc : 커널 용도의 malloc  //GFP_KERNEL : '왠만하면 기다려 주겠다' 라는 옵션
                unregister_chrdev(MYDRV_MAJOR, DEVICE_NAME);
                return -ENOMEM;
        }
        mydrv_read_offset = mydrv_write_offset = 0; //잘 할당 받았으면 offset값 0으로 초기화
        return 0;
}
```

**kmalloc()**

-<linux/slab.h> 에 정의됨

-void *kmalloc(size_t size, int flags) -> return 값 : 할당된 메모리의 위치

-flag

1) **GFP_KERNEL** :  **GFP_WAIT | GFP_IO | GFP_FS 가 합쳐진 매크로** (느슨한 옵션)

-GFP_WAIT : the allocator can sleep

-GFP_IO : the allocator can start disk I/O

-GFP_FS : the allocator can start filesystem I/O

2) **GFP_ATOMIC** : GFP_KERNEL아 상반된 옵션 ( 인터럽트와 같은)

-주로 시스템 상에서 메모리를 할당받지 못해서 NULL이 리턴됨

```
void mydrv_cleanup() {
//디바이스가 빠져나갈 때 – rmmod 사용
```

```
        kfree(mydrv_data);
        unregister_chrdev(MYDRV_MAJOR, DEVICE_NAME);
}
module_init(mydrv_init);
module_exit(mydrv_cleanup);
```

**디바이스 모듈이 제거가 안되는 경우** : 다른 task가 device file을 사용하고 있는 중간에 rmmod 를 수행하는 것은 허용되지 않음
따라서, **counter** 라는 것을 사용해서 device file을 open 할 때 마다 1씩 증가시키고 close 할 때 마다 1씩 감소시킨다.

수행과정

```
make
su
insmod char_test.ko
dmesg | tail -1
[ 9799.563834 ] major NO = 249
mknod /dev/mydrv c 249 0
chmod o+w /dev/mydrv
exit
gcc -O -o mydrv_test mydrv_test.c
./mydrv_test
```

//mydrv_test는 유저 영역에 있고 다른 커널영역에 작성한 모듈들이 잘 작동하는지 테스트

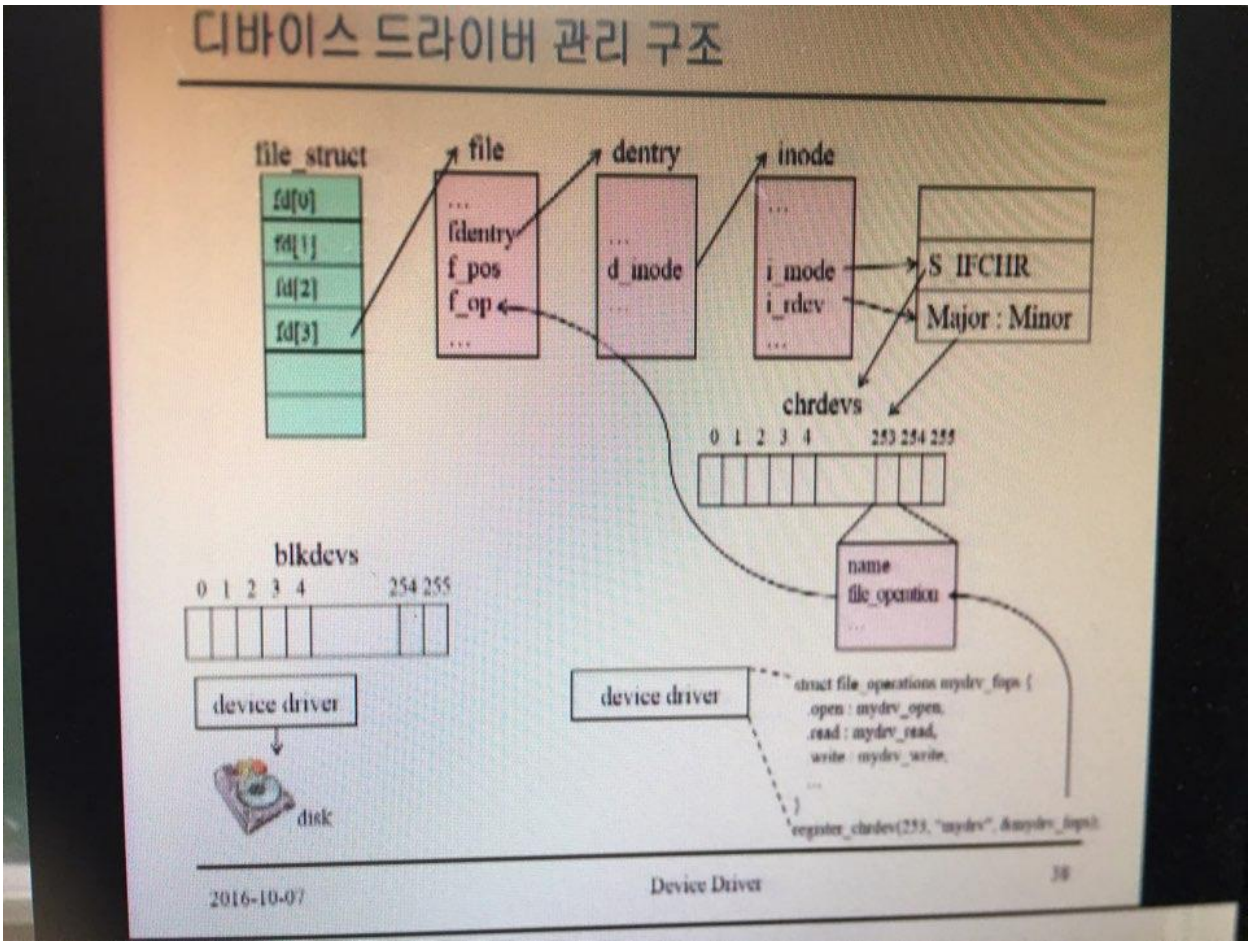
**//mydrv_test.c**

```
#include <stdio.h>
#include <fcntl.h>
#define MAX_BUFFER 26
char buf_in[MAX_BUFFER];
char buf_out[MAX_BUFFER];

int main() {
        int fd, i, c = 65;
        if ((fd = open("/dev/mydrv", O_RDWR)) <0) {
```

