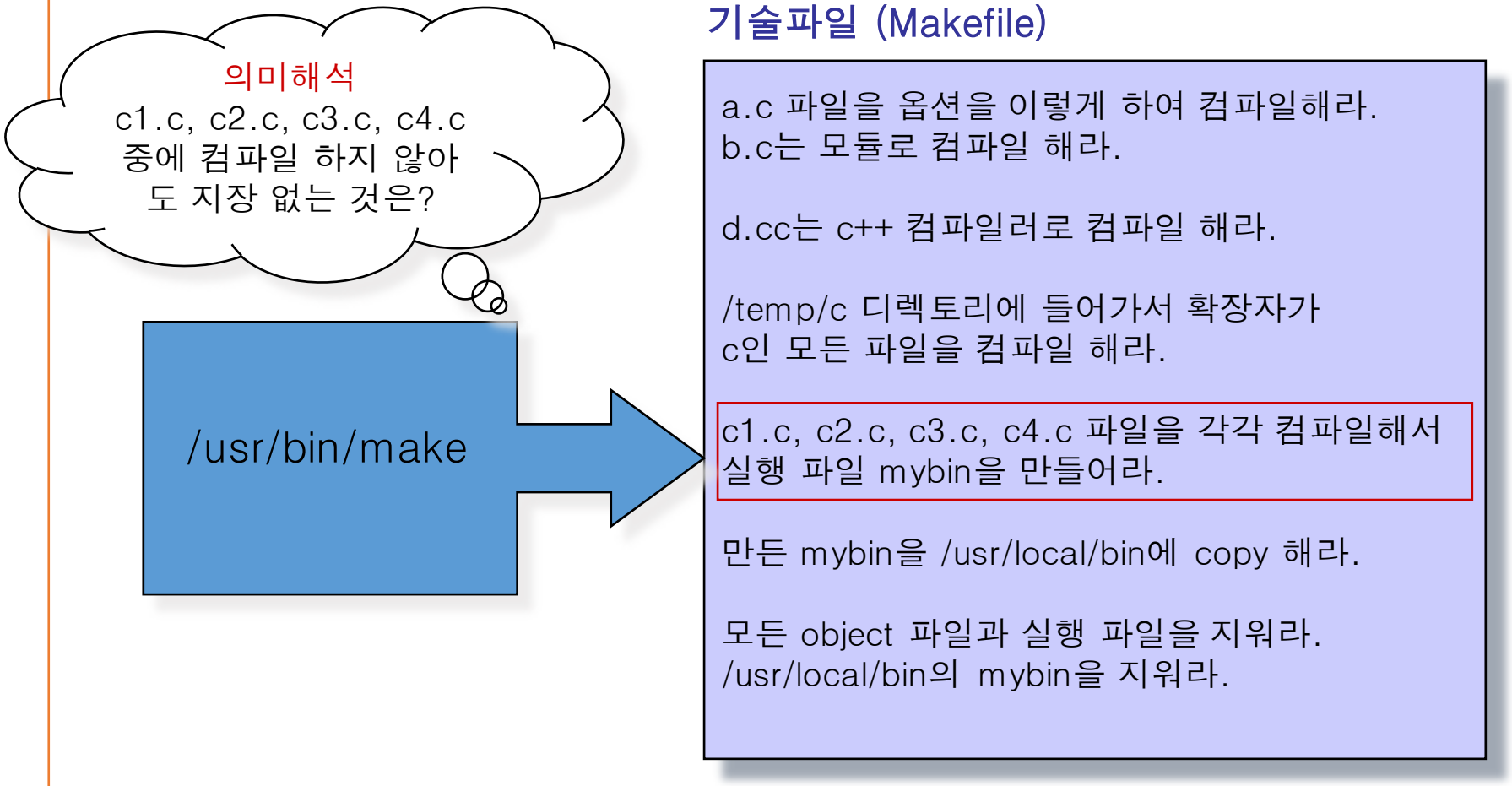
GNU make

make의 정의

☞**정의** : make는 프로젝트 파일관리 유틸리티이다.

☞**구조** :



make의 필요성

☞make의 필요성

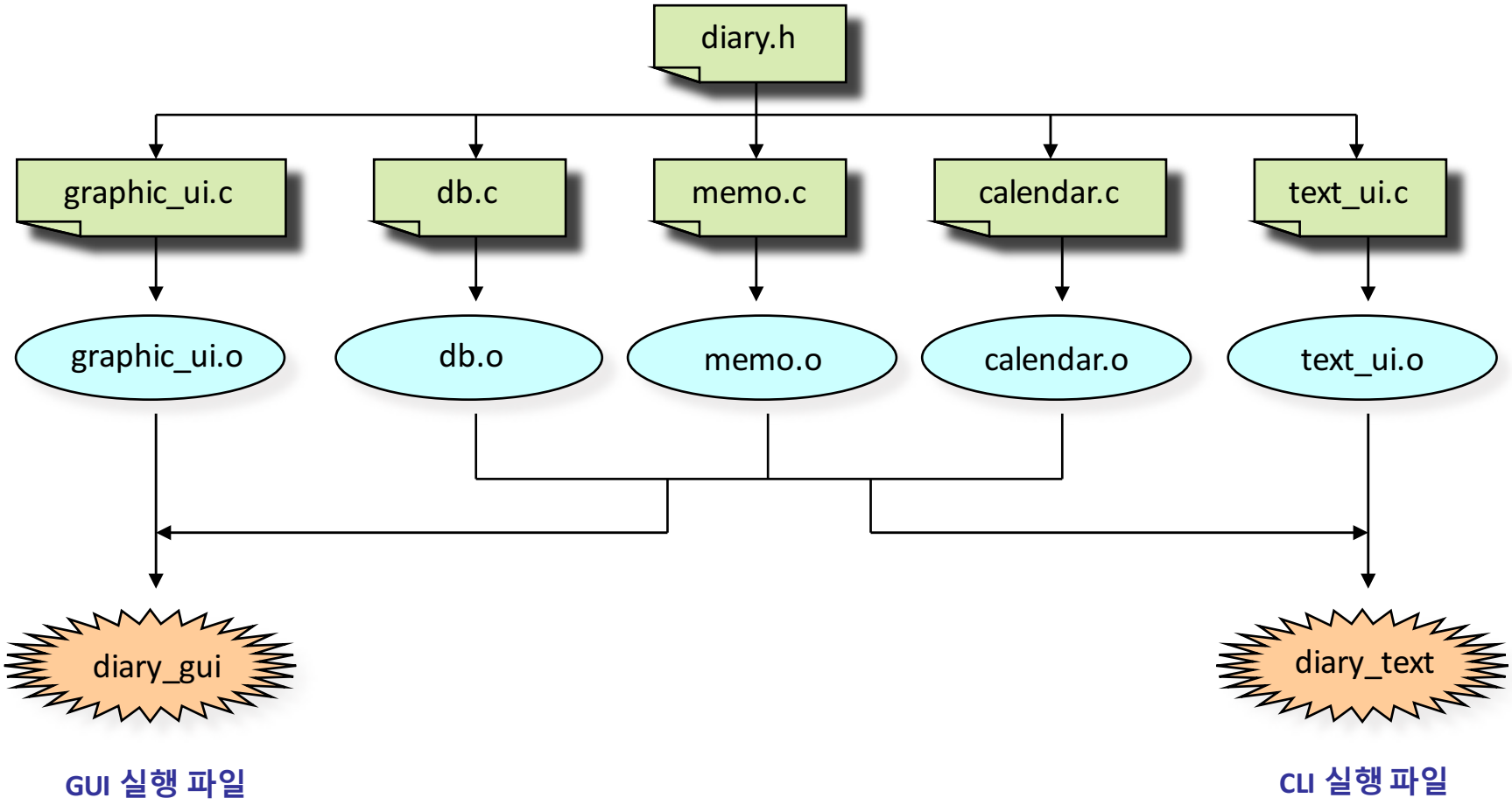
- 프로그램의 종속구조를 빠르게 파악할 수 있고 관리가 용의
  - 컴파일 시간을 절약
  - 소스의 그룹에서 실행 파일의 공통된 루틴들 쉽게 컴파일시에 포함 시킬 수가 있다.
  - 단순 반복작업을 자동화시켜 개발하는 노력을 줄일 수 있다.
  - 재작성의 최소화

