리눅스 C 프로그래밍

Doo-ok Seo

clickseo@kw.ac.kr

http://www.clickseo.com

Contents

- gcc
- make

gcc

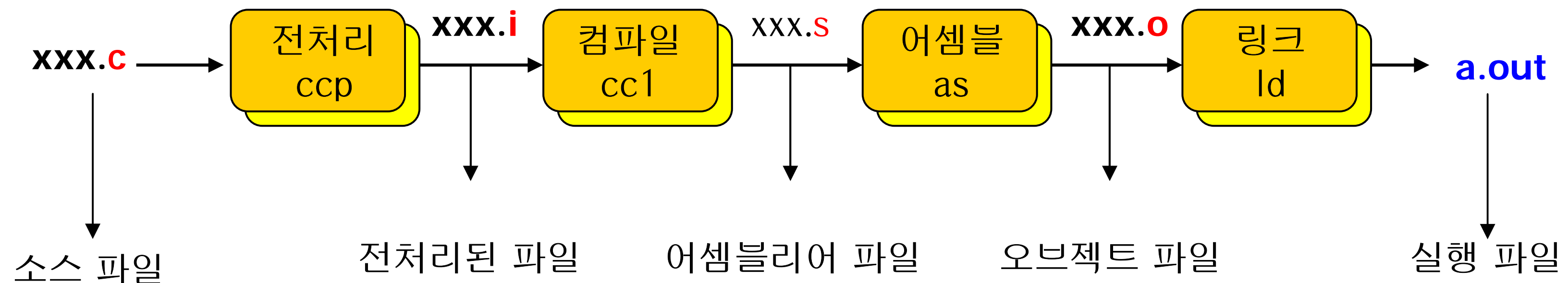
- gcc
  - gcc 기본 개념 및 사용법
  - gcc 옵션
  - 라이브러리 지정 옵션
- make

gcc 기본 개념 및 사용법

- **gcc란 ?**
  - 원래는 “GNU C Compiler”를 의미
  - 1999년부터 **“GNU Compiler Collection”**을 의미한다. 따라서 C 언어뿐 아니라 C++, 오브젝티브 C, 포트란, 자바 등의 컴파일러를 포함하는 포괄적 의미를 가진다.

gcc 기본 개념 및 사용법 (cont'd)

- gcc란 ? (cont'd)
  - 일반적으로 gcc를 컴파일러라고 하지만 정확히 말하면 gcc는 소스 파일을 이용해 실행 파일을 만들 때까지 필요한 프로그램을 차례로 실행시키는 툴이다.



gcc 기본 개념 및 사용법 (cont'd)

- **gcc 기본 사용법**
  - gcc는 파일 확장자에 따라 처리 방법을 달리한다.
    - 예) 대표적인 확장자 .c 인 경우는 gcc로 전처리기, 컴파일, 어셈블, 링크 과정을 거쳐야 실행 파일이 완성되는 것이다.

확장자	종류	처리 방법
.c	C 소스 파일	gcc로 전처리, 컴파일, 어셈블, 링크
.c .cc	C++ 소스 파일	g++로 전처리, 컴파일, 어셈블, 링크
.i	전처리된 c 파일	gcc로 컴파일, 어셈블, 링크
.ii	전처리된 c++ 파일	g++로 컴파일, 어셈블, 링크
.s	어셈블리어로 된 파일	어셈블, 링크
.S	어셈블리어로 된 파일	전처리, 어셈블, 링크
.o	오브젝트 파일	링크
.a .so	컴파일된 라이브러리 파일	링크

gcc 기본 개념 및 사용법 (cont'd)