```
                perror("open error");
                return -1;
        }
        for(i = 0; i<MAX_BUFFER; i++) {
                buf_out[i] = c++;
                buf_in[i] = 65;
        }
        for(i = 0; i<MAX_BUFFER; i++) fprintf(stderr, "%c", buf_in[i]);
        fprintf(stderr, '₩n');
        write(fd, buf_out, MAX_BUFFER); // dev/mydrv 에 쓴다 (mydrv_data)
        read(fd, buf_in, MAX_BUFFER);
        for(i = 0; i<MAX_BUFFER; i++) {
                fprintf(stderr, "%c", buf_in[i]);
        }
        fprintf(stderr, "₩n");
        close(fd);
        return 0;
}
```

여러 응용프로그램들이 동시에 /dev/mydrv에 접근할 때 어떤 문제가 생길 수 있겠는가?




fd 를 만들면 , file에 대한 구조체, dentry에 대한 구조체를 만들어주고 해당 inode를 찾는다

스케쥴러 진입 위치

likely 매크로 : 조건문이 참일 거 같으니 컴파일러에게 그 다음 코드를 캐쉬로 받아오라는 의미

**waiting queue** : 프로세스들을 관리하는 집단
-프로세스는 어떤 사건이 일어나기를 기다릴 때 그에 맞는 waiting queue에 삽입된다.
그리고, waiting queue에 들어가 있던 프로세스는 그에 맞는 조건이 되면 깨어나고 waiting queue에서 제거된다.

wake_up() : waiting queue에 있는 모든 task들을 깨움
-try_to_wake_up()을 깨우는데 모든 task_struct를 TASK_RUNNING으로 변경함
-그리고 ready_queue에 집어넣음
-resource가 도착(인터럽트) 혹은 signal 도착– **condition에 따라 wake_up()** 한다.
**(두 가지 경우임)**

MMU는 내부에 PTBR 이라는 실행 페이지 테이블을 가리키는 레지스터가 있다.

**need_resched flag** : 커널에게 schedule이 필요함을 알리는 flag
(set_tsk_need_resched() 함수를 통해서 세팅(1)할 수 있음, clear_tsk_need_resched() 은 세팅해지(0), need_resched()는 need_resched_flag 리턴 )
-예전에는 need_resched_flag 전역변수를 직접 접근이 가능했는데 이제는 set_tsk_need_resched()를 통해서 해야한다.
-해당 flag는 스케쥴링의 필요성만 세팅하는 거고, 실질적으로 스케쥴링을 하는 것은 항상 **시스템 콜이 끝나기 직전 schedule() 함수를 사용하여 하게 된다.**
-need_resched flag를 thread_info 구조체로 옮겼다.

커널이 시스템 콜을 수행하다가 중간에 인터럽트가 걸렸을 때 커널 preemption이 발생한다.
다만, lock이 걸리지 않은 (민감하지 않은 자료구조)를 수행하고 있을때 preemtion이 발생함
-이를 'safe state' 라고 하는데 **thread_info->preempt_count** (초기값 0, lock하면 +1)가 **0일때 선점(preempt) 될 수 있는 조건이 된다**

(하드웨어가 소프트웨어쪽보다 우선순위가 더 높기 때문에)
safe state 인 경우
-interrupt routine이 끝나고 kernel space로 돌아갈 때 등등

요즘은 cpu가 듀얼인데, 커널이 system call을 수행하고 있을 때 대체로 system call을 수행하고 있지 않은 **다른 cpu에게 인터럽트가 걸리도록 운영체제가 조정을 하고 있다**.
**real-time** : **시간을 반드시 엄수해야 되고 오류가 반드시 없어야 하는** (미사일 하드웨어 등등)
->**task의 수행시간, cpu 사용률을 미리 예측해야 함**
->리눅스 커널은 real-time-behavior을 보장하도록 설계되어 있지 않음

real-time scheduling policy
-real – time priority 범위 : 1 ~ 99
1) sched_fifo
2) sched_rr
real-time scheduling policy가 아닌 일반 스케쥴링 정책
1) sched_normal : 범위 : 100 ~ 139

시스템 콜로써 nice 값을 설정할 수도 있다
-(sched_setscheduler(), sched_getscheduler() : 특정 프로세스의 real-time-priority 설정)

-nice() -> 시스템 콜이 아닌 일반적으로 설정
(음수설정은 root만 가능하고 일반 사용자는 0 ~ 19까지만 설정가능하다)

perror : 전역함수로써 문자열 형태의 오류 메세지를 출력가능하다

system call 예시 (getpid())
**SYSCALL_DEFINE0 (getpid)** **->0은 parameter가 없다는 의미**
{
return task_tgid_vnr(current);
}
**SYSCALL_DEFINE0 매크로**
=>asmlinkage long sys_getpid()
**-asmlinkage : parameter는 stack에서만 찾음**
-prefix sys가 함수 명 앞에 넣음

각 syscall에는 고유 번호가 할당됨 (프로세스가 syscall을 호출할 때 사용)
sys_ni_syscall() : 구현되지 않은 시스템 호출을 대표하기 위해 제공
: error code return (다른 일은 안함)
모든 syscall 목록을 sys_call_table 이라는 테이블에 저장한다 (syscall 마다 number 부여)

system call handler (점프가 총 2번 일어남 – 빨간색)
**사용자가 system call을 호출할 때, 호출한 시스템 콜의 번호가 eax 레지스터에 저장**
1) software interrupt를 발생시킴
2) kernel mode로 전환
**3) interrupt handler vector 128번에 있는 주소로 이동**
4) 해당 주소에는 **system_call() 함수 테이블이 존재함 ( 번호 + 시스템 콜 함수 주소)**

5) eax에 들어있는 시스템 콜 번호를 NR_syscalls(systemcall 테이블에 들어있는 번호 크기)와 비교
->eax에 들어있는 값이 NR_syscalls 의 값 초과이면 오류
**6)** eax 레지스터 값에 해당하는 시스템 콜 함수의 주소로 이동

User space Kernel space

Application => read() wrapper => system_call() => sys_read()
(call read() ) C library read() wrapper Syscall Handler sys_read()

system call 호출 parameter
1) 5개 이하인 경우 : ebx, ecx, edx, esi, edi 레지스터 순서대로
2) 5개 초과인 경우 : parameter를 구조체에 저장해서 ebx에 주소를 넘겨준다

**return value : eax에 저장**

parameter verifying : syscall은 parameter들이 유효하고 적당한지 확인해야 함

Kernel space와 User space 사이의 Data Copy
-Kernel 이 정보를 User space에 보낼 때 직접 포인터를 사용하면 안됨
=> User space에 page fault가 발생할 수 있기 때문에
-**copy_to_user(p1,p2,p3)** : user space에 쓰는 경우
=>p1 : destination memory address
=>p2 : kernel space의 source pointer
=>p3 : copy할 데이터의 byte 개수
-**copy_from_user(p1,p2,p3)** : user space로부터 읽는 경우
=>p1 : kernel space의 destination pointer
=>p2 : source memory address

성공하면 0을 리턴, 실패하면 copy에 실패한 byte 수를 리턴
**커널로 들어오게 되는 2가지 경우**
1) **System call Context**
-System call이 실행 중일 때 커널은 'process context'에 있음
=>현재 수행되는 프로세스의 task_struct를 가리키는 current 를 통해서 접근이 가능하다.
=>커널 스택을 쓸 수 있다. (커널 스택에 thread_info가 있기도 하고)
=>커널은 preempt 될 수 있고 sleep 될 수 있다
2) **인터럽트**
=>인터럽트를 통해 커널로 들어오게 되면 1번에 해당하는 것이 없다.
=>커널은 preempt 될 수 없고 sleep도 될 수 없다.
=>부득이하게 상대방이 준비가 안되서 기다려야 한다면 인터럽트 핸들러 루틴 코드안에서 for( ;;) – buzy waiting 을 한다.

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER) 매크로
TYPE : 부모 구조체의 TYPE, MEMBER : 부모 구조체에 있는 filed
ex. int offset = **offsetof(struct fox, list)** 이라 하면 offset은 **부모 구조체에 있는 field의 시작주소**

```
#define container_of(ptr, type, member) ({ ₩
        const typeof( ((type *)0)->member ) * __mptr = (ptr); ₩
        (type *) ( ( char *)__mptr – offsetof(type,member) ); })
```

->**결과적으로, 부모 구조체의 시작 포인터를 리턴**

->**(char *)__mptr** : 부모 구조체의 특정 필드의 시작주소를 char 으로 바꾸는 이유는 1바이트식 계산하기 위해서 바꾼다

->ptr : 부모 구조체내의 특정 field를 가리키는 포인터임

Linux의 linked list 예제

```
struct list_head {
        struct list_head *next;
        struct list_head *prev;
}
struct fox {
        int weight;
        struct list_head list;
}
int main() {
        struct fox *myFox, *myEntry;
        struct list_head *myPtr;
        myFox = (struct fox *)malloc(sizeof(struct fox));
        myFox->weight = 10;
        myPtr = &(myFox->list);
        myEntry = list_entry(myPtr, struct fox, list); => container_of 매크로와 똑같다
        printf("%d₩n", myEntry->weight);
        return 0;
}
```

**list_from_each 매크로**

```
struct list_head *p;
list_for_each(p, fox_list) {
        /* p points to an entry in the list */
}
```

-fox_list는 부모 구조체 안의 list 구조체를 가리킴

-p는 fox_list에 있는 node 들을 하나씩 가리킴. 되돌아 오면 종료

실제 데이터에 접근하려면

```
struct list_head *p;
struct fox *f;
list_for_each(p, &fox_list) {
        f = list_entry(p, struct fox, list);
}
```

실제 적용

```
struct fox *f;
list_for_each_entry( f, &fox_list, list) {
        /* on each iteration f points to the next fox structure */
}
```

**-&fox_list : 부모 구조체에서 list 필드의 주소**

**-f : 부모 구조체를 가리키는 포인터**

**-list : 부모 구조체에서 list 필드**

실제 예시

```
static struct inotify_watch *inode_find_handle( struct inode *inode, struct inotify_handle *ih) {
        struct inotify_watich *watch;
        list_for_each_entry(watch, &inode->inotify_watches, i_list) {
                if(watch->ih == ih) return watch;
        }
        return NULL;
}
```

kfifo : Linux queue의 기본 이름 // <linux/kfifo.h>

in offset : 작업이 입력될 위치

out offset : 빼기 작업이 일어날 위치

Queue 생성

**int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask);**

**-size byte 크기의 queue가 생성됨**

-**return value : 성공하면 0**, 실패하면 음수의 오류 코드

ex)

```
struct kfifo fifo;
int ret;
ret = kfifo_alloc(&fifo, PAGE_SIZE, GFP_KERNEL);
if(ret) return ret;
```

Queue 초기화

**void kfifo_init( struct kfifo *fifo, void *buffer, unsigned int size);**

Queue에 데이터 넣기

**unsigned int kfifo_in( struct kfifo *fifo, const void *from , unsigned int len)**

Queue에 데이터 빼기

**unsigned int kfifo_out( struct kfifo *fifo, vodi *to, unsigned int len);**

이외에도 Queue의 크기 알아내기, 데이터 byte수 알아내기, 비어있는 공간 byte 수 알아내기 가능

, Queue의 상태

**balanced binary search tree** : 모든 leaft들의 depth가 최대 1차이가나는 binary search tree

**self-balancing binary search tree** : 일반적인 연산에도 balance를 유지하는 binary search tree

(굉장히 엄격함 – 해당 특정 노드를 찾을 때 무조건 log(N)의 시간 복잡도를 가지지만 걸핏하면 lotation을 해야 되서 시간이 오래걸림)

**종류 : Red-black-tree가 있다.**

-모든 노드는 red 또는 black 이다

-leaf node는 black이고 데이터가 없다

-모든 non-leaf-node는 두 개의 자식 node가 있다.

-Red node의 자식들은 모두 black이다. 데이터가 있는 경우라면 Red, 없는 경우라면 black 이다.

-Linux에서 ready queue에서도 red black tree를 쓴다.

-부모가 블랙이면 자식은 레드로 만든다. 부모가 레드면 자식은 블랙으로 만든다.

-<linux/rbtree.h>에 선언, lib/rbtree.c 에 구현

inode 가 열어놓은 file 안에서 최근에 열린 page 들을 담고있는 cache 에서 내가 찾고싶은 위치가 있는가?