☞make를 주로 사용하는 분야

- 소스의 컴파일
- DocBook 문서의 컴파일 (pdf, txt, html 등)
- 기타 순차적 반복작업에 이용

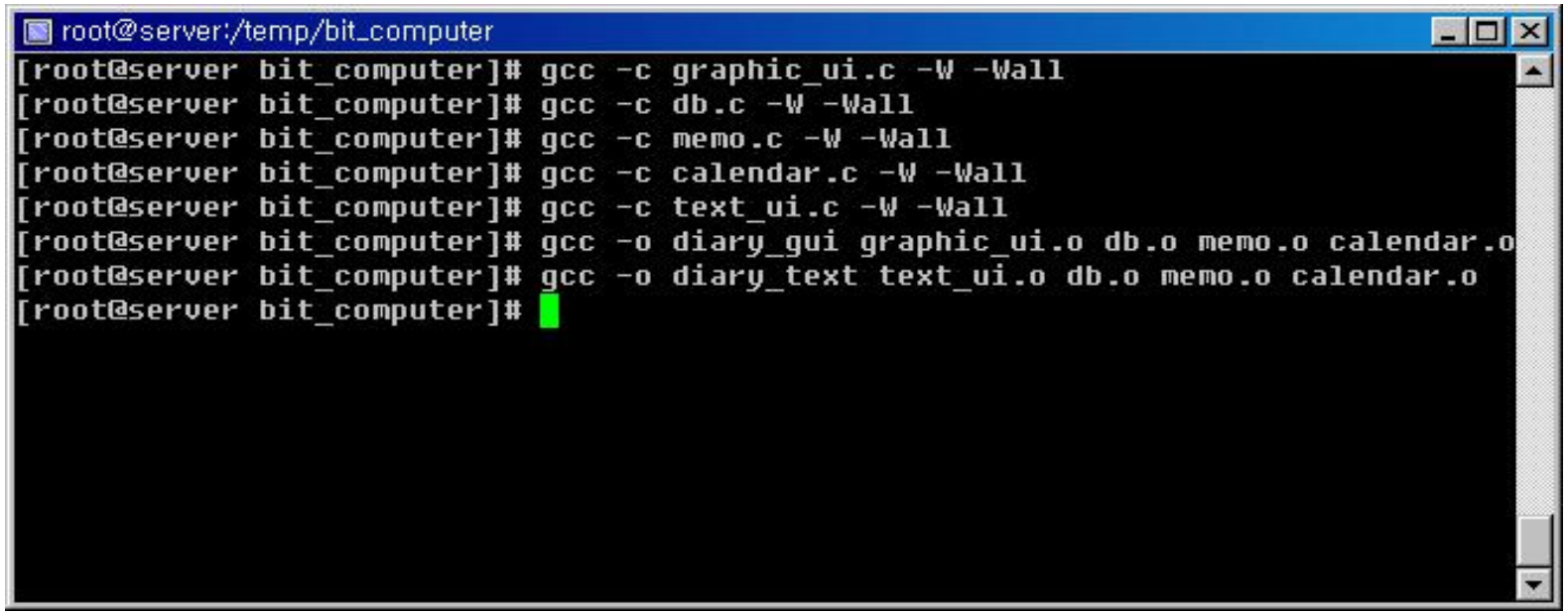
make의 사용 예 (1/4)

여기 UI가 두 종류인 다이어리 프로그램 작성 예가 있다.



make의 사용 예 (2/4)

make를 사용하지 않는 컴파일 예 1



make를 사용하지 않는 컴파일 예 2

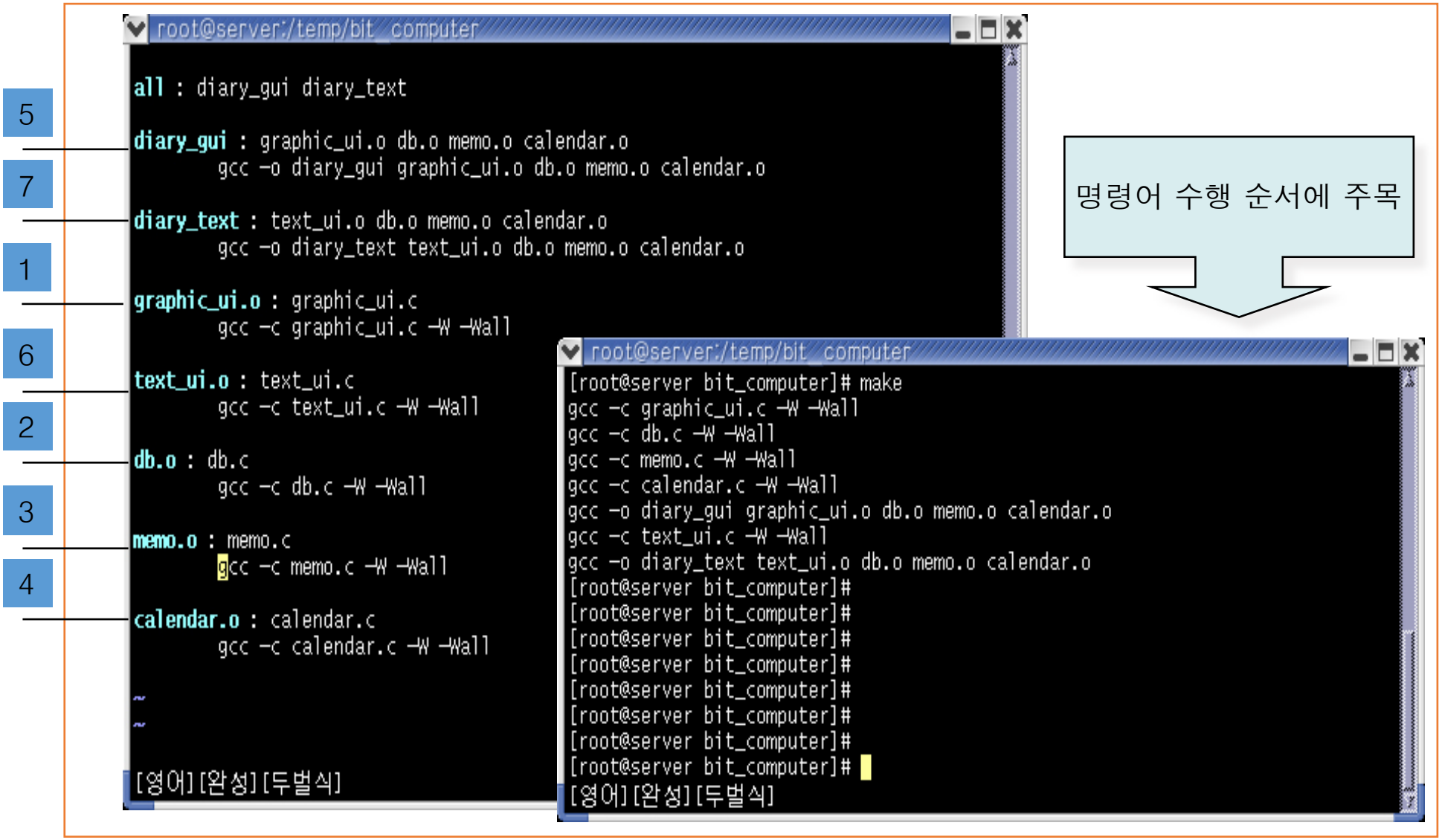
make의 사용 예 (3/4)

make를 사용하여 컴파일 한다면 다음과 같은 기술 파일을 작성하고
make라고 명령만 내려주면 된다.