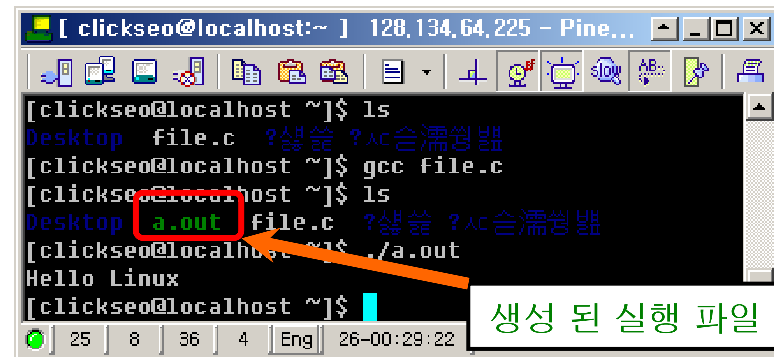
- 실습예제

[실습예제] file.c

```
#include <stdio.h>
int main()
{
        printf ("Hello Linux\n");

        return 0;
}
```

```
[clickseo@comlab /]$ gcc file.c
[clickseo@comlab /]$ ./a.out
```



gcc 옵션

● gcc 옵션

옵 션	의 미
-E	전처리를 실행하고 컴파일을 중단하게 한다.
-c	소스파일을 오브젝트 파일로만 컴파일하고 링크하는 과정을 생략한다.
-o	바이너리 형식의 출력 파일 이름을 지정하는데, 지정하지 않을 시 a.out 이라고 기본이름 생성
-I	헤더 파일을 검색하는 디렉토리 목록을 추가한다.
-L	라이브러리 파일을 검색하는 디렉토리 목록을 추가한다.
-l	라이브러리 파일을 컴파일 시 링크한다.
-g	바이너리 파일에 표준 디버깅 정보를 포함한다.
-ggdb	바이너리 파일에 GNU 디버거인 gdb 만이 이해할 수 있는 많은 디버깅 정보를 포함시킨다.
-O	컴파일 코드를 최적화한다.
-O*level*	최적한 *level* 단계를 지정한다.
-D*FOO=RAR*	명령라인에서 BAR 값을 가지는 FOO 라는 선행 처리기 매크로를 정의한다.
-static	정적 라이브러리에 링크한다.

gcc 옵션 (cont'd)

- gcc 옵션 (cont'd)

옵 션	의 미
-ansi	표준과 충돌하는 GNU 확장안을 취소, ANSI/ISO C 표준을 지원, ANSI 호환코드를 보장 안함
-traditional	과거 스타일의 함수 정의 형식과 같이 전통적인 K&R C언어 형식을 지원한다.
-MM	make 호환의 의존성 목록을 출력한다.
-V	컴파일의 각 단계에서 사용되는 명령을 보여준다.

gcc 옵션 (cont'd)