```
struct page *rb_search_page_cache (struct inode *inode, unsigned long offset) {
        struct rb_node *n = inode->i_rb_page_cache.rb_node;
        while(n) {
                struct page *page = rb_entry(n, struct page, rb_page_cache);
                if(offset < page->offset) {
                        n = n->rb_left;
                }
                else if(offset > page->offset) {
                        n = n->rb_right;
                }
                else {
                        return page;
                }
        }
}
```

rbtree 삽입 함수 예시

```
struct page *rb_insert_page_cache( struct inode *inode, unsigned long offset, struct rb_node *node) {
        struct rb_node **p = &inode->i_rb_page_cache.rb_node;
        struct rb_node *parent = NULL;
        struct page *page;
        while(*p) {
                parent = *p;
                page = rb_entry(parent, struct page, rb_page_cache);
                if(offset < page->offset) {
                        p = &(*p)->rb_left;
                }
                else if(offset > page->offest) {
                        p = &(*p)->rb_right;
                }
                else return page;
```

```
        }
        rb_link_node(node, parent, p);
        rb_insert_color(node, &inode->i_rb_page_cache);
        return NULL;
}
```

**VFS** : 파일처럼 읽고 쓸 수 있게 하는 기능
device driver는 H/W 로 부터 읽기 완성에 따른 인터럽트가 발생되기를 기다린다. User process를 wait 상태로 대기시키고 H/W가 인터럽트를 발생시키면, H/W에 연결된 인터럽트 핸들러가 동작이 된다. 인터럽트 핸들러가 동작이 되면서 H/W local buffer에 있던 내용을 user process나 경우에 따라서 커널 메모리에 적재한다. 그 다음에 wait 되어 있던 User process를 다시 ready 상태로 바꾼다.

**PIC칩에서 H/W 인터럽트 (timer, keyboard 등)를 총괄한다. PIC가 CPU에게 인터럽트를 건다.**
CPU는 인터럽트 테이블(**메모리 가장 위쪽에 있는 – idt_table**)을 확인해서 몇번 인터럽트인지 확인하고 번호에 해당하는 주소를 읽어온다 (ex. 0 : divide error, 1: debuf, 128 : system_call)
결과적으로는, **하드웨어 인터럽트를 호출하면 동일하게 idt_table 내의 번호에 해당하는 주소인 do_IRQ 함수의 주소가 있다**. (반면, **소프트웨어 인터럽트는 번호에 따라 다 다르다**)
do_IRQ 함수를 호출하면 irq_desc 배열이 있는데 거기에 번호에 해당하는 action을 취한다.

**Interrupt handler** : 인터럽트를 처리하기 위해 커널이 실행하는 함수
(인터럽트 서비스 루틴 이라고 하기도 함)
-CPU 마다 자기 나름대로는 인터럽트 스택, 인터럽트 핸들러가 각각 존재한다.
-HW device가 PIC를 통해 발생 (**PIC가 인터럽트 번호를 저장하고 있음**)
-Network device의 인터럽트 핸들러인 경우, 굉장히 많은 일을 해야함

하드웨어적으로 인터럽트를 빨리 처리해야 하는 경우는 Top halves에 넣고, 이걸로 미처 다 처리를 못한 부분은 bottom halves에 넣는다. **(bottom halves는 queue로 되어 있음)**
-즉, 보통 하드웨어 인터럽트가 걸리면 interrupt handler(Top halves)가 처리를 하고 커널 안에 있는 bottom halves를 등록을 시켜주고 빠져나가게 된다.
-또다른 인터럽트가 있으면 Top halves 에서 다시 처리를 하게 되고, 인터럽트가 없으면 **bottom halves에서 남은 인터럽트 잔처리를 한다음에 User process의 상태를 바꿔준다.**

Network device인 경우
-network device가 패킷 수신 : 인터럽트 발생
-top half : h/w에 확인신호 보냄. 패킷을 main memmory에 copy, device를 수신가능상태로 전환

-bottom half : 패킷을 protocol stack으로 이동

## Device Driver

GPIO를 인터럽트와 연결

//즉 GPIO 25번을 GPIOF_IN으로 쓰겠다고 등록을 시켜주는 거임

**int gpio_request_one(unsigned gpio, unsigned long flags, const char *label)**

->성공하면 return 0, 실패시 음수 리턴

여러 개의 GPIO에 대한 설정

**int gpio_request_array(const struct gpio *array, size_t num)**

=>여러 GPIO 관리위해 구조체의 배열 지원

```
struct gpio {
        unsigned gpio; // GPIO 번호
        unsigned long flags; // GPIO configuration flag
        const char *label;
}
```

ex.

```
static struct gpio buttons[] = {
        {25, GPIOF_IN, "BUTTON_1"},
        {26, GPIOF_IN, "BUTTON_2"},
};
```

배열 메모리 반환

**gpio_free_array(const struct gpio *array, size_t num)**

**int gpio_to_irq(unsigend gpio);** -> gpio : 하드웨어 gpio 핀 번호

**-성공시 return 값 : IRQ 번호** , 실패시 음수 리턴

***-gpio_request_one 이나 gpio_request_array를 등록해준 후에 gpio_to_irq를 실행해야 함***

-return된 irq 번호는 request_irq() 에서 사용가능

즉, 할당받은 irq 번호로 interrupt handler 등록

ex)

```
ret = gpio_to_irq(button[0].gpio);
if(ret < 0) error
button_irqs[0] = ret;
ret = request_irq(button_irqs[0], button_isr, IRQF_TRIGGER_RISING, "gpiomod#button1", NULL)
```

-인터럽트 핸들러 등록 : <linux/interrupt.h> 파일의 request_irq() 함수 이용

**int request_irq(unsigned int irq, irq_handler_t handler,unsigned long flag, const char *name, void *dev)**

-인터럽트 핸들러는 const char *name 이라는 이름을 가지고 있음

-**dev** : 디바이스에 대해 여러개의 인터럽트 핸들러를 등록시켰을 때 구분하기 위해 사용, 1개라면 null

**정리하면,**

**1) gpio_request_one , gpio_request_array : 특정 gpio 번호를 등록**

**2) re = gpio_to_irq(gpio) : gpio 번호에 대한 irq 번호를 리턴 받음**

**3) request_irq(irq , 핸들러, RISING.....) : irq로 인터럽트 핸들러 등록**

**4) gpio_free_array() : gpio 배열 정리**

밑에 더 있음

**Blocking I/O**

-Device driver의 read() 함수는 H/W 에게 명령을 요청하고 그 결과를 기다린다

-요청을 보낸 뒤 TASK_INTERRUPTIBLE 상태가 되고 해당 wait queue에 대기한다

(wait queue : device driver 마다 관리한다)

-H/W가 요청을 완료하면 인터럽트를 호출하고 이 인터럽트 핸들러에서 wait queue에 있는 task를 깨운다

**관련 함수**

1) **DECLARE_WAIT_QUEUE_HEAD(queue) : queue를 wait queue로 사용하기 위해 선언하고 초기화**

2**) wait_event_interruptible(queue, condition)** -> <linux/wait.h>

-wait_queue_head_t queue : 여기에 이 task가 저장된다

-condition : boolean 표현식으로 true가 되면 깨어남

(Ctrl+C 와 같은 외부 signal로부터 깨어날 수 있도록 TASK_INTERRUPTIBLE 상태로 만듬)

3) **void wake_up_interruptible(wait_queue_head_t *queue)**

-잠자는 queue의 모든 task를 깨움

예제 코드

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_write( struct file *filp, const char __user *buf, size_t count, left_t *poa) {
        flag = 1;
        wake_up_interruptible(&wq);
        return count;
}
ssize_t sleepy_read( struct file *filp, char __user *buf, size_t count, left_t *poa) {
        wait_event_interruptible(wq, flag != 0); // flag가 1이면 깨어남
        flag = 0;      // 먼저 깨어난 놈이 나머지 깨어난 놈을 다시 flag = 0으로 설정해서 재움
        return 0;
}
```