./Makefile의 내용

```
all : diary_gui diary_text
diary_gui : graphic_ui.o db.o memo.o calendar.o
        gcc -o diary_gui graphic_ui.o db.o memo.o calendar.o
diary_text : text_ui.o db.o memo.o calendar.o
        gcc -o diary_text text_ui.o db.o memo.o calendar.o

graphic_ui.o : graphic_ui.c
        gcc -c graphic_ui.c -W -Wall
text_ui.o : text_ui.c
        gcc -c text_ui.c -W -Wall
db.o : db.c
        gcc -c db.c -W -Wall
memo.o : memo.c
        gcc -c memo.c -W -Wall
calendar.o : calendar.c
        gcc -c calendar.c -W -Wall
```

make의 사용 예 (4/4)


5
7
1
6
2
3
4
root@server:/temp/bit_computer
all : diary_gui diary_text
diary_gui : graphic_ui.o db.o memo.o calendar.o
gcc -o diary_gui graphic_ui.o db.o memo.o calendar.o
diary_text : text_ui.o db.o memo.o calendar.o
gcc -o diary_text text_ui.o db.o memo.o calendar.o
graphic_ui.o : graphic_ui.c
gcc -c graphic_ui.c -W -Wall
text_ui.o : text_ui.c
gcc -c text_ui.c -W -Wall
db.o : db.c
gcc -c db.c -W -Wall
memo.o : memo.c
gcc -c memo.c -W -Wall
calendar.o : calendar.c
gcc -c calendar.c -W -Wall
[영어][완성][두벌식]
명령어 수행 순서에 주목
root@server:/temp/bit_computer
[root@server bit_computer]# make
gcc -c graphic_ui.c -W -Wall
gcc -c db.c -W -Wall
gcc -c memo.c -W -Wall
gcc -c calendar.c -W -Wall
gcc -o diary_gui graphic_ui.o db.o memo.o calendar.o
gcc -c text_ui.c -W -Wall
gcc -o diary_text text_ui.o db.o memo.o calendar.o
[root@server bit_computer]#
[영어][완성][두벌식]

기술 파일의 기본 구조

설명 : Makefile은 make의 액션을 기술해 놓은 파일임

```
# 주석은 '#'으로 시작
CC = arm-linux-gcc            # 매크로의 정의

target1  :  dependency1  dependency2  …        # 룰1
            command1
            command2
            …
            …

target2  : dependency1  dependency2  …         # 룰2
            command1
            command2
            …
```

dependency 파일들이 존재하고 command가 정상적으로 수행 되었을 때의 결과

target2를 위해 필요한 항목들

target2를 만들 때 처리 명령들

기술파일 구문 기본 규칙 (1/2)

1. 명령어의 시작은 반드시 tab으로 시작되어야 한다.

```
target1 : dependency1 dependency2 ...
	command1
	command2
```

Tab으로 시작 한다.

2. 비어 있는 행은 무시된다.

```
target1 : dependency1 dependency2 ...

	command1
```

이 행은 무시된다.

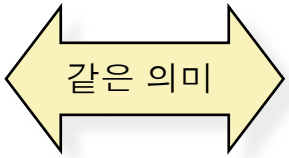
3. '#'을 만나면 개행 문자를 만날 때까지 무시한다.

```
target1 : dependency1 dependency2 ...    # 주석
	command1                         # 주석
# 주석
```

기술파일 구문 기본 규칙 (2/2)

4. 기술 행이 길어지면 '₩'를 사용하여 이어 나갈 수 있다.

```
all : diary_gui ₩
              diary_text
```




```
all : diary_gui diary_text
```

5. ';'은 명령어 행을 나눌 때 사용한다.

```
target1 : dependency1  dependency2;        command1
```

6. 종속 항목이 없는 target도 사용 가능하다.

```
clean :
            rm -rf *.o target1 target2
```

7. 명령어 부분에는 어떠한 명령어가 와도 상관없다.

```
target1  :  dependency1  dependency2 ...
            cp dependency1  dependency2  /temp/backup
            gcc -o target1 dependency1  dependency2
```

macro의 사용

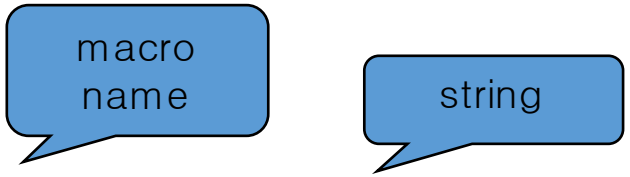
☞ macro란?

- C 언어의 macro와 같이 특정한 사용자 정의 변수에 특정한 string을 정의하고 표현하는 것

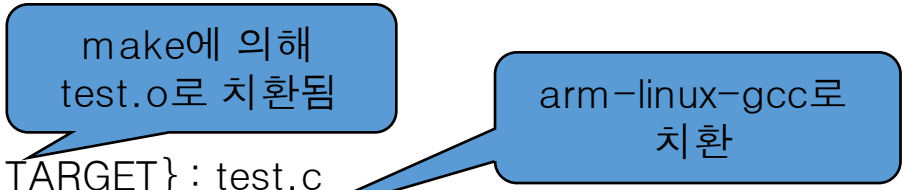
☞ macro를 사용하는 이유

- 반복된 file들이나 명령 option을 쉽게 참조 할 수 있다.
- 일관성을 유지하면서 기술 파일을 쉽게 변경할 수 있다.
- 프로그램을 한번 build하고 또 변화를 주어 build할 때 편리하다.

macro의 사용 예




```
CC          = arm-linux-gcc
CXX         = arm-linux-g++
INCPATH     = -I../2.4.18-rmk7-pxa1-xhyper255/include
CFLAGS      = -D__KERNEL__ -Wall -O2 -march=armv4 -Wa,-mxscale -DMODULE

TARGET      = test.o
```

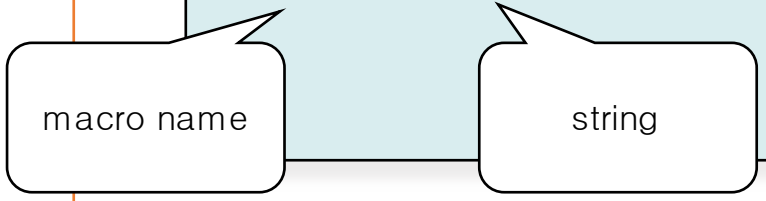



```
${TARGET} : test.c
        ${CC} ${CFLAGS} ${INCPATH} -c -o ${TARGET} test.c

# 위 두 라인은 make에 의해 다음과 같이 치환된다.
#
# test.o : test.c
#         arm-linux-gcc -D__KERNEL__ -Wall -O2 -march=armv4 -Wa,-mxscale
# -DMODULE -I../2.4.18-rmk7-pxa1-xhyper255/include -c -o test.o test.c
```

macro의 구문 규칙 (1/3)

1. macro의 정의는 '='를 포함하는 하나의 문장(행)이다.

```
NAME        = string
```



2. '#'은 주석문의 시작이다.

```
NAME        = string      # 주석
```

3. 여러 행을 사용 할 때는 '₩'를 사용한다.

```
NAME        = -D__KERNEL__ -DMODULE -W -Wall ₩
              -DNO_DEBUG
```

macro의 구문 규칙 (2/3)

4. macro를 참조할 때는 괄호(())나 중괄호 ({})를 둘러싸고 앞에 '$'를 붙인다.

```
NAME        = string
${NAME}             # string
$(NAME)             # string
${NAME}.c           # string.c
macro_${NAME}       # macro_string
```

5. 정의 되지 않은 macro를 참조할 때는 null 문자열로 치환된다.

```
NAME        = string
my ${NAME}          # my string로 치환된다.
my ${KKK}           # my로 치환된다. (KKK는 정의 되지 않음)
```

6. 매크로 정의시 이전에 정의된 매크로를 참조하여 정의 할 수 있다.

```
NAME        = string
NAME2       = my ${NAME}
```

macro의 구문 규칙 (3/3)

7. 중복된 정의는 최후에 정의된 매크로를 사용한다.

```
NAME      = stringA
NAME      = stringB
${NAME}             # stringB로 치환된다.
```

8. 여러 대입 기법을 사용할 수 있다.

```
NAME1     = string          # 재귀적 확장 매크로
NAME2     := string         # 단순 확장 매크로
NAME2     += string
NAME3     ?= string
```

여러 대입 기법에 따른 변화 (1/2)

재귀적 확장 매크로와 단순 확장 매크로의 차이

```
A       = $(B) BB
B       = $(C) CC
C       = D

a       := $(b) bb
b       := $(c) cc
c       := d

all :
        @echo ${A}              # D CC BB를 출력한다.
        @echo ${a}              # bb를 출력한다.
```

+=의를 사용한 대입

```
NAME2 := string1
NAME2 += string2      # string1 string2로 치환된다.
```

여러 대입 기법에 따른 변화 (2/2)

?= 를 사용한 대입 기법

```
NAME1     = my
NAME1     ?= string          # NAME1이 앞에 정의되어 있기 때문에
                             # 이절은 무시 된다.


NAME2     ?= string          # string으로 치환된다.
```

macro 사용시 주의 사항

1. 구분을 위해 문자열에 따옴표를(‘’,””)를 넣으면 따옴표를 문자열의 일부로 인식한다.

```
NAME      = “string”
${NAME}                # string이 아니라 “string”을 의미한다.
```

2. macro의 이름 앞에 tab을 두거나, 등호(=) 앞에 콜론(:)을 두어서는 안된다.

```
          NAME      = string          X
NAME:2    = string                    X
```

3. macro는 반드시 사용할 항목보다 먼저 정의해야 한다.

```
${NAME}                # 위에 정의 되어 있지 않음으로 null 문자로 치환된다.
NAME      = string
```

4. macro는 관습적으로 대문자로 표기한다.