- **-o 옵션**
  - 생성되는 출력 파일 이름을 지정한다.
  - Syntax

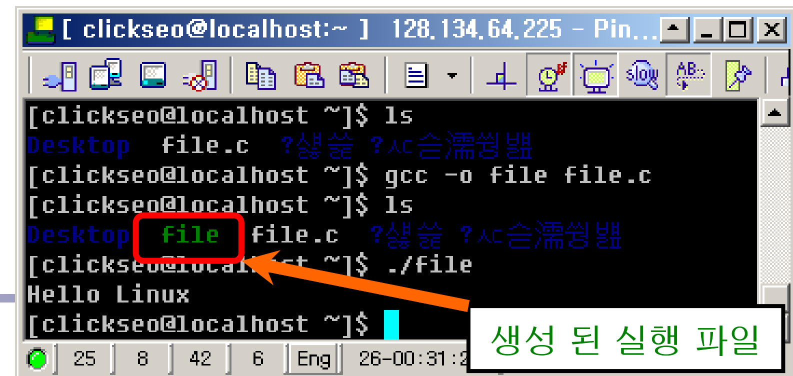
    gcc –o 출력파일이름 소스파일이름

    gcc 소스파일이름 -o 출력파일이름

    (출력파일과 소스파일의 순서는 바뀌어도 상관없다.)

```
[clickseo@comlab /]$ gcc -o file file.c
```

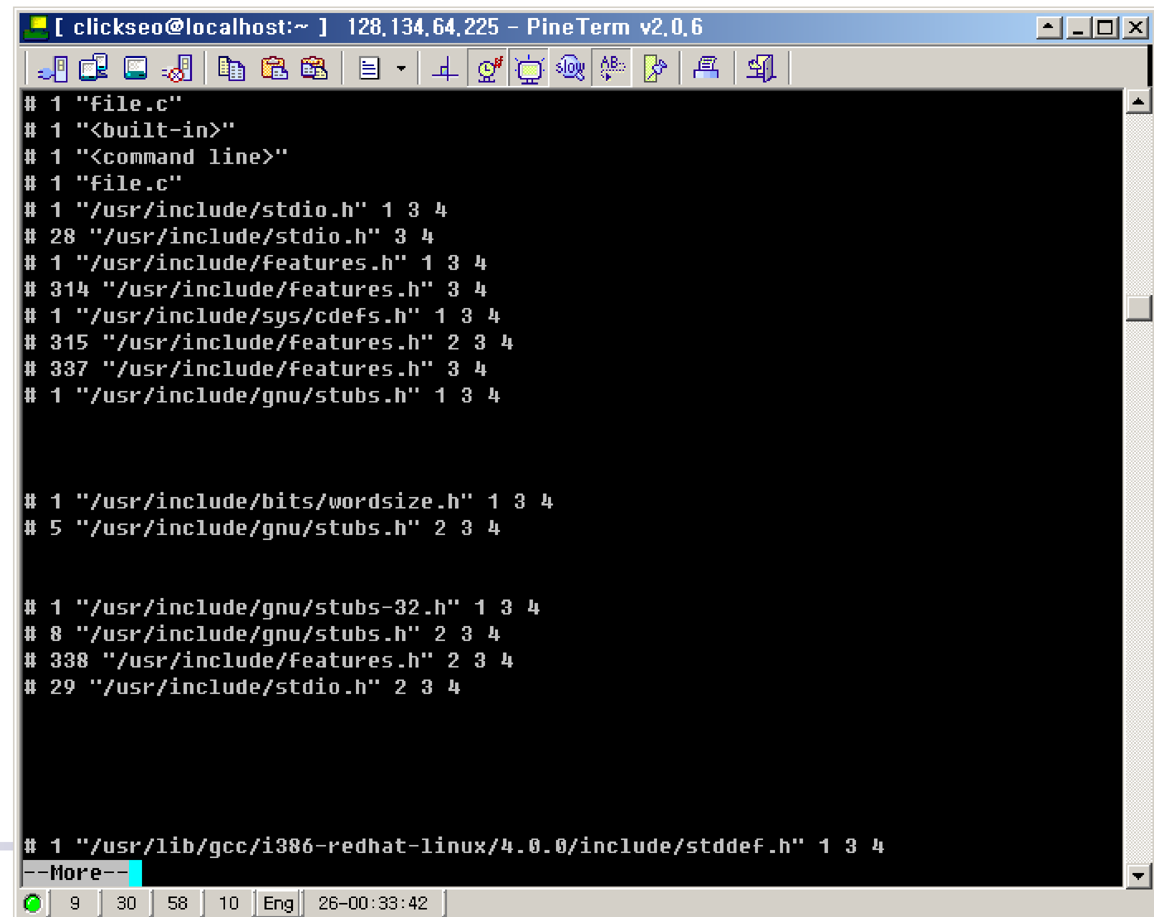
- file.c 소스파일에 file 이라는 출력파일 이름을 지정해 주어서 a.out 이라는 기본 파일을 생성하지 않는다.



gcc 옵션 (cont'd)

- -E 옵션
  - 전처리까지만 실행하고 결과를 화면에 출력한다.
  - Syntax

    gcc –E 소스파일이름



gcc 옵션 (cont'd)