**request_irq() 함수는 sleep될 수 있기 때문에** 코드 실행이 중단될 수 있는 상황에서는 부르면 안됨

왜냐하면

1) **interrupt handler 등록**

2) **해당 항목이 /proc/irq 에 만들어짐**

3) **proc_mkdir() 함수가 새로운 procfs 항목을 만듬**

-proc_create() 함수를 호출해 procfs 항목을 설정

-kmalloc() 함수를 호출해 메모리를 할당 => kmalloc()은 sleep 될 수 있음

**free_irq( unsigned int irq, void *dev)** : 인터럽트 핸들러 삭제

인터럽트 핸들러 작성

인터럽트 핸들러 선언

static irqreturn_t intr_handler( int irq, void *dev)

-irq : 핸들러가 처리할 인터럽트 번호

-dev : device 구분

-return value (irqreturn_t 를 사용)

1) IRQ_HANDLED : device가 핸들러와 재대로 구성될 때
2) IRQ_NONE : handler가 device와 맞지 않을 때

**

**Device driver의 module_init 부분에서** request_irq()를 통해 **인터럽트 핸들러를 등록하고**, free_ireq()로 해지시켜 준다
-인터럽트 핸들러가 sleep이 될 수 있기 때문에 인터럽트 핸들러에서 등록을 시켜주지 않고 디바이스 드라이버에서 등록을 시켜준다.
**

**reentrancy** : 자기가 수행될 때 똑같은 함수가 수행 될 수 있으니 조심해서 코드를 짜라는 의미
그러나, **인터럽트 핸들러인 경우 자기가 수행될 때 이 핸들러가 겹쳐서 수행되는 경우는 없다**
(**수행 될 때 해당 irq가 mask가 되기 때문에**)

ㅋ

인터럽트 핸들러(Top halves)와 bottom halves(soft_irq / tasklet / work queue)가 있다 배웠다.
인터럽트 핸들러 그 자체에서는 reentrancy를 신경쓸 필요가 없어서 공유변수 관리를 안해도 되지만, **듀얼코어(cpu가 2개)인 경우 cpu 0번의 Top halves가 가져온 데이터를 어떤 자료구조에 저장해** cpu 0번의 bottom halves가 처리하도록 할 수 있다. 그러나, **cpu1번의 bottom halves도 해당 자료구조에 접근할 수가 있으므로 이런 부분에서는 공유변수에 lock을 건다**.

cpu마다 interrupt context에 해당하는 interrupt stack이 있다.

**unsigned int do_IRQ( struct pt_regs regs)**
-이 함수가 불리기 직전에 레지스터에는 유저 프로세스가 실행한 내용들이 있다. 해당 레지스터 값들을 커널스택에 저장한다. => **regs에 저장한 register 값과 interrupt 값이 있음**
-해당 IRQ 번호에 해당하는 하드웨어를 전부 mask 시킨다.
-해당 인터럽트 핸들러가 없으면 그냥 리턴된다.
-해당 인터럽트 핸들러가 있으면 아래 코드를 실행

```
irqreturn_t handle_IRQ_event( unsigned int irq, struct irqaction *action) {
        irqreturn_t ret, retval = IRQ_NONE;
```

```
        unsigned int status = 0;
        if(! (action->flags & IRQF_DISABLED)) {
                local_irq_enable_in_hardirq();
        }
        do {
                trace_irq_handler_entry(irq, action);
                ret = action->handler(irq, action->dev_id);
                switch(ret) {
                        .......
                }
        }
}
```

**//action : 특정 인터럽트 핸들러에 연결리스트 형태로 엮여있는 일종의 action**

**ret_from_intr() 함수 호출** : need_resched가 set 되어 있는가 확인

(action을 수행하다 보면 현재 수행하는 프로세스를 쫓아내야 되는 경우 – time slice 소진 ..등 확인)

-schedule() 함수 호출

/proc/interrupts

**-cat /proc/interrupts : 인터럽트 번호에 해당하는 irq 리스트 출력됨**

1) local_irq_disable()
2) local_irq_enable()
3) local_irq_save()
4) local_irq_restore()
5) void disable_irq(unsigned int irq) : **특정 interrupt irq line에 대해서 disable 시킴**

- 실행중인 인터럽트 핸들러가 종료될 때까지 기다림. 그 후에 인터럽트를 disable 시킴

6) void enable_irq(unsigned int irq)

Question

1) Multi-processor system에서 인터럽트가 발생하면 어느 CPU에 있는 프로세서가 그 인터럽트를 처리하게 되는가?

=>PIC가 결정하게 되는데, 일반적으로 그 시점에서 인터럽트 처리를 안하는 CPU에게 할당된다

만약, 두 개의 CPU가 모두 처리를 안하고 있으면 CPU0번이 처리하게 된다.

2) Multi-processor system에서 CPU-0에서 수행 중인 kernel code가 한 interrupt line을 disable 시키면 다른 CPU에서는 그 interrupt line으로부터 interrupt를 받을 수 있는가?
=> X

**irqs_disabled()** : 현재 프로세서의 interrupt가 disable 되어 있는지(0이 아닌값 리턴) 확인 macro
**in_interrupt()** : interrupt handler(top half)나 bottom half를 처리하고 있는 경우 0이 아닌 값 return
**in_irq()** : interrupt handler(top half)를 처리하고 있는 경우 0이 아닌 값 리턴

bottom half vs top half
1) top half : H/W 에게 인터럽트가 수신되었다는 것, data를 메모리로 copy 하는 수준만 처리
2) bottom half : 나머지 전부

**bottom half : softirq, tasklet은 sleep을 허용하지 않음**
1) **softirq : 병렬적으로 수행 될 수 있는 정적인 배열 형태=>Network, block device에서 사용**

```
struct softirq_action {
        void (*action)(struct softirq_action *);
};
```

즉, -------softirq_action------
*(action) -> 어떤 함수의 주소를 가지고 있음
---------- my_softirq->action(my_softirq)로 handler 부름
.....
---

my_softirq : softirq_vec[]의 한 element를 가리키는 pointer
=>등록된 softirq는 hardware interrupt code가 반환되는 경우(H/W 인터럽트 종료) 실행됨
=>**ksoftirqd kernel thread** 에서 실행되는데 ksoftirq kernel thread는 시스템이 부팅될 때 실행되었다가 인터럽트 핸들러(top half)가 끝날때마다 ksoftirqd가 active 되어서 수행한다.
(ksoftirqd 데몬은 softirq, tasklet 을 차례대로 수행한다)
=>실질적으로 ksoftirq 안에 do_softirq() 에서 실행된다

(각 softirq에 해당하는 핸들러를 차례대로 호출한다)
=>do_softirq() 안에는 __do_softirq()가 있다.

__do_softirq() 코드

```
u32 pending;
pending = local_softirq_pending();          //softirq_action에서 해야할 작업이 있으면 true 리턴
if(pending) {
        struct softirq_action *h;
        set_softirq_pending(0);       // 내가 처리할테니 다른 cpu는 꺼져라 의미
        h = softirq_vec;              // softirq_action의 시작주소 : softirq_vec
        do {
                if( pending & 1) {      // softirq_action의 각 항목에 값이 있으면 차례대로 수행
                        h->action(h);      // 등록된 코드가 수행된다
                }
                h++;
                pending >>=1;
        } while(pending);
}
```

-해야할 작업은 1, 그렇지 않으면 0 으로 구성된 pending 배열이 0이 되면 while문을 빠져나감

soft_irq 에서

| **Tasklet** | **Priority** | **Softirq Description** |
|---|---|---|
| HI_SOFTIRQ | 0 | High-priority tasklets |
| TIMER_SOFTIRQ | 1 | Timers |
| NET_TX_SOFTIRQ | 2 | send network packets |
| NET_RX_SOFTIRQ | 3 | receive network packets |
| ..... | | |

softirq는 선언(등록), 활성화, run 상태로 나뉘어져 있는데 ksoftirqd가 실행하는 단계는 run 상태다

---

1) **선언** : **open_softirq( NET_TX_SOFTIRQ, net_tx_action)** 와 같이 선언한다
여기서 NET_TX_SOFTIRQ는 전역변수로 선언된다. 왜냐하면, softirq 자체가 top half(인터럽트 핸들러)가 미뤄놓은 일을 하는 것이기 때문이다. 해당 부분은 critical section이기 때문에 lock을 사용한다.

2) **활성화** : **raise_softirq( NET_TX_SOFTIRQ);**
=> 인터럽트 핸들러가 해당 함수를 실행한다

----------여기까지 Device Driver의 code 작업

3) **run** : **ksoftirq 에서 실질적으로 수행한다**.