5. ‘₩’나 ‘<‘, ‘>’ 같은 shell 메타 문자는 사용을 자제한다.

내부적으로 정의되어 있는 macro

${CC} : 내부적으로 C 컴파일러인 cc로 정의 되어 있다.
${LD} : ld로 정의 되어 있다.
$? : 현재의 target보다 최근에 변경된 필요 항목 리스트
(확장자 규칙에서 사용불가)
$@ : 현재 target의 이름
$< : 현재 target보다 최근에 변경된 필요 항목 리스트
(확장자 규칙에서만 사용가능)
$* : 현재 target보다 최근에 변경된 현재 필요 항목의 이름(확장자 제외).
(확장자 규칙에서만 사용 가능)
$% : 현재의 타깃이 라이블러리 모듈일 때 .o 파일에 대응되는 이름.

(예제) /temp/target : /usr/local/c/test.c
${@F} : 현재 target의 파일 부분 (target)
${<F} : 현재 필요 항목의 파일 부분 (test.c)
${@D} : 현재 target의 디렉토리 부분 (/temp)
${<D} : 현재 필요 항목의 디렉토리 부분 (/usr/local/c)

이러한 내부적으로 정의되어 있는 macro는 make -p 명령어로 확인 할 수 있다.
make -p | grep ^[[:alpha:]]*[[:space:]]*=[[:space:]]

확장자 규칙의 사용

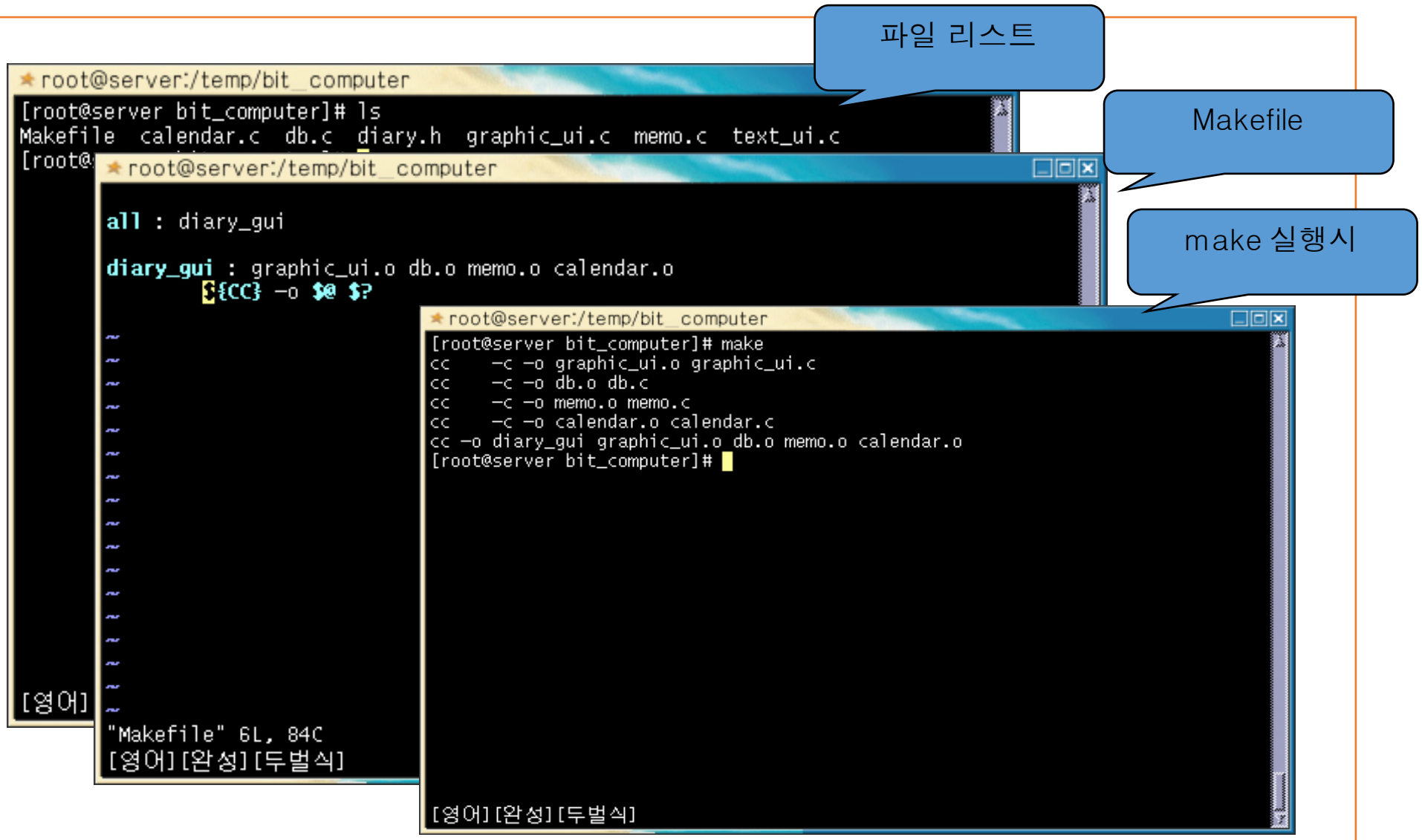
☞ **확장자 규칙이란?**

- 미리 확장자에 따라 정의된 기술 파일 항목
- ex) C 언어 소스 파일은 항상 확장자가 .c를 사용하는데 이것은 C 컴파일러의 필수 요구사항이다. 이러한 확장자 규칙에 따라 make는 여러 동작을 수행 할 수 있다.

☞ **확장자 규칙을 사용하는 이유**

- 기술 파일을 단순, 명료하게 만들 수 있다.

확장자 규칙의 사용 예 (1/2)



확장자 규칙의 사용 예 (2/2)

Makefile의 의미 해석

```
all : diary_gui

diary_gui : graphic_ui.o db.o memo.o calendar.o
        ${CC} -o $@ $?
```

1. 위 기술 파일에서 diary_gui target를 생성하기 위해 make는 필요 항목을 살펴보고 필요 항목을 각각 target으로 설정한다.

2. diary_gui 파일은 graphic_ui.o를 의존하고 있고 graphic_ui.o는 만들어져 있지 않고 graphic_ui.o를 위한 target도 기술 파일에 기술되어 있지 않다.

3. make는 확장자가 .o인 내부 확장자 규칙을 이용해서 다음과 같은 기준으로 현재 디렉토리에서 graphic_ui.o를 생성할 파일을 찾는다.
   - *확장자를 제외하고 graphic_ui.o와 같은 이름이 있어야 한다.*
   - *중요 확장자를 가지고 있어야 한다. (ex : .c, .f, .s, …)*
   - *내부 확장자 규칙에 따라 graphic_ui.o를 만드는데 사용할 수 있어야 한다.*

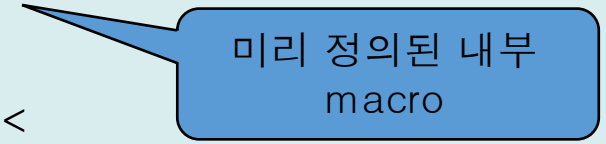
4. 내부 정의 확장자 규칙의 명령어를 이용하여 graphic_ui.o를 생성한다.

이러한 내부 확장자 규칙은 make -p 옵션으로 확인 할 수 있다.