- -c 옵션
  - 소스를 오브젝트 파일로만 컴파일하고 링크하는 과정을 생략
  - Syntax

    gcc –c 소스파일이름

```
[ clickseo@localhost:~ ]  128.134.64.225 - PineTerm v2.0.6
[clickseo@localhost ~]$ ls
Desktop  file.c
[clickseo@localhost ~]$ gcc -c file.c
[clickseo@localhost ~]$ ls
Desktop  file.c  file.o
[clickseo@localhost ~]$ gcc file.o -o file
[clickseo@localhost ~]$ ls
Desktop  file  file.c
[clickseo@localhost ~]$ gcc file.o
[clickseo@localhost ~]$ ls
Desktop  a.out  file  file.c  file.o
[clickseo@localhost ~]$
```

생성 된 오브젝트 파일

오브젝트 파일을 이용해 file 이라는 실행 파일 생성

오브젝트 파일을 이용해 생성 된 실행 파일

gcc 옵션 (cont'd)

- -c 옵션 (cont'd)
  - 분리 컴파일
    - 여러 파일로 분리 작성된 하나의 프로그램을 컴파일

[예제 1] main.c

```
extern void hi() ;
main()
{
        hi() ;
}
```

[예제 2] hi.c

```
#include <stdio.h>
void hi()
{
        printf ("Linux World \n");
}
```

```
[clickseo@comlab /]$ gcc  main.c  hi.c  -o  test
```

```
[clickseo@comlab /]$ gcc -c main.c
[clickseo@comlab /]$ gcc -c hi.c
[clickseo@comlab /]$ gcc main.o hi.o -o test
```

```
[clickseo@localhost ~]$ ls
Desktop  file.c  hi.c  main.c
[clickseo@localhost ~]$ gcc main.c hi.c -o test
[clickseo@localhost ~]$ ls
Desktop  file.c  hi.c  main.c  test
[clickseo@localhost ~]$ ./test
Linux Wolrd
[clickseo@localhost ~]$
```

```
[clickseo@localhost ~]$ ls
Desktop  file.c  hi.c  main.c
[clickseo@localhost ~]$ gcc -c main.c
[clickseo@localhost ~]$ gcc -c hi.c
[clickseo@localhost ~]$ ls
Desktop  file.c  hi.c  hi.o  main.c  main.o
[clickseo@localhost ~]$ gcc main.o hi.o -o test
[clickseo@localhost ~]$ ls
Desktop  file.c  hi.c  hi.o  main.c  main.o  test
[clickseo@localhost ~]$ ./test
Linux Wolrd
[clickseo@localhost ~]$
```

gcc 옵션 (cont'd)

- **-I 옵션**
  - 표준 디렉토리가 아닌 위치에 있는 헤더 파일의 디렉토리를 지정한다.
  - Syntax

    gcc 소스파일이름 **–I** 디렉토리이름

[예제 1] age.c

```
#include <stdio.h>
#include "myheader.h"
main()
{
        printf("%d\n", AGE);
}
```

[예제 2] myheader.h

```
#define AGE 20
```

```
[clickseo@comlab /]$ gcc age.c –I 헤더파일이 있는 디렉토리 경로
```

```
[clickseo@localhost ~]$ ls
Desktop  age.c  file.c  hi.c  main.c  test
[clickseo@localhost ~]$ gcc age.c
age.c:2:22: error: myheader.h:
age.c: In function
age.c:6: error:       undeclared (first use in this function)
age.c:6: error: (Each undeclared identifier is reported only once
age.c:6: error: for each function it appears in.)
[clickseo@localhost ~]$ gcc age.c -I ./test
[clickseo@localhost ~]$ ls
Desktop  a.out  age.c  file.c  hi.c  main.c  test
[clickseo@localhost ~]$ ./a.out
20
[clickseo@localhost ~]$
```

라이브러리 지정 옵션

- **라이브러리란?**
  - 자주 사용되는 유용한 함수에 대해 오브젝트 파일을 모아둔 것
    - 시스템에서 제공하는 라이브러리 디렉토리 /usr/lib
    - lib 로 시작하고 **ar** 명령어에 의해 생성, 확장자 **.a**