2) **tasklet => softirq 보단 중요성이 낮음**

```
struct tasklet_struct {
        struct tasklet_struct *next;
        unsigned long state; // 0: 끝난 상태, TASKLET_STATE_SCHED(실행기다림),RUN(실행중)
        atomic_t count; // 0 이 아니라면 중요도가 높다는 의미(건드릴 수 없는 상태임)
        void (*func)(unsigned long);
        unsigned long data;
};
```

Scheduling tasklet => raising softirq 와 같은 개념

-tasklet_schedule() => __tasklet_schedule() 호출

tasklet_action() : 실제로 tasklet 연결리스트를 처리

**ksoftirq는 일종의 커널 task로 돌면서 먼저 softirq에 처리할게 있으면 처리하고, 그 다음에 tasklet(hi, tasklet), work queue 순으로 이동하면서 처리한다**

tasklet 사용

---

1) 선언 : 정적으로 = **DECLARE_TASKLET(name, func, data)** // **data는 func의 parameter로 사용**

ex) DECLARE_TASKLET( my_tasklet, my_tasklet_handler, dev);

동적으로 = tasklet_init(t, tasklet_handler, dev); // t : tasklet struct의 pointer

tasklet 핸들러 작성 : void tasklet_handler( unsigned long data);

-semaphore나 blocking function을 사용하면 안댐 ( tasklet은 sleep될 수 없음)

2) 활성화 : tasklet_schedule(&my_tasklet)

3) run : ksoftirq

tasklet 에는 중요도에 따라 2가지로 나뉘어져 있다.

-**HI : tasklet_hi_vec 구조체**

-**TASKLET : tasklet_vec 구조체**

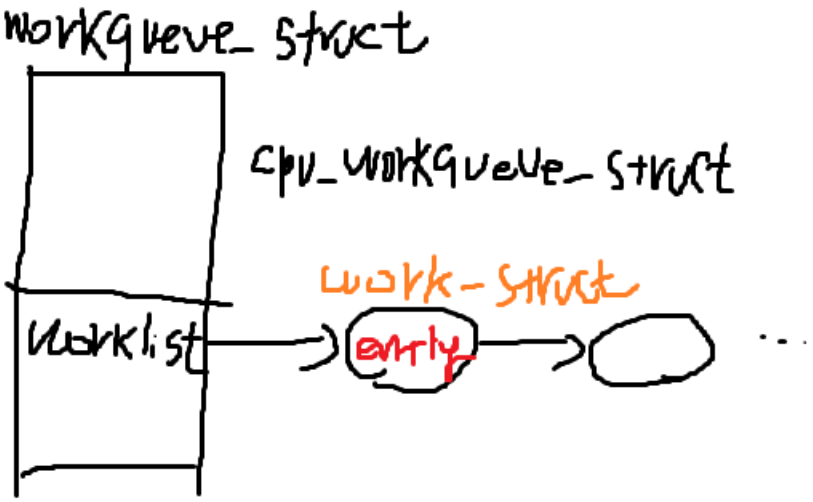
3) work queue : softirq, tasklet 는 interrupt context인 반면 work queue는 process context 이다

그러나, work queue는 system call 과는 다르기 때문에 kernel stack을 사용할 수가 없고 device driver 를 등록할 때(insmod) 스택을 할당받아서 사용
-scheduling이 가능해서 sleep 되는데 제약이 없다
(다량의 메모리 할당, semaphore 할당, block i/o 가능)
-프로세스를 수행하다 보면 백그라운드로 실행해야 할 것이 많은데 그런것들을 work queue에서 수행
-softirq, tasklet 의 ksoftirqd 를 사용하지 않고 별도의 worker thread 인 events 라는 커널스레드 사용

**struct workqueue_struct *create_workqueue( const char *name); : 새로운 work queue 만들기**
-name : kernel thread의 이름 (events)
ex) struct workqueue_struct *keventd_wq;
keventd_wq = create_workqueue("events");

```
struct workqueue_struct {
        struct cpu_workqueue_struct cpu_wq[NR_CPUS];
        struct list_head list;
        const char *name;
        int singlethread;
        int freezeable;
        int rt;
}
struct cpu_workqueue_struct {
        spinlock_t lock;
        struct list_head worklist;          //수행해야 하는 작업들 링크드리스트로 관리
        struct work_struct *current_struct;   // worklist에 연결되어 있는 노드
        struct workqueue_struct *wq;     //자기 자신을 가리킴

}
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;              // 하나의 작업 객체로써 수행시키고 싶은 함수의 포인터
}
```




work_thread() 함수의 주요 코드 (events)

```
for ( ; ; ) {
        prepare_to_wait( &cwq->more_work, &wait, TASK_INTERRUPTIBLE);
        if( list_empty(&cwq->worklist)) schedule();
        finish_wait( &ceq->more_work, &wait);
        run_workqueue(cwq);
}
```

run_workqueue() 를 들어가보면

```
while(!list_empty(&cwq->worklist)) {
        struct work_struct *work;
        work_func_t f;
        vodi *data;
        work = list_entry(cwq->worklist.next, struct work_struct, entry);
        //worklist.next는 맨처음 work_struct
        f = work->func;
        list_del_init(cwq->worklist.next);
        ...
}
```

work queue 사용

---

1) **work 생성** 정적으로 = DECLARE_WORK(name, void(*func)(void *), void *data)

포인터로 = INTT_WORK( struct work_struct *work, void(*func)(void*), void *data)

work queue handler : void work_handler( void *data)

-work thread가 실행

-process context 에서 실행

**2) scheduling work**

-기본 events work thread를 가진 work handler 함수를 work queue에 추가

ex) schedule_work( &work);

-일정 시간이 지난 다음에 실행시키려는 경우

ex) schedule_delayed_work( &work, delay)

**3) void flush_scheduled_work(void);**

-해당 queue의 모든 작업들이 완료될 때까지 sleep 상태에 있다가 반환됨

-디바이스 드라이버에서 작업이 끝나서 work queue에 있는 작업들을 없애버려야 할 때

-int cancel_delayed_work( struct work_struct *work)

(scheduling work에서 일정 시간이 지난 다음에 실행시키려는 경우, 바로 실행 취소)

**위의 내용은 기본적인 커널 드라이버에서 사용하는거고 사용자가 새로운 work queue를 사용하고자 할 때 아래의 과정을 따른다**

1) 새로운 work queue 만들기 : struct workqueue_struct *create_workqueue(const char *name)

2) DECLARE work ~ 를 통해서 work 생성

3) int queue_work( struct workqueue_struct *wq, struct work_struct *work); 를 통해 work queue 등록

4) flush_workqueue(struct workqueue_struct *wq) : 종료할 때

커널 동기화 방법

**1) atmoic**

```
typedef struct {
        volatile int counter;
} atomic_t;

atomic_t v;        //define v
atomic_set( &v, 4);   // v = 4
atomic_add( 2, &v);   // v = v + 2 = 6;
atomic_inc(&v); // v = v + 1 = 7;
printf("%d₩n", atomic_read(&v)); // 7
```

int atomic_dec_and_test(atomic_t *v); // 값을 1 감소시키고 그 결과가 0이면 true 반환

**atomic** vs **race condition**

=>get, increment, store을 한 cycle에 해결 일반적으로 3부분으로 나누어 진행함

ex. Get,increment,store i (7->8) get i (7)-> increment i(7->8) -> write back i (8)

**Atomic bitwise operations**

-define ine <asm/bitops.h>

unsigned long word = 0;

set_bit( 0 , &word) // 0번째 비트를 1로 세팅

set_bit(1, &word); // 1번째 비트를 1로 세팅

printf("%u₩n", word); // 3

clear_bit(1, &word); // 1번째 비트를 0으로

change_bit(0, &word); // 0번째 비트를 change -> 원래 1이었으면 0, 0이었으면 1로

if( test_and_set_bit(0, &word)) { // 0번째 비트의 값을 리턴

}

spin lock : 인터럽트 핸들러와 같이 sleep을 하면 안되는 코드에 쓴다

-<asm/spinlock.h> or <linux/spinlock.h>

**1) interrupt handler 라면 아래와 같이 사용한다.**

**DEFINE_SPINLOCK(mr_lock);** // spinlock 변수 하나 선언

**spin_lock(&mr_lock);**

/* critical section */

**spin_unlock(&mr_lock);**

**2) bottom half 라면 아래와 같이 사용한다**. => interrupt를 disable 시키고 lock을 얻는다

DEFINE_SPINLOCK(mr_lock);

unsigned long flags;

spin_lock_irqsave(&mr_lock, flags); =>interrupt 상태를 모를 때, save, restore을 사용한다.