확장자 규칙의 정의

make에는 기본적으로 내부적으로 정의된 확장자 규칙이 있다. 그러나 이러한 확장자 규칙을 별도로 정의하여 사용할 수도 있다.

```
.SUFFIXES: .o .c .cpp .cc .cxx .C          # make가 중요하게 여길 확장자 리스트

.c.o:       # .o와 대응되는 .c를 발견하면 아래의 명령을 실행한다.
      ${CC} ${CFLAGS} ${INCPATH} -c -o $@ $<

.C.o:
      ${CC} ${CFLAGS} ${INCPATH} -c -o $@ $<

.cpp.o:
      ${CXX} ${CXXFLAGS} ${INCPATH} -c -o $@ $<

.cc.o:
      ${CXX} ${CXXFLAGS} ${INCPATH} -c -o $@ $<

.cxx.o:
      ${CXX} ${CXXFLAGS} ${INCPATH} -c -o $@ $<
```



확장자 규칙의 정의 예

Makefile

```
root@server:/temp/bit_computer
CFLAGS          = -O2 -W -Wall -g

########  Implicit rules

.SUFFIXES: .o .c .cpp .cc .cxx .C

.c.o:
        ${CC} ${CFLAGS} ${INCPATH} -c -o $@ $<

.C.o:
        ${CC} ${CFLAGS} ${INCPATH} -c -o $@ $<

.cpp.o:
        ${CXX} ${CXXFLAGS} ${INCPATH} -c -o $@ $<

.cc.o:
        ${CXX} ${CXXFLAGS} ${INCPATH} -c -o $@ $<

.cxx.o:
        ${CXX} ${CXXFLAGS} ${INCPATH} -c -o $@ $<

######### Build rules

all : diary_gui

diary_gui : graphic_ui.o db.o memo.o calendar.o
        ${CC} -o $@ $?

~

[영어][완성][두벌식]
```

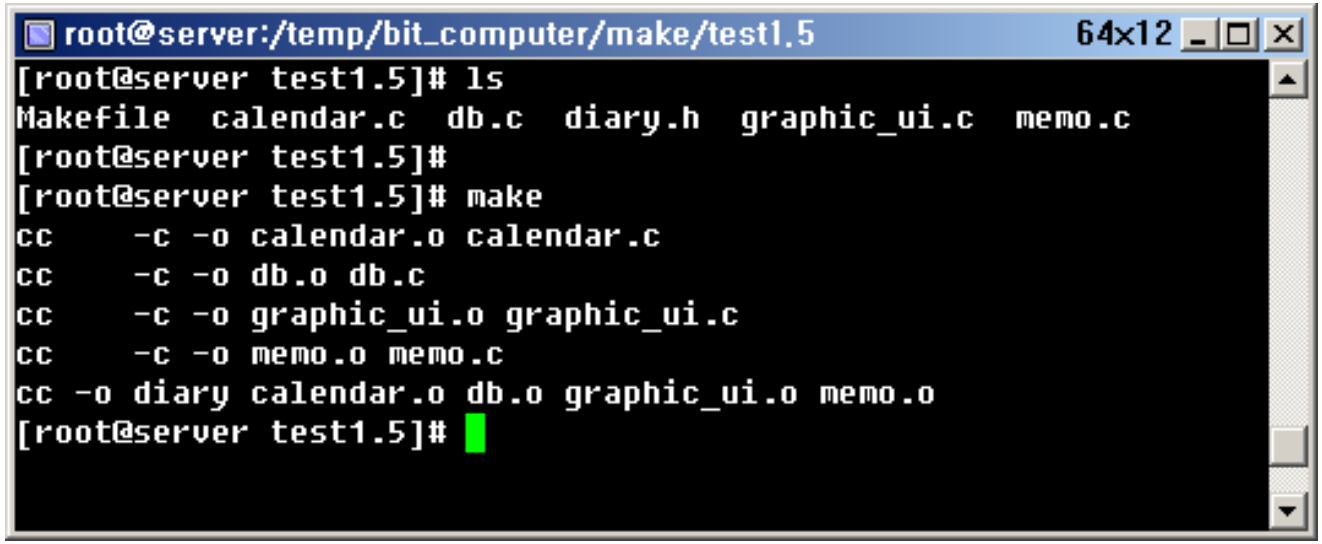
make 수행시

```
root@server:/temp/bit_computer
[root@server bit_computer]# make
cc -O2 -W -Wall -g  -c -o graphic_ui.o graphic_ui.c
cc -O2 -W -Wall -g  -c -o db.o db.c
cc -O2 -W -Wall -g  -c -o memo.o memo.c
cc -O2 -W -Wall -g  -c -o calendar.o calendar.c
cc -o diary_gui graphic_ui.o db.o memo.o calendar.o
[root@server bit_computer]#

[영어][완성][두벌식]
```

대입 참조 기법 사용

Makefile

```
SRCS      = $(wildcard *.c)      # memo.c calendar.c db.c diary_gui.c
OBJECTS   = $(SRCS:.c=.o)        # memo.o calendar.o db.o diary_gui.o
# OBJECTS = $(patsubst %.c, %.o, $(wildcard *.c))    한줄로 사용.

all : diary

diary : $(OBJECTS)
        $(CC) -o $@ $^
```



명령어 사용

명령어 사용

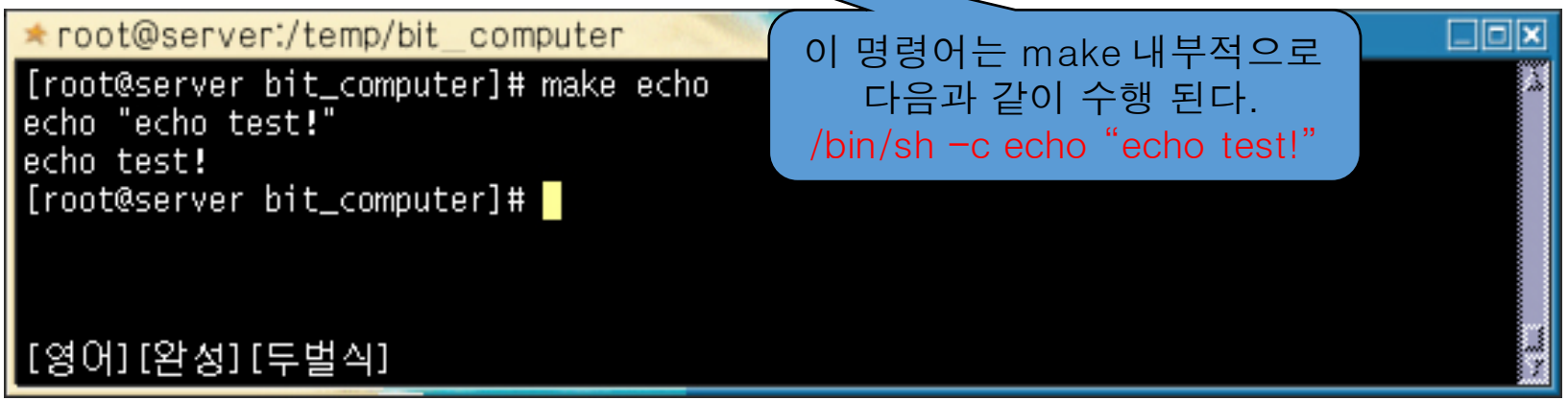
: target을 생성 하는 규칙에는 어떠한 명령어도 사용할 수가 있다.

그러나 이러한 명령어를 사용하는 데는 몇 가지 지켜야 할 주의사항이 있다.

그것은 make는 명령어 수행 시 항상 새로운 shell을 띄워서 명령어를 실행하기 때문에 발생하는 문제들이다.