라이브러리 지정 옵션

- 라이브러리 생성 절차
  - 오브젝트 파일 생성

```
[clickseo@comlab /]$ gcc  -c plus.c minus.c
```

  - 라이브러리 파일 생성

```
[clickseo@comlab /]$ ar r libmy.a plus.o minus.o
```

  - 라이브러리에 자체에 정보 추가

```
[clickseo@comlab /]$ ranlib libmy.a
```

```
[clickseo@comlab /]$ ar rs libmy.a
```

  - 헤더파일 생성
  - 라이브러리 옵션 지정을 포함한 gcc 컴파일

라이브러리 지정 옵션 (cont'd)

- 라이브러이 지정 옵션
  - **-I** 옵션
    - 표준 라이브러리가 헤더 파일을 찾는 헤더 경로를 추가한다.
  - **-L** 옵션
    - 표준 라이브러리가 디렉토리를 추가한다.
  - **-l** 옵션
    - 표준 라이브러리 이외의 라이브러리를 링크하려고 할 때 사용
      - 예) 수학 라이브러리(math.h)를 사용하려고 한다면...
        - » gcc 명령행 끝에 -lm을 추가하여 libm 라이브러리를 링크시켜야 한다.

라이브러리 지정 옵션

- 라이브러리 생성 예제

plus.c

```
int plus (int x, int y)
{
        return x+y;
}
```

minus.c

```
int minus (int x, int y)
{
      return x-y ;
}
```

./lib/libmy.h

```
extern int plus(int, int);
extern int minus (int, int);
```

```
[clickseo@localhost lib]$ pwd
/home/clickseo/lib
[clickseo@localhost lib]$ ls
libmy.a  libmy.h
[clickseo@localhost lib]$
```

```
[clickseo@localhost ~]$ ls
Desktop  age.c  file.c  hi.c  lib  main.c  minus.c  plus.c  test  test.c
[clickseo@localhost ~]$ gcc -c plus.c minus.c
[clickseo@localhost ~]$ ls
Desktop  file.c  lib     minus.c  plus.c  test
age.c    hi.c    main.c  minus.o  plus.o  test.c
[clickseo@localhost ~]$
```

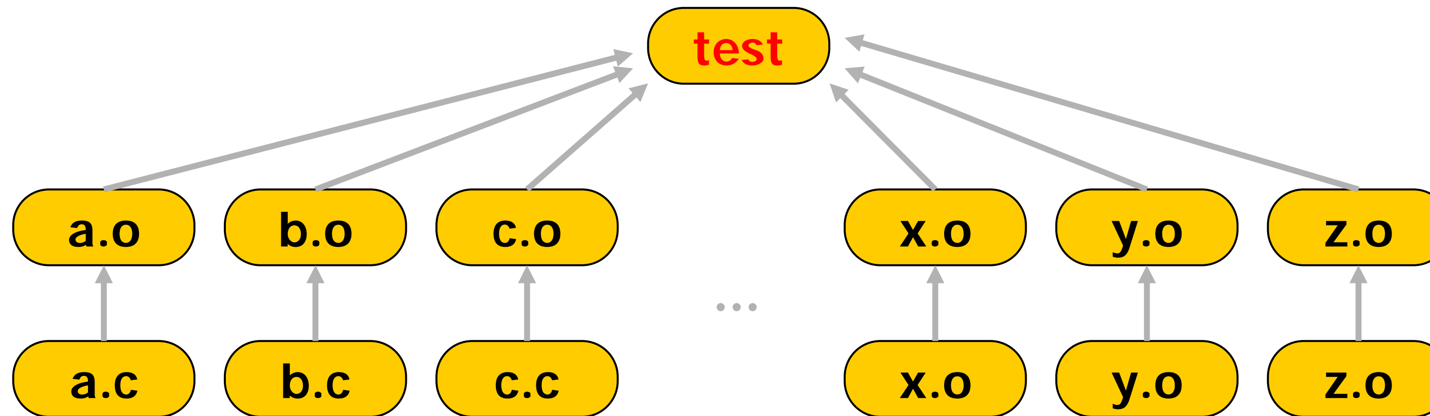
```
[clickseo@localhost ~]$ ar r libmy.a plus.o minus.o
ar: creating libmy.a
[clickseo@localhost ~]$ ls
Desktop  file.c  lib      main.c   minus.o  plus.o  test.c
age.c    hi.c    libmy.a  minus.c  plus.c   test
[clickseo@localhost ~]$ ar rs libmy.a
[clickseo@localhost ~]$ ls
Desktop  file.c  lib      main.c   minus.o  plus.o  test.c
age.c    hi.c    libmy.a  minus.c  plus.c   test
[clickseo@localhost ~]$
```

```
[clickseo@localhost ~]$ gcc -I ./lib -L ./lib -o 1234 test.c -lmy
[clickseo@localhost ~]$ ls
1234     age.c   hi.c  main.c   minus.o  plus.o  test.c
Desktop  file.c  lib   minus.c  plus.c   test
[clickseo@localhost ~]$ ./1234
[clickseo@localhost ~]$
```