/* critical section */

spin_unlock_irqrestore(&mr_lock, flags);

interrupt enable 상태를 미리 알고 있는 경우라면, 위의 save, restore 코드와 역할을 동일함
DEFINE_SPINLOCK(mr_lock); 단지, interrupt의 상태를 알 때는 해당 코드를 써도 됨
spin_lock_irq(&mr_lock);
...
spin_unlock_irq(&mr_lock);

수 많은 spin lock 부분에서 어떤 곳에서 문제가 생겼는지 알고 싶다면, kernel configuration option 에서 CONFIG_DEBUG_SPINLCOK을 활성화 시키고 다시 디바이스 드라이버를 컴파일하면 오류메세지 나옴

그 외,
-spin_lock_init() : spin lock 초기화
-spin_lock_bh() : bottom half 를 수행시키는 ksoftirqd 를 수행못하도록 lock을 거는 것.
-
tasklet 까지는 spin lock (busy waiting) 밖에 못쓴다. 왜냐하면 sleep을 할 수가 없으니까

Reader-write semaphore
-<linux/rwsem.h>에 정의됨
1) 선언 : static DECLARE RWSEM(name);
동적 초기화 : init_rwsem( struct rw_semaphore *sem);

lock 을 거는 이유
-critical section 을 수행할 때 preempt 당하는 걸 방지
-multi-processor 환경이기 때문에

**disable interrupt** : **인터럽트를 처리 하지 않는다.**
원래 interrupt 를 disable 시켜줄라는 큰 이유는, preemption을 막기 위해서 이다.
(timer 를 다 쓴 경우, disk i/o 를 완료한 프로세스로부터 다른 프로세스가 쫓겨나는 경우 ....)
=>타이머가 늦어질 수도 있고, 네트워크 패킷도 손실할 수 있는 역효과가 난다.

**disable preemption** : **인터럽트를 처리는 하되, preempt 되지는 않는다**.
=>이것 같은 경우는 disable interrupt와 크게 다른점이 인터럽트 처리는 하고 만약 본인이 enable_preemption 코드를 실행한다면 쫓겨나게 된다.
**실제 함수** : disable preemption : preempt_disable()
enable preemption : preempt_enable()
preempt_enable_no_resched() : enable 하더라도 어느 정도 실행하고 scheduled 당함

커널의 시간
1) system timer : 보통 PIC 가 system timer 역할을 한다.
2) Tick
3) wall time : 현재 시각, 사용자 프로그램 입장에서 필요
4) system uptime : booting된 뒤로 경과한 시간
=>딱 부팅 했을 때 booting time 은 real time clock(RTC)로 부터 얻어온다.
즉, **wall time(현재 시간) = booting time(real time clock) + system uptime(booting 후 경과시간)**

system uptime 은 전역변수로 jiffies 가 있다.
=>0부터 시작하며, timer interrupt가 발생할 때마다 1씩 증가한다.
=>extern unsigned long volatile jiffies; 라고 <linux/jiffies.h>에 선언되어 있다.

RTC( real – time clock) : 시스템 시간을 저장하는 non-volatile device
-Kernel은 시스템 시작 시 RTC를 읽음

**Timer interrupt handler**
두 부분으로 구성된다
1) architecture dependent
2) architecture independent
**=>handler의 처리내용** : **xtime_lock을 요청 (xtime 안에 wall time 이 있음)**
architecture dependent 함수인 **tick_periodic() 함수 호출**
**tick_periodic()**
-jiffies_64를 1 증가
-현재 실행중인 프로세스가 소모한 system time과 user time과 같은 resource time 통계 갱신
(cpu 과금 서비스인 AWS 에서 주로 사용)
-xtime에 저장된 현재 시간(wall time) 갱신

-sleep에 들어간 프로세스를 깨워주는 역할을 하는 코드가 있다.

**struct timespec xtime;**

**struct timespec** {

kernel time_t **tv_sec**; // 초

long **tv_nsec**; // 나노초

}

=>tv_sec 에는 1970년 1월 1일 이후 몇 초가 지났는지를 저장

**struct timer_list** {

struct list_head entry;

**unsigned long expires; // 깨어날 시간**

**void (*function)(unsigned long); // 깨어나고 나서 수행될 함수**

unsigned long data;

}

**1) Timer 생성, 초기화, 및 값 할당**

struct timer_list my_timer;

**init_timer(&my_timer);** // 초기값으로 clear 됨

my_timer.expires = jiffies + delay; // 깨어날 시간 = 현재 jiffies값 + delay 값

my_timer.data = 0;

my_timer.function = my_function; // 깨어나고 나서 수행될 함수

**2) Timer 활성화**

add_timer(&my_timer);

**Timer의 delay 값을 뒤 늦게 수정 하려면 (즉, mod_timer는 add_timer + 시간 수정 기능)**

mod_timer( &my_timer, jiffies + new_delay); 로 깨어날 시간의 값을 수정한다.

1) add_timer 로 활성화 되었는데 mod_timer를 사용하면, 시간이 수정된다

2) add_timer 로 활성화 되지 않았는데 mod_timer로 수정하면, 수정하고 큐에 등록된다.(활성화)

**(그러나, 이미 큐 안에 들어간 상태라 에러가 발생하기 쉬움)**

**따라서,**

-del_timer( &my_timer) 로 우선 비활성화 시키고, // 아니면 del_timer_sync( &my_timer);

-다시 수정한 다음에 add_timer( &my_timer)로 Timer 활성화 한다.

**커널 내부에서 Timer 구현**

-Timer handler 는 softirq 처럼 timer interrupt가 끝난 후 bottom half context에서 수행
(***softirq_vec 에서 0번 인덱스에 timer 수행 핸들러가 있다 => 제일 우선순위 높음***)

-Timer interrupt handler 는 update_process_times() 호출
=>run_local_timers() 호출

```
void run_local_timers() {
        hrtimer_run_queues();
        raise_softirq( TIMER_SOFTIRQ);          // Timer 핸들러 등록
        softlockup_tick();
}
```

-등록된 Timer 들은 5개의 group 으로 묶어 관리한다.

```
struct page {       // physical page 관리를 위한 정보
        unsigned long flags;   // page 변경 여부, lock 여부
        atomic_t count;        // page 참조횟수 (library page에서 주로 count가 많이 증가)
        ..
        void *virtual;
}
```

여기서 말하는 struct page는 커널에서 사용하는 page이다
유저 페이지는 task_struct 의 mm 구조체에서 따로 관리한다.

**Zone : 비슷한 속성을 가진 page 들의 집합** , 3 ~ 4개의 memory zone이 있음
**1) ZONE_DMA** : DMA 수행 관련 < 16MB 영역
(디바이스의 로컬 버퍼에 있는 내용을 메모리로 카피 / 메모리의 내용을 로컬 버퍼로 카피)

**2) ZONE_NORMAL** : 일반적인 Kernel address space 16 ~ 896MB 영역

(일반적으로 메모리를 할당받는 공간)

**3) ZONE_HIGHMEM** : high memory > 896MB 영역

**struct zone** {

unsigned long **watermark**[NR_WMARK]; // 3단계로 나눈다( 충분, 보통, 부족)

(커널이 메모리 할당을 하면 남은 메모리 정보를 충분,보통,부족으로 나누어 그에맞게행동)

.....

spinlock_t lru lock; //

spinlock_t lock; // zone의 영역에 대해 lock

const char ***name**; // ZONE_DMA , ZONE_NORMAL , ZONE_HIGHMEM

}

사용자 영역에서 메모리를 할당 하는 것은 free를 시켜주지 않아도 커널이 알아서 해주는데, 커널쪽의 메모리를 다룰 때에는 반드시 메모리를 free 시켜주어야 한다

커널의 메모리

**1) 저수준** : 리눅스 커널에서는 **기본적으로 page 단위로 관리를 하는데, page 단위로 할당**

**-메모리 획득 함수**

**struct page *alloc_pages( gfp_t gfp_mask, unsigned int order);**

**mask** : allocation 받을 때 sleep 해도 괜찮은지 등을 세팅하는 bit

**order** : **2^order 개의 연속된 physical page 할당받음 (연속된 page가 없으면 오류남)**

return : 성공 시 첫번째 page의 struct page에 대한 포인터를 리턴받음 , 오류 시 NULL

ex) alloc_pages( GFP_NORMAL, 0);

=>1개의 page를 할당 받음

=>연속된 page를 할당받는 것은 어렵기 때문에 DMA의 경우에 주로 적용됨

**-Physical page의 logical address를 얻을 때**

**void * page_address( struct page *page);**

**앞의 두 개의 함수를 합친 것 (메모리 획득 + logical address 얻는 함수)**

**unsigned long __get_free_pages( gfp_t gfp_mask, unsigned int order);**

=>unsigned long으로 리턴되기 때문에 void *page로 타입 캐스팅을 해서 사용해야 한다.