명령어 사용의 예

```
echo :
        echo “echo test!”
```



명령어 사용 규칙 (1/4)

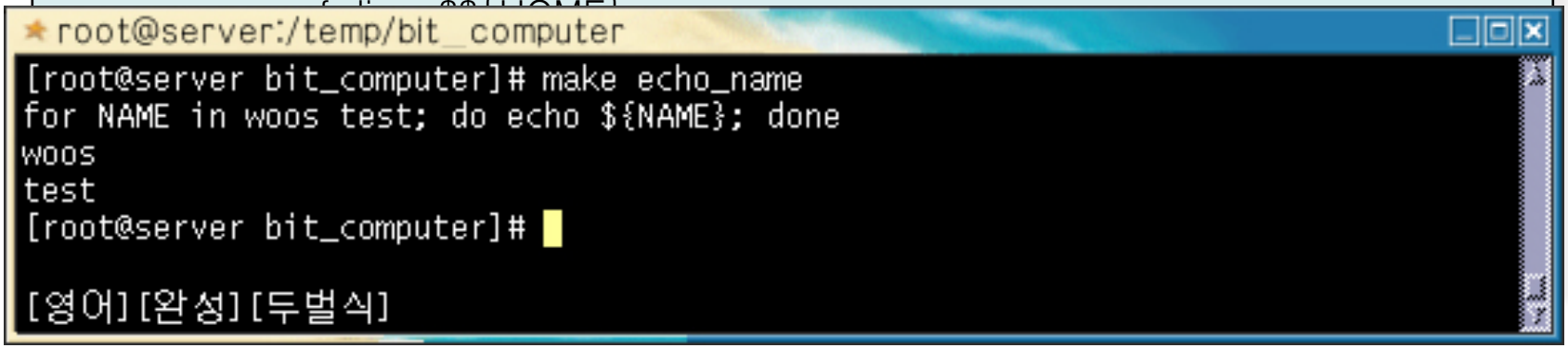
1. 연속된 명령어는 한 줄에 표현해야 한다.

```
del  :
            cd ./backup
            rm -rf *

# 원래 의도는 ./backup 디렉토리의 모든 파일을 지울려고 하였어나 실제로는
# 현재 디렉토리의 모든 파일을 지우게 된다. 이유는 두 명령어가 다른 shell에서
# 실행되기 때문이다. 그래서 원래 의도대로 실행 시킬려면 다음과 같이
# 세미콜론(;)을 이용하여 한줄에 작성하여야 한다.

del :
            cd ./backup; rm -rf *
```

명령어 사용 규칙 (2/4)

2. shell 변수의 참조에서는 두 개의 달러($$) 기호를 사용한다.

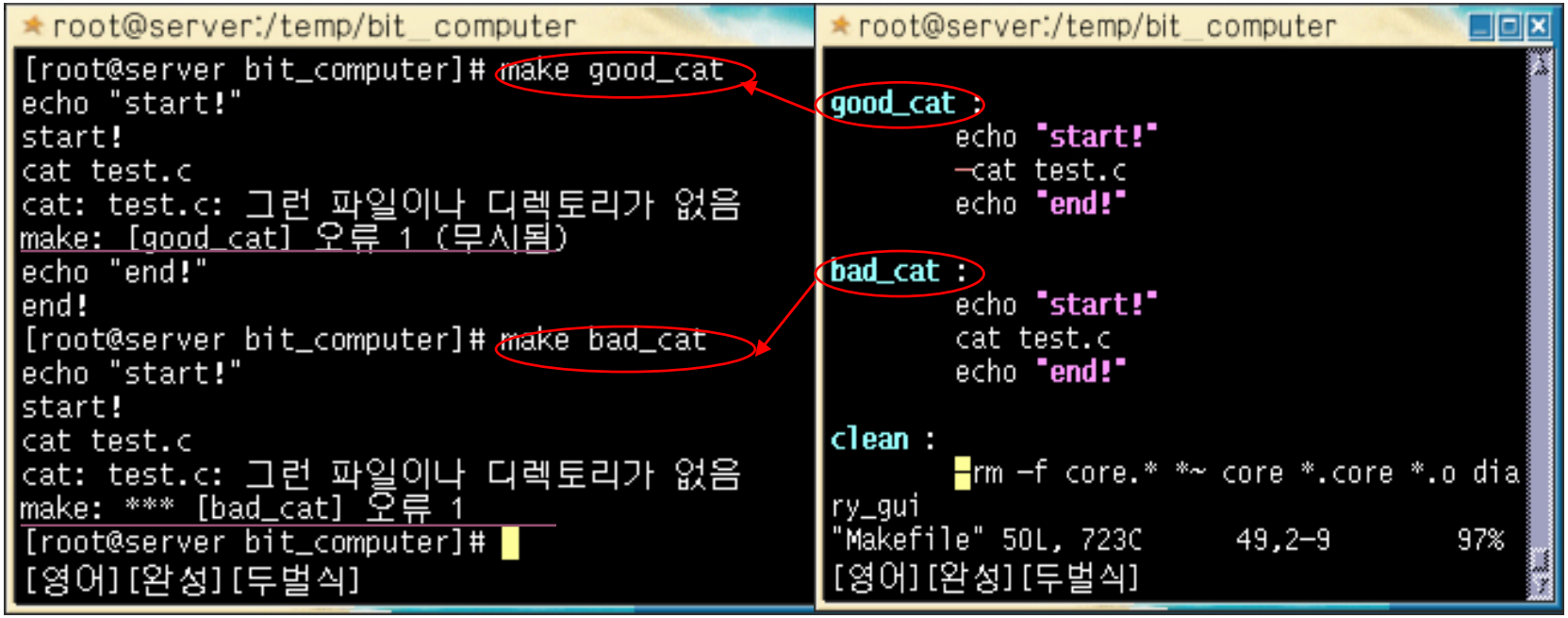
```
echo_name :
        for NAME in woos test; do echo $${NAME}; done
```



명령어 사용 규칙 (3/4)

3. 명령어 수행에 오류가 발생해도 make를 종료하지 않고 계속 수행 하고자 할 시에는 명령어 앞에 마이너스(-)를 붙인다.

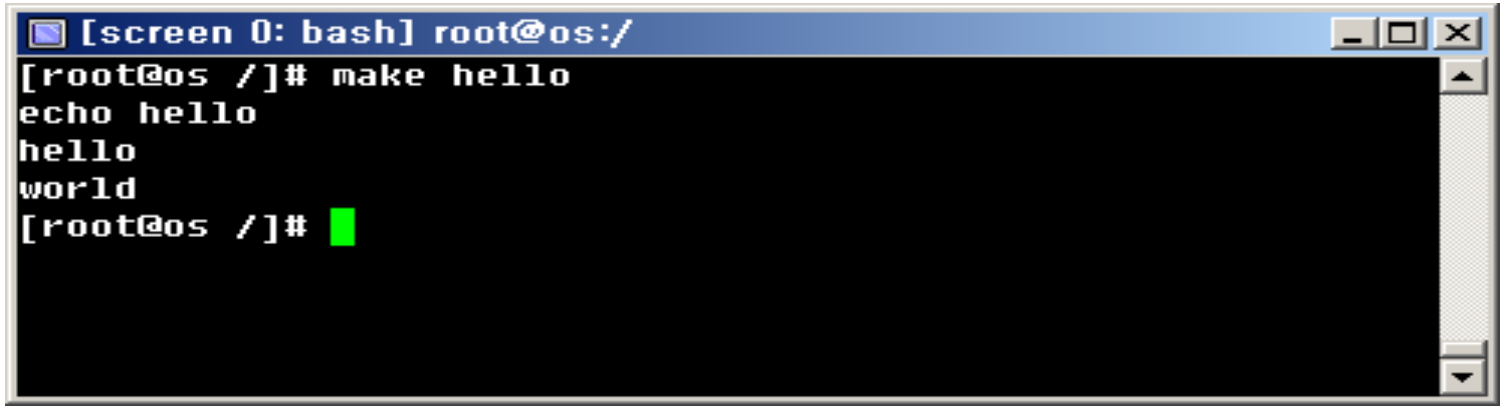
```
# make는 명령어 수행 후 리턴 값이 0이 아니라면 종료하게 되어있다. 명령어가 정상적으로 처리 되지
# 않았더라도 make의 수행을 계속하길 원한다면 명령어 앞에 '-'를 붙이면 된다.
echo_name :
        -cat ./test.c
```



명령어 사용 규칙 (4/4)

4. 화면에 수행 할 명령어를 출력해주는 명령어 에코 기능을 끄기 위해서는 명령어 앞에 @을 붙인다.

```
# make는 기본적으로 다음 수행 할 명령어를 출력해주는 명령어 에코 기능이 활성화되어 있는 상태임
# 으로 명령어가 수행 될 때마다 화면에 출력해준다. 이러한 명령어 에코  기능을 끄기 위해서는 명령어
# 앞에 '@'를 붙인다.


hello  :
        echo hello
        @echo world
```



Dummy target의 사용

☞Dummy target이란?