make

- gcc
- make
  - make 란
  - make 파일
  - 몇 가지 문법 규칙
  - make 옵션

make란?



위와 같이 여러 파일로 구성된 프로그램이 있을 경우 c.c 소스파일을 수정하면,
모든 파일을 다시 컴파일하고 링크해야 수정이 반영된 test 파일이 생성된다.
그러므로 파일을 하나만 수정해도 모든 파일을 다시 컴파일해야 한다는 것이다.

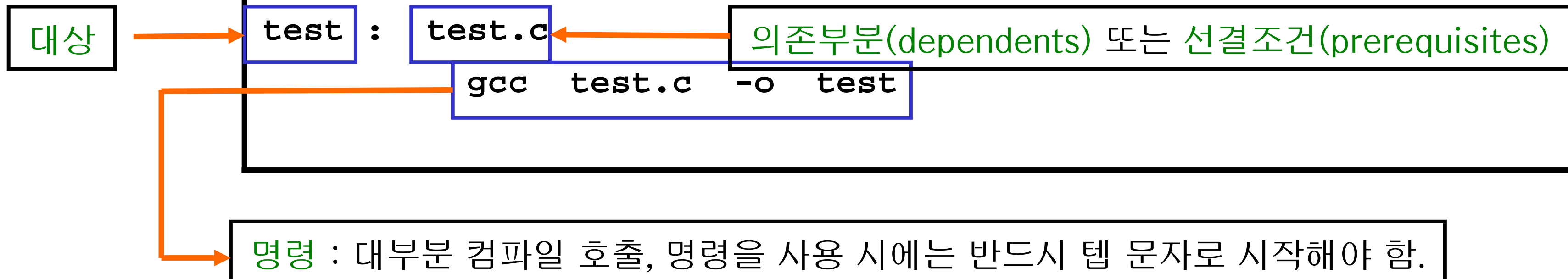
이러한 수고를 들이지 않고 수정된 파일만 자동으로 알아내 컴파일 하고
수정하지 않은 파일에 대해서는 기존 오브젝트 파일을 그대로 이용하게 해주는 유틸리티 툴 이다.

make 파일

make 파일

- make 파일은 애플리케이션의 구성방법을 make에 알려주는 텍스트 파일

```
대상(target) : 대상에 의존되는 파일1 [파일2 …]
[tab간격]  명령(command)
```



make 파일 (cont'd)

make 파일 생성시 주의 사항

- 각 요소를 구분하는데 있어 콤마(,) 같은 건 사용하지 않고 공백으로 한다.
- 명령을 시작하기 전에는 항상 <TAB>을 넣는다.
  - 절대 스페이스 키나 다른 키는 사용해선 안된다.
  - 그 밖의 다른 곳에서는 TAB을 사용하지 말라.
- make 파일 내에서 항목의 순서는 중요하지 않다.
  - make는 어떤 파일이 어느 곳에 의존적인지 알아내어 올바른 순서로 명령을 수행한다.