**page 반환 함수**

void __free_pages( struct page *page, unsigned int order);  // logical로 변환 안했을 때 사용
void free_pages( unsigned long addr, unsigned int order);  // logical로 변환 했을 때 사용
void free_page( unsigned long addr)

**2) 고수준 : byte 단위로 관리, 할당**
=> 함수 내부에서 저수준 할당 방식을 사용한다.
void *kmalloc( size_t size, gfp_t flags);

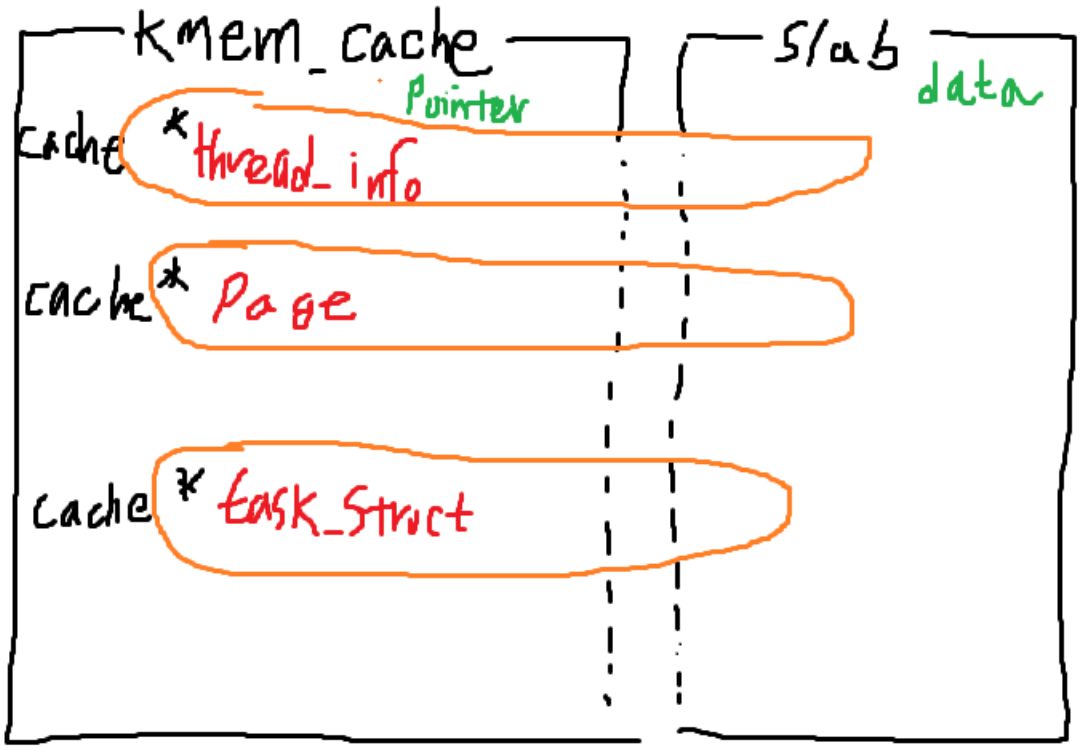
ex)
struct dog *p;
p = kmalloc( sizeof(struct dog), GFP_KERNEL);
// GFP_KERNEL : 진행 도중 sleep 가능
    GFP_ATOMIC : 진행 도중 sleep 안됨 (요청이 실패할 경우가 높다)

**메모리 해제**
**void free( const void *ptr);**

**물리적으로 연속될 필요가 없는, 가상적으로 연속된 메모리 공간 할당 (큰 영역의 memory 할당)**
**void *vmalloc( unsigned long size);  // GFP 플래그가 없어서 항상 sleep 가능**

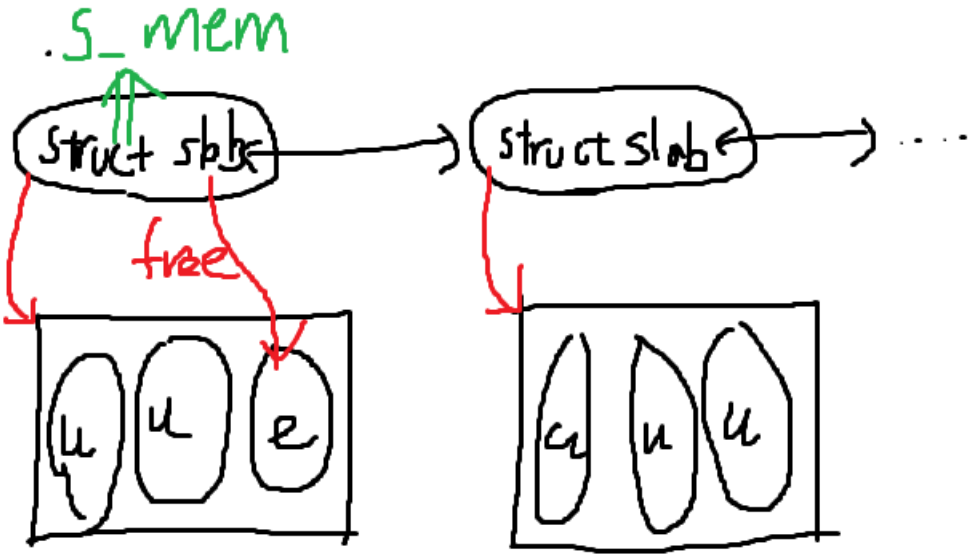
할당 해제
**void vfree(const void *addr);**
즉, kmalloc은 물리적으로 연속된 적은 메모리를 할당 받기 때문에 디바이스 드라이버와 직접적으로 DMA로 주고 받을 때 사용.

Slab layer
-object cache : 특정 object에 대한 free list




```
struct slab {
        struct list_head list;     // struct slab 끼리 서로 연결(full, partial, empty list)
        void *s_mem;   // first object in the slab
        ....
        kmcm_bufctl_t free;       // first free obejct
}
```

=>struct slab 은




-kmem_getpages() 함수 : cache가 사용할 새로운 slab을 __get_free_pages()를 통해 얻어옴

```
static inline void * kmem_getpages( struct kmem_cache *cachep, gfp_t flags) {
        void *addr;
        flags |= cachep->gfpflags;
        addr = (void *)__get_free_pages(flags, cachep->gfporder);
        return addr;
}
```

Kernel code 에서 slab을 이용하기 위한 함수들

**1) 새로운 cache 생성**

```
struct kmem_cache *kmem_cache_create(
                                const char *name,
                                size_t   size,
                                size_t align,
                                unsigned long flags,
                                void (*ctor)(void *) );
```

flag parameter

-SLAB_PANIC : 실패 시 PANIC 발생

-SLAB_CACHE_DMA : DMA_ZONE에 slab 할당

성공 시 pointer, 실패 시 NULL 반환

Cache 제거

**int kmem_cache_destory( struct kmem_cache *cachep);**

Cache 에서 객체를 할당 받음

**void * kmem_cache_alloc( struct kmem_cache *cachep, gfp_t flags);**

// kmem_cache_create로 만들어진 cache 에서 객체를 하나 할당 받음

사용이 끝난 객체를 cache에 반납

**void kmem_cache_free( struct kmem_cache *cachep, void *objp);**

**Slab allocator 예제**

```
struct kmem_cache *task_struct_cachep;
int err;
task_struct_cachep = kmem_cache_create( "task_struct",
                                        sizeof( struct_task_struct),
                                        ARCH_MIN_TASKALIGN,
                                        SLAB_PANIC | SLAB_NOTRACK,
                                        NULL);
err = kmem_cache_destroy(task_struct_cachep);

....
struct task_struct *tsk;
tsk = kmem_cache_alloc( task_struct_cachep, GFP_KERNEL);
if( !tsk) return NULL;
kmem_cache_free(task_struct_cachep, tsk);
```

High memory mapping

-x86 architecture에서 896MB 너머의 Physical memory에 있는 page 는 kernel space와 자동으로 연결되지 않는다.

-High memory 가 아닌 영역에서 page 를 할당받아서 아래 함수를 사용

```
void kmap( struct page *page);                  // sleep 될 수 있음

unmapping
void kunmap( struct page *page);
```

sleep 하지 말아야 하는 context (softirq, tasklet...) 에서 high memory mapping이 필요한 경우
-디바이스 드라이버의 module_init() 부분에서 kmap, 드라이버 부분에서 kmap_atomic 사용
(미리 예약해둔 mapping을 할당)
void *kmap_atomic( struct page *page, enum km_type type);
=>인터럽트는 걸릴 수 있지만 자동으로 preempt disable 된다

void kunmap_atomic( void *kvaddr, enum km_type type);
=>disable 된 preemption이 다시 enable 됨

정리
1) kmalloc() : 하드웨어와 관련되고, 물리적으로 연속된 page가 필요한 경우
2) alloc_pages() : kmalloc() 내부에서 쓰이는 함수
3) kmap() : x86 에서 high memory 접근이 필요한 경우
4) vmalloc() : 논리적으로만 연속되는 memory 가 필요한 경우
5) slab : 큰 자료구조를 다량으로 생성하고 해제하는 경우

<악성코드 분석 – 보안 세상>

함수 호출 규약
1) cdecl : caller가 처리 (C언어)
-ex. ADD ESP, 8

2) stdcall : callee가 처리 (win32 api)
-ex. RETN 8 (RETN 후, 8바이트 만큼 ESP를 증가시킨다)
-전체 코드 크기를 줄일 수가 있다

3) fastcall : 전체적으로는 stdcall 방식이나, 함수 parameter 전달 시, 파라미터 2개 까지 레지스터를 통해 전달할 수가 있다 (ECX, EDX 사용)