– 일반적으로 target으로 지정되는 것들은 생성될 파일들을 말하지만 dummy target은 생성되지 않는 개념적인 target이다.

☞Dummy target 사용의 예

```
all : diary_gui

diary_gui : graphic_ui.o db.o memo.o calendar.o
        ${CC} -o $@ $?

install : all
                cp ./diary_gui /usr/local/bin

clean :
        -rm -f core.* *~ core *.core *.o diary_gui
```

여기서 all과 install, clean이 dummy target이다.
all과 install,clean은 명령어 수행을 위한 개념적인 target이다.

이러한 dummy target의 활용은 make를 훨씬 유연하게 만든다.

여러 디렉토리에 재귀적 make 사용

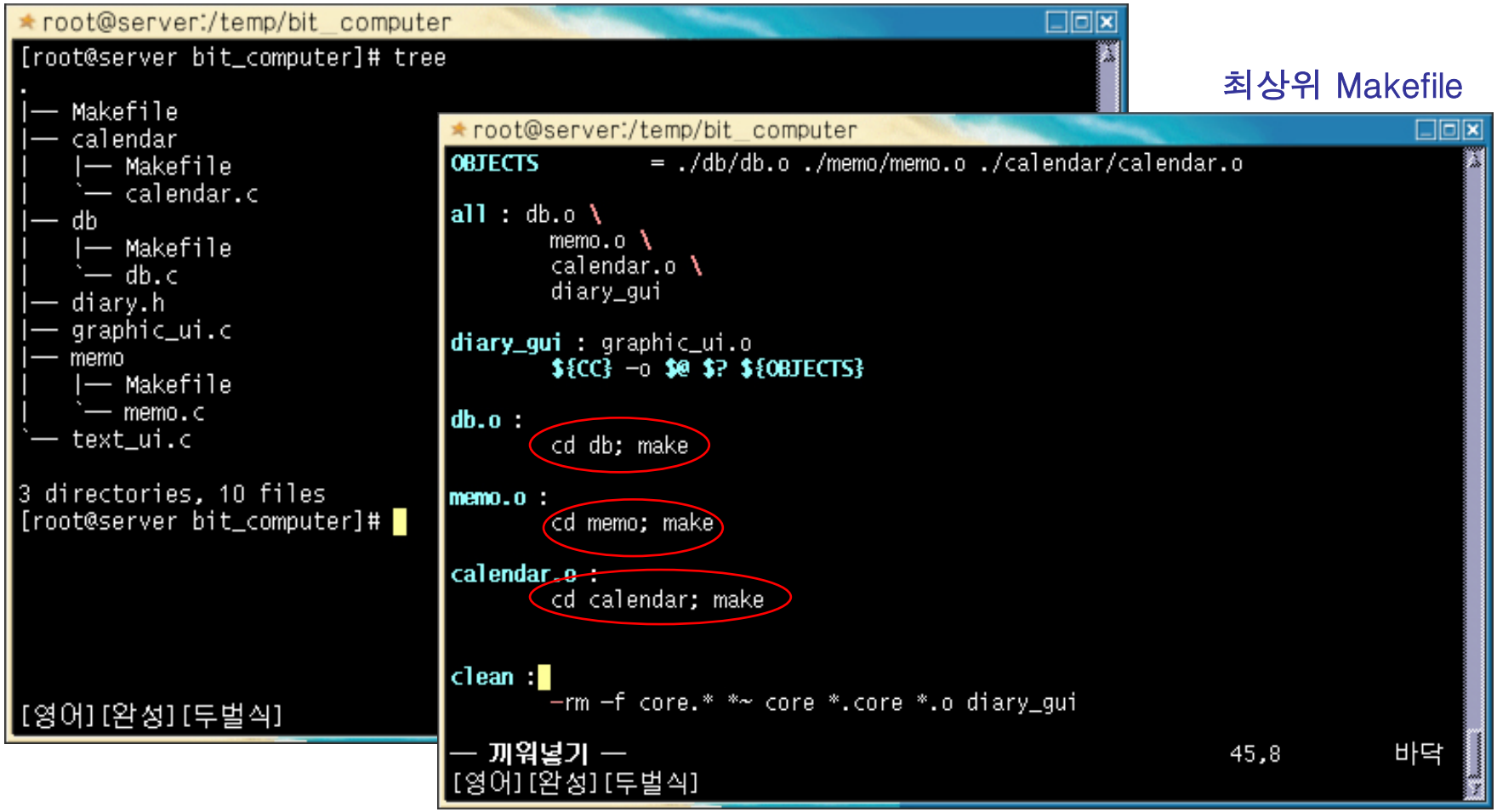
☞make를 사용한 프로젝트 관리

- 일반적으로 프로젝트 파일들은 여러 개의 디렉토리로 이루어져 있는 경우가 대부분이다.

  이러한 여러 디렉토리를 make로 관리하는 기법에는 기술 파일 내부에 경로를 지정하는 방법과 재귀적인 make 사용법이 있다. 경로를 지정하는 방법 보다는 재귀적 make의 사용이 훨씬 효율적이고 유연하기 때문에 재귀적 make 사용법을 알아보기로 한다.

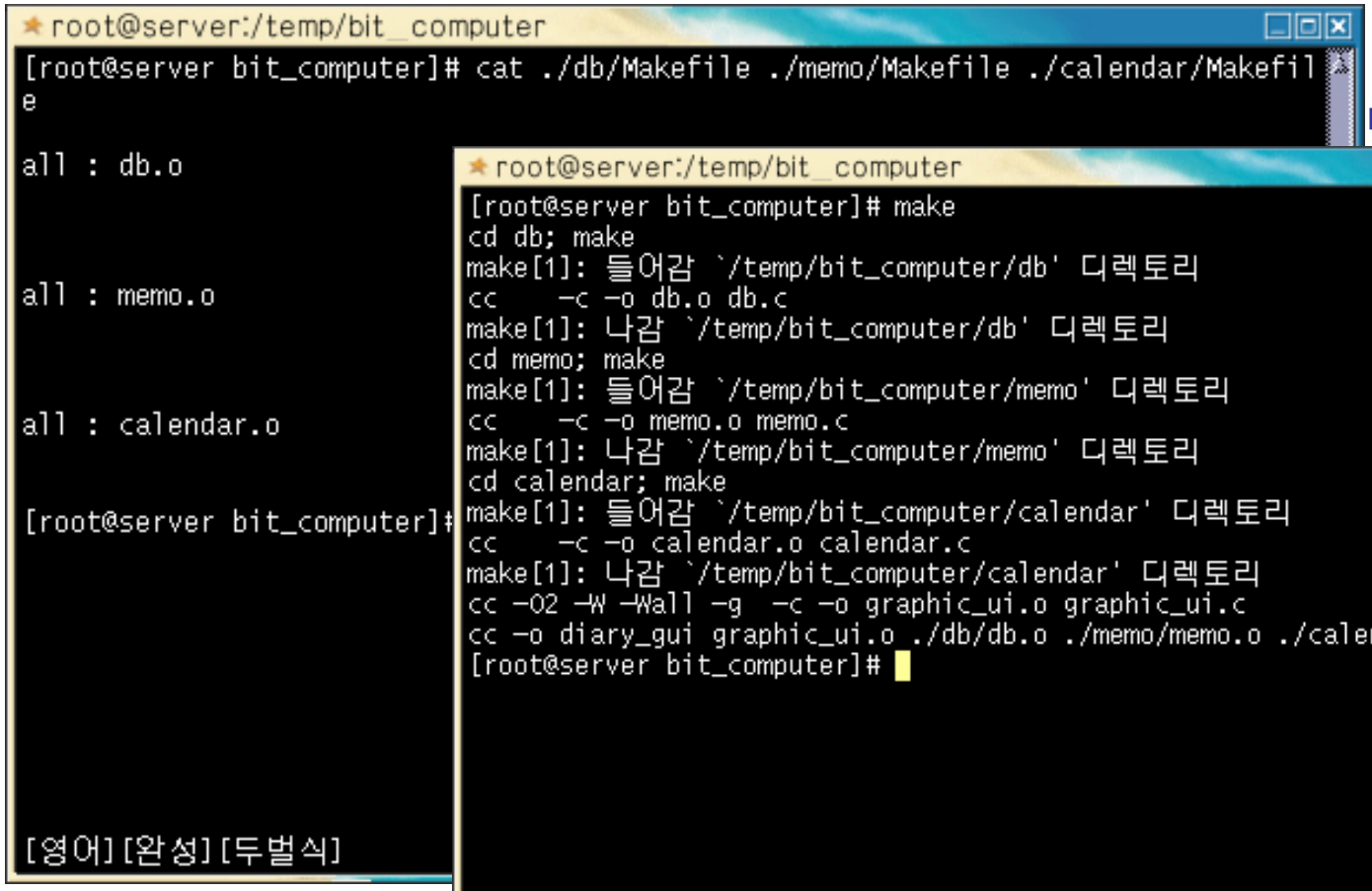
재귀적 make 사용의 예 (1/3)