make 파일 (cont'd)

- make 파일 예제

[예제 1] test1.c

```
#include <stdio.h>
#include "a.h"
void func1();
void func2();
main()
{
        printf("test1\n");
        func1 ();
        func2 ();

}
```

[예제 2] test2.c

```
#include <stdio.h>
#include "a.h"
#include "b.h"
extern void func1()
{
        printf("test2 \n");
}
```

[예제 3] test3.c

```
#include <stdio.h>
#include "b.h"
#include "c.h"
extern void func2()
{
        printf("test3 \n");
}
```

make 파일 (cont'd)

- make 파일 예제 (cont'd)

① 헤더 파일 생성

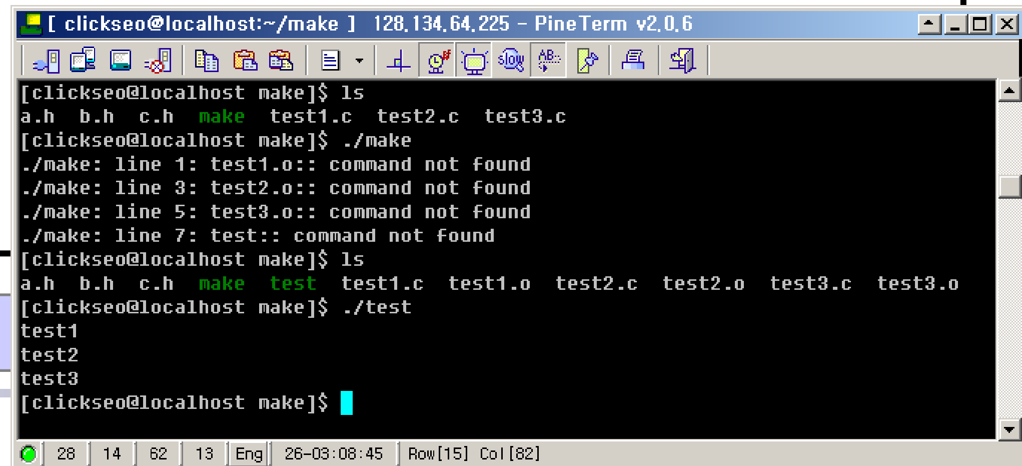
```
[clickseo@comlab /]$ touch a.h
[clickseo@comlab /]$ touch b.h
[clickseo@comlab /]$ touch c.h
```

② **make** 파일 생성

[makefile]

```
test:   test1.o   test2.o   test3.o
        gcc   -o   test   test1.o   test2.o   test3.o
test1.o:   test1.c   a.h
        gcc   -c   test1.c
test2.o:   test2.c   a.h   b.h
        gcc   -c   test2.c
test3.o:   test3.c   b.h   c.h
        gcc   -c   test3.c
```

③ **make** 실행



몇 가지 문법 규칙

매크로(Macro)

- make 파일을 작성하다 보면 같은 파일 이름을 여러 번 써야 하는 경우가 있다. 이런 경우에 매크로를 사용하면 편리하고 명령을 단순화 시킬 수 있다.

```
M_NAME = value
```

- 매크로 사용시 대소문자 모두 가능
  - 보통 대문자로 쓰는 것이 관례이다.
- make 파일 상단에 정의

몇 가지 문법 규칙 (cont'd)

- 매크로 사용 예제

〔makefile〕

```
OBJF = test1.o  test2.o  test3.o
test: $(OBJF)
       gcc  -o  test  $(OBJF)
test1.o:  test1.c  a.h
       gcc  -c  test1.c
test2.o:  test2.c  a.h  b.h
       gcc  -c  test2.c
test3.o:  test3.c  b.h  c.h
       gcc  -c  test3.c
clean:
       rm $(OBJF)
```

매크로 정의

$(OBJF) = test1.o test2.o test3.o

몇 가지 문법 규칙 (cont'd)