<PE 파일>
분류
1) 32 비트 형태의 실행 파일 : PE 또는 PE32
2) 64 비트 형태의 실행 파일 : PE+ 또는 PE32+

| | 종류 | 주요 확장자 |
|---|---|---|
| 1) | 실행 계열 | EXE , SCR( 화면 보호기 확장자) |
| 2) | 라이브러리 계열 | DLL, OCX, CPL, DRV |
| 3) | 드라이버 계열 | SYS, VXD |
| 4) | 오브젝트 파일 계열 | OBJ |

VA : 프로세스 가상 메모리의 절대 주소
RVA(Relative VA) : 기준 위치로 부터의 상대 주소
(PE 헤더 내의 정보는 RVA 형태가 많음)

**PE 헤더**

**1) DOS Header**

-DOS 파일에 대한 호환성을 고려하여 만든 헤더

```
typedef struct_IMAGE_DOS_HEADER {
        WORD e_magic;      // DOS signature (4D5A => ASCII 값 : MZ )
        WORD e_cblp;
        ...
        LONG e_lfanew;     // NT Header의 옵셋(위치)를 표시
        // E0 00 00 00 이면 000000E0 부터 NT Header가 시작된다.
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER
```

**2) DOS Stub**

-DOS Header 다음에 위치한다

-존재 할 수도 있고 존재하지 않을 수도 있어서(크기도 가변적) DOS Header의 e_lfanew 를 통해 NT HEADER의 위치를 표시한다.

-코드와 데이터가 함께 구성 된다.

-Window 32비트 운영체제에서는 이 부분의 코드가 실행되지 않는다(PE 파일로 인식하기 때문에)

-DOS 환경이나 DOS용 디버거를 이용하여 실행하면, 이 코드 부분이 실행된다

**3) NT Header : PE 파일의 실질적인 시작**

```
typedef struct _IMAGE_NT_HEADERS {
        DWORD         Signature;          // #define 으로 지정됨 (PE 파일이기 때문에)
        IMAGE_FILE_HEADER   FileHeader;
        IMAGE_OPTIONAL_HEADER32 OptionalHeader;
}IMAGE_NT_HEADER32, *PIMAGE_NT_HEADERS32;
```

필드 중

IMAGE_FILE_HEADER  FileHeader

-파일의 개략적인 속성을 표시

```
typedef struct _IMAGE_FILE_HEADER {    // 20바이트
        WORD          Machine;
        WORD          NumberOfSections;
        DWORD         TimeDataStamp;
```

```
    ...
    WORD        SizeOfOptionalHeader;
    WORD        Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

**주요 필드**

**1) Machine : CPU 별 고유 값 =>**

**(#define IMAGE_FILE_MACHINE_ARM  0x01c0 ) 이렇게 지정되어 있음**

2) NumberOfSections : PE 파일에 저장된 세션의 수

3) SizeOfOptionalHeader : IMAGE_NT_HEADERS32 구조체의 마지막 filed 인 IMAGE_OPTIONAL_HEADER32의 크기

4) Characteristics : 파일의 속성을 표시

(실행 파일 ? DLL 파일 ? bit OR 형식으로 제공)

```
typedef struct_IMAGE_OPTIONAL_HEADER {
    WORD        Magic;
    DWORD       AddressOfEntryPoint;
    DWORD       BaseOfCode;     // Code 섹션의 가장 처음 시작 주소
    DWORD       BaseOfData;     // Data 섹션의 가장 처음 시작 주소
    DWORD       ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    DWORD       SizeOfImage;
    DWORD       SizeofHeaders;
    WORD        Subsystem;
    DWORD       NumberOfRvaAndSizes;
} IMAGE_OPTIONAL_HEADER32 ~
```

**1) Magic**

-IMAGE_OPTIONAL_HEADER32 : 010B

-IMAGE_OPTIONAL_HEADER64 : 020B

**2) AddressOfEntryPoint**

-EP의 RVA 값

-프로그램에서 최초로 실행되는 코드의 시작 주소 값

**3) ImageBase**

-32비트의 경우, 프로그램의 가상메모리는 주소 범위 : 0 ~ FFFFFFFF
-모듈을 access 할 때 기준이 되는 주소(기준점)
-가상 메모리에서 PE 파일이 로딩되는 시작 주소
( EXE, DLL 파일 : 0 ~ 7FFFFFFF => User memory 영역, SYS 파일 : 80000000 ~ FFFFFFFF =>Kernel memory 영역)
-PE 로더는 PE 파일을 실행하기 위해 프로세스를 생성하고, 파일을 로딩한 후 EIP = ImageBase + AddressOfEntryPoint 로 VA 설정한다

**4) SectionAlignment** : 메모리에 저장된 섹션의 최소 단위
**5) FileAlignment** : 파일에서 저장된 섹션의 최소 단위
=> 파일/메모리에 저장된 PE 파일의 Body를 구성하는 섹션의 크기는 각각 이 두 값의 배수가 되어야 한다.

6) SizeOfImage : PE 파일이 메모리에 로딩 될 때 가상 메모리에서 PE Image의 크기

7) SizeOfHeaders : PE 헤더의 전체 크기
(파일 시작에서 SizeOfHeaders 옵셋만큼 떨어진 위치에 첫 섹션이 존재 – FileAlignment의 배수)

**8) Subsystem** : *.sys 파일인지 또는 일반 실행 파일(*.exe, *.dll)인지를 구분한다

## 4) 섹션 헤더

-각 섹션의 속성을 정의
-섹션으로 구분된 데이터 (code, data, resource, etc) 관리
-섹션 헤더의 개수 : NT_Header.FileHeader.NumberOfSections에 기록

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct_IMAGE_SECTION_HEADER {
        BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
        // section의 이름 (text, data, rdata ...)
        union {
                DWORD PhysicalAddress;
                DWORD VirtualSize;
        }Misc;
```

```
        DWORD VirtualAddress;   // section entry point (RVA)
        DWORD SizeOfRawData;   // size of section in file (파일에서의 해당 섹션의 크기)
        DWORD PointerToRawData;   // 파일에서의 section 시작 주소
        DWORD Characteristics;   // 권한
        // => #define IMAGE_SCN_MEM_EXECUTE   0x20000000 // 실행가능
        // => #define IMAGE_SCN_MEM_READ   0x40000000 // 읽기가능
        // => #define IMAGE_SCN_MEM_WRITE   0x80000000 // 쓰기가능
        .....
}
```

1) VirtualSize : 메모리에서 섹션이 차지하는 크기
2) VirtualAddress : 메모리에서 섹션의 시작주소(RVA)
3) SizeOfRawData : 파일에서 섹션이 차지하는 크기
4) PointerToRawData : 파일에서 섹션의 시작 위치
5) Characteristics : 섹션의 속성

**RVA to RAW**

-Image : PE 파일의 정보가 메모리에 로딩된 상태를 지칭
-섹션에서 메모리 시작 주소 값인 VirtualAddress와 파일의 시작 주소인 PointerToRawData는 다를 수가 있다. 그러므로 메모리와 파일 사이에 데이터 매핑이 필요하다 (RVA to RAW, RAW to RVA)
-다만, RAW – PointerToRawData = RVA - VirtualAddress

**IAT** (Import Address Table) => DLL 의 암시적 링크에 대한 매커니즘을 제공한다.
-process, memory, dll 구조등에 대한 내용이 기록됨
-프로그램이 어떤 라이브러리에서 어떤 함수를 사용하고 있는지를 기술한 테이블

**DLL** (Dynamic Linked Library)
-멀티 태스킹을 수행할 때, 정적 라이브러리의 비효율성을 개선하기 위해 개발됨
=>개별 프로그램에 라이브러리를 각각 포함시키지 말고 별도로 DLL로 구성하여 불러 쓴다
=>**한번 로딩된 DLL의 코드와 리소스는 Memory mapping 기술로 여러 프로세스에서 공유하여 쓴다**
=>라이브러리의 업데이트가 필요하면 **해당 DLL 파일만 교체**하자
-종류
1) **명시적 링크** : 프로그램에서 사용하는 순간에 로딩되고, 사용이 끝나면 메모리에서 해제
2) **암시적 링크** : 프로그램 시작할 때 같이 로딩되고, 프로그램이 종료될 때 메모리에서 해제

-윈도우가 부팅되면 system dll 부터 자동으로 올라온다 (ex. kernel32.dll )
-PE 파일은 프로그램이 import 하여 사용할 라이브러리를 IDT에 IMAGE_IMPORT_DESCRIPTOR(IID) 구조체로 명시한다
-프로그램은 여러 개의 라이브러리를 사용하기 때문에, 라이브러리 개수만큼 구조체 배열 형식으로 존재한다. 구조체 배열의 마지막 원소는 NULL 구조체

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
        union {
                DWORD Characteristics;
                DWORD OriginalFirst
        }
}
```