디렉토리 및 파일구조



재귀적 make 사용의 예 (2/3)

각 디렉토리의 Makefile



make 명령어 수행 모습

재귀적 make 사용의 예 (3/3)

sub Makefile에 대한 매크로 전달

```
OBJECTS  = ./db/db.o ./memo/memo.o ./calendar/calendar.o
export CC = gcc                    # 각 서브 디렉토리로 CC 매크로 정의가 전달 된다.

      all : MEMO CALENDAR MAIN diary
MEMO :
        $(MAKE) -C memo

      CALENDAR :
        $(MAKE) -C calendar

DB :
        $(MAKE) -C db

diary : $(OBJECTS)
        $(CC) -o $@ $^

      clean :
        cd memo && $(MAKE) clean
        cd calendar && $(MAKE) clean
        cd main && $(MAKE) clean
        -rm -rf *.o diary
```

조건부 수행 (1/2)

1. ifeq ~ else ~ endif 문의 사용

```
all :
ifeq ($(CC),gcc)                    # 내부 정의 매크로 CC는 cc로 되어 있다.
        @echo "C 컴파일러는 GNU gcc 입니다."
else
        @echo "C 컴파일러는 $(CC) 입니다"    # 이 부분이 수행된다.
endif
```

2. ifneq ~ else ~ endif 문의 사용

```
all :
ifneq ($(CC),gcc)              # 만약 $(CC)가 gcc와 같지 않다면?
        @echo "C 컴파일러는 GNU gcc 입니다."        # 이 부분이 수행된다.
else
        @echo "C 컴파일러는 $(CC) 입니다."
endif
```

조건부 수행 (2/2)

3. ifdef ~ else ~ endif 문의 사용

```
all :
ifdef CC                              # 만약 CC가 정의되어 있다면?
        @echo "CC 매크로는 정의되어 있습니다."          # 이 부분이 수행된다.
else
        @echo "CC 매크로는 정의 되지 않았습니다."
endif
```

4. ifndef ~ else ~ endif 문의 사용

```
all :
ifndef NO_DEFINE                      # 만약 $(NO_DEFINE)이 정의되어 있다면?
        @echo "NO_DEFINE 매크로는 정의 되지 않았습니다."
else
        @echo "NO_DEFINE 매크로는 정의 되었습니다."
endif
```

make의 옵션 정리

-d	: 디버거 모드. 내부 flag와 파일들의 마지막 변경시간을 상세하게 출력한다.
-e	: macro와 환경 변수의 이름이 같을 때 환경 변수를 우선시 한다.
-f	: 뒤에 오는 기술 파일을 적용한다. 기술 파일의 이름은 어떠한 것이라도 상관없다.
-i	: 오류 코드 무시. 오류가 발생해도 중단하지 않는다.
-k	: 오류가 발생하면 오류가 발생한 파일에 종속된 부분만 중단하고 계속 수행한다.
-n	: 명령 행을 화면에 표시하고 실제로 수행하지는 않음
-p	: macro의 정의, 확장자 규칙 등을 자세하게 출력한다.
-s	: 명령 행을 화면에 표시하지 않는다.
-t	: 어떠한 명령도 실행하지 않고 target 파일들의 날짜만 현재로 바꾼다.
-w	: 재귀적 make 사용을 추적하여 명령을 화면에 출력한다.
-r	: 기본 규칙들을 사용하지 않는다.

make의 특별한 target

.DEFAULT
: make가 요청된 target을 build하기 위한 기술 파일 항목이나 확장자 규칙들을 찾지 못했을 때 이 target에 관련된 명령어가 실행된다.

.IGNORE
: 오류 코드를 무시한다. -i 옵션과 동일하다.

.PRECIOUS
: 이 target으로 지정한 파일들은 build를 중단하라는 신호를 보내거나, 기술 파일의 명령 행에서 오류를 반환하더라도 삭제 되지 않는다.

.SILENT
: 명령은 실행 하지만 실행되는 명령 string을 화면에 출력하지 않는다. -s 옵션과 동일하고 명령어 앞에 '@'를 붙인 것과 동일하다.

.SUFFIXES
: 이 target과 연관된 필요 항목이 make에 중요한 확장자 이며, 확장자 규칙과 연관될 수 있다.

kernel Makefile 분석 기초 (1/7)

```
ifeq (.config,$(wildcard .config))
include .config
endif
```

1. ifeq는 “만약 같다면”의 의미이고 ifeq (a,b)와 같은 형태로 사용한다.
2. $(wildcard .config)에서 “wildcard”는 make 내부 확장 규칙으로써 “wildcard” 뒤에 오는 string을 wildcard 확장한다는 의미이다. 그래서 $(wildcard .config)의 의미는 현재 디렉토리의 .config 파일을 의미한다.
   또한 $(wildcard *.c)의 의미는 현재 디렉토리의 모든 C 소스 파일을 의미한다.
3. include는 현재 위치에 뒤에 오는 파일의 내용을 추가 함을 의미

그러므로 위 make 기술 파일의 의미는 다음과 같다.

“ 만약 현재 디렉토리에 .config 파일이 존재한다면 .config 파일의 내용을 현재 위치에 추가하라.”

kernel Makefile 분석 기초 (2/7)

```
ifneq (.config,$(wildcard .config))
include .config
endif
```

1. ifneq는 “만약 같지 않다면”의 의미이고 나머지는 앞에서 설명한 것과 동일하다.

kernel Makefile 분석 기초 (3/7)

```
ifeq ($(wildcard .depend),)
include .depend
endif
```

만약 현재 디렉토리에서 .depend wildcard 확장하였을 때 NULL이라면…
결국 현재 디렉토리에 .depend 파일이 없다면 의미

kernel Makefile 분석 기초 (4/7)

```
TOPDIR     := $(shell /bin/pwd)
```

1. shell은 make 내부 확장 규칙 중 하나로써 shell에서 `` (back quote)와 같은 의미이다.

```
TOPDIR := `/bin/echo test!`
${TOPDIR}
```

내부적으로 `/bin/echo test!`로 치환된다.

```
TOPDIR1 := $(shell /bin/echo test!)
${TOPDIR1}
```

내부적으로 test!로 치환된다.

kernel Makefile 분석 기초 (5/7)

```
DRIVERS-y = test.o

DRIVERS-y += serial.o ₩
             char.o
```

DRIVERS-y macro에 serial.o와 char.o를 추가 하겠다는 의미이다.
그래서 DRIVERS-y가 치환되어 사용되어질 때는 “test.o serial.o char.o”의 의미를 갖는다.

kernel Makefile 분석 기초 (6/7)

```
SUBDIRS = kernel drivers mm fs net

$(patsubst %, _modinst_%, $(SUBDIRS))
```

patsubst는 3개의 인자를 가지는데 처음 인자는 반드시 '%'가 와야 하고 두번째 인자는 반복되는 string을 의미하며 두번째 인자의 '%' 기호는 세번째 인자로 치환된다.
위의 $(patsubst %, _modinst_%, $(SUBDIRS))는 다음과 같이 치환된다.

_modinst_kernel _modinst_drivers _modinst_mm _modinst_fs _modinst_net

kernel Makefile 분석 기초 (7/7)

obj-y or subdir-y
: kernel에 static하게 추가되는 object 또는 sub 디렉토리

obj-m or subdir-m
: module object 또는 sub 디렉토리

obj-n or subdir-n
: 아무일도 하지 않음

O_TARGET
: obj-y를 링크하여 만드는 최종 target

export-objs
: export 하는 object. 여러 object에 같이 링크되는 object를 말함