내부 매크로

내부 매크로	의　미
$@	현재 목표 파일의 이름
$*	확장자를 제외한 현재 목표 파일의 이름
$<	현재 필수 조건 파일 중 첫 번째 파일 이름
$?	현재 대상보다 최근에 변경된 필수 조건 파일 이름
$^	현재 모든 필수 조건 파일들

몇 가지 문법 규칙 (cont'd)

- 내부 매크로 사용 예제

(makefile)

```
OBJF = test1.o  test2.o  test3.o
test:  $(OBJF)
       gcc  -o $@ $^
test1.o:  test1.c  a.h
       gcc  -c $<
test2.o:  test2.c  a.h  b.h
       gcc  -c $*.c
test3.o:  test3.c  b.h  c.h
       gcc  -c $*.c
clean:
       rm $(OBJF)
```

- `$@` : 현재 대상 파일의 이름을 의미하므로 test를 나타냄
- `$^` : OBJF로 정의된 매크로 값
- `$<` : 현재 대상 파일 test1.o가 의존하는 필수 조건 파일 중 첫 번째 파일 이름을 의미 test1.c
- `$*.c` : 확장자 .c 를 제외한 현재 대상 파일의 이름을 의미 각 각 test2 , test3 을 의미

몇 가지 문법 규칙 (cont'd)

● 매크로 치환

- 이미 정의된 매크로의 내용을 치환으로 변경

```
$(M_NAME:old=new)
```

예제)

```
OBJF   =   test1.o   test2.o   test3.o
SRCS   =   $(OBJF:.o=.c)
```

➔ OBJF의 확장자 부분이 .o에서 .c로 바뀌게 되어 SRCS에 저장된다.
결국 SRCS 는 test1.c test2.c test3.c' 를 의미한다.

몇 가지 문법 규칙 (cont'd)

- **명시적 규칙**
  - make 가 해야 할 일을 명확히 지정

- **암시적 규칙**
  - make 내에 미리 정의된 규칙을 이용해 make파일을 단순화 시키는 규칙

(makefile)

```
OBJF  =  test1.o  test2.o  test3.o
test:  $(OBJF)
        gcc  -o  $@  $(OBJF)
clean:
        rm $(OBJF)
```

make 실행 시 오브젝트 파일을 암시적 규칙에 따라 의존하는 파일을 찾고 컴파일러 호출까지 수행한다.
(단, 내장된 규칙에 의해 gcc 가 아니라 cc 를 호출)

몇 가지 문법 규칙 (cont'd)

● 접미사 규칙

예제)

```
.c.o:                                       ...①
              gcc  -c  $<  $(CFLAGS)        ...②
```

① **.c.o**

.c 라는 확장자를 가진 파일을 사용해 .o 라는 확장자를 가진 파일을 만들 것임을 make에 알리는 역할을 한다.

② **$<**

확장가가 .c인 파일명을 의미한다.

$(CFLAGS)

C 컴파일러를 위한 플래그를 위해 미리 정의된 변수

몇 가지 문법 규칙 (cont'd)

- **패턴 규칙**
  - 암시적 규칙에 의존 했을 경우 일어나는 오류방지
  - 접미사 규칙과 비슷하나 더 뛰어난 기능을 가짐

[makefile]

```
OBJF  =  test1_d.o  test2_d.o  test3_d.o
test:  $(OBJF)
        gcc  -o  $@  $(OBJF)
%_d.o:  %.c
        gcc  -c  -g  $<  -o  $@
clean:
        rm $(OBJF)
```

- `%_d.o:  %.c` → 확장자 .c 인 모든 파일에 대해 _d 를 붙인 오브젝트 파일을 생성하겠다는 의미
- `gcc  -c  -g  $<  -o  $@` → -g 옵션을 주어 컴파일 시 디버깅 정보를 삽입

References

[1] 이만용 역, “러닝 리눅스 4/e”, 한빛미디어, pp. 537 - 564