**make 프로그램 기초 설명**

makefile을 서명하기에 앞서 make 프로그램에 대해 먼저 설명하겠습니다. make 프로그램은 소스 파일을 이용해서 자동으로 실행 파일 또는 라이브러리 파일을 만들어주는 빌드 관련 유틸리티입니다. make 프로그램은 소스 파일과 목적 파일을 비교한 뒤 마지막 빌드 후에 수정된 파일만 선택하여 빌드를 수행하므로 빌드 시간을 크게 줄여줍니다. 또한 빌드를 편리하게 해주는 여러 가지 문법과 규칙을 지원하므로 이를 활용하면 많은 수의 소스 파일도 한 번에 빌드할 수 있습니다. 하지만, make 프로그램도 만능이 아닙니다. make 프로그램이 빌드를 자동으로 수행하려면 각 소스 파일의 의존 관계나 빌드 순서, 빌드 옵션 등에 대한 정보가 필요합니다. 이러한 내용이 저장된 파일이 바로 makefile입니다.

make 문법

make의 문법은 복잡하고 다양합니다.

[make의 기본 문법]

Target: Dependency

<tab> command

<tab> command

Target은 일반적으로 생성할 파일을 나타내며, 특정 레이블(label)을 지정하여 해당 레이블과 관련된 부분만 빌드하는 것도 가능합니다. Dependency는 Target 생성에 필요한 소스 파일이나 오브젝트 파일 등을 나타내고, command는 Dependency에 관련된 파일이 수정되면 실행할 명령을 의미합니다. command에는 명령창이나 터미널에서 실행할 명령 또는 프로그램을 기술합니다. <tab>으로 표시한 부분은 반드시 TAB 문자로 띄워야 합니다. 공백으로 탭 문자를 대체하면 make가 정상적으로 실행되지 않으니 주의해야 합니다.

[간단한 makefile 예제]

# a.c, b.c를 통해서 output.exe 파일을 생성하는 예제 <- 주석 (comment)

all: output.exe <- 별다른 옵션이 없을 떄 기본적으로 생성하는 target을 기술

a.o: a.c

gcc –c a.c

b.o: b.c

gcc –c b.c

output.exe: a.o b.o

gcc –o output.exe a.o b.o

make는 최종으로 생성할 Taget의 의존성을 추적하면서 빌드를 처리하기 때문에 makefile은 역순으로 따라가면 됩니다.

make는 빌드를 수행하는 도중에 다른 make를 실행할 수 있습니다. 이는 빌드 단계를 세부적으로 나누고, 계층적으로 수행할 수 있음을 의미합니다. 최상위 디렉터리의 하위에 Library 디렉터리가 있고, 빌드 과정에서 Library 디렉터리를 빌드해야 한다면 –C 옵션을 사용해서 다음과 같이 간단히 처리할 수 있습니다.

[계층적 빌드]

all: output.exe

# Library 디렉터리로 이동한 후 make를 수행

libtest.a:

make –C Library

output.o: output.c

gcc –c output.c

output.exe: libtest.a output.o

gcc –o output.exe output.c –ltest –L./

**automate variable**

$@: target

$<: 첫 번째 dependency

$^: 나열된 모든 dependency 의미

ex) all: f1.o f2.o 일떄

$@ : all

$< : f1.o

$^ : f1.o f2.o

**recursive make**

만약, 자기 자신의 makefile을 수행하고 싶고, working directory만 바꾸고 싶을 경우 다음과 같은 명령어를 실행하면 된다.

make –C $(목적 directory) –f ../makefile 목적 Target

즉, 목적 directory로 cd한 뒤, 현재 수행중인 makefile의 목적 Target을 수행하는 것이다.

**Target이 없는 명령어**

make는 기본적으로

Target: dependency

명령어

로 구성되지만 Target이 없고 tab으로 띄워지지 않게

include dependency.dep 와 같은 명령어가 존재할 수 있다. 해당 명령어는 맨 처음 makefile을 실행할 때 수행된다.

다음과 같이 조건문을 추가해도 된다.

ifeq (dependency.dep $(wildcard dependency.dep))

include dependency.dep

endif

Generating Prerequisites Automatically !

해당 기능을 사용하는 정교한 makefile을 만들기 위해서는 다음과 같은 사항들을 이해하고 있어야 한다.

1. include

2. multiple rules for one target

3. header file dependency

4. pattern match

1. include

The include directive tells make to suspend reading the current makefile and read one or more other makefiles before continuing. 단순히 –MM 옵션의 header file dependency를 include 하는 경우 target: dependency를 makefile에 추가하는 거라고 생각해도 된다. 그냥 include의 경우 해당하는 dependency 파일이 없을 경우 오류가 발생하게 된다. 따라서 condition 명령어를 만들거나 –include을 사용해야 한다.

ex) 1. condition

ifeq (dependency.dep $(wildcard dependency.dep))

include dependency.dep

endif

2.

-include dependency.dep

2. multiple rules for one target

One file can be the target of several rules. All the prerequisites mentioned in all the rules are merged into one list of prerequisites for the target. If the target is older than any prerequisite from any rule, the recipe is executed.

There can only be one recipe to be executed for a file. If more than one rule gives a recipe for the same file, make uses the last one given and prints an error message.

즉, 한 target에 여러 개의 rule이 있고, 그 rule마다 명령어가 존재하는 경우, 마지막 명령어만 실행되게 된다. 또한, 다음과 같은 warning이 발생하게 된다. makefile:6: warning: ignoring old commands for target `f1.o'

단 %.o: %.c와 같은 패턴 매치의 경우 include가 있을 경우, include의 .o파일은 패턴 매치 되지 않고 include의 dependency를 따르게 된다. 따라서, include에 –MM 옵션 dependency 다음에 명령어가 나타나는 경우, %.o:%.c 의 명령어가 실행되지 않는다. 반면, include에 –MM 옵션 dependency 만 존재하는 경우 %.o:%.c의 명령어가 실행된다.

ex)

[dependency.dep]

f1.0 : f1.c f1.h

[makefile]

%.o : %.c

@echo $@ $^

@echo hello

결과:

f1.0 : f1.c f1.h

hello

[dependency.dep]

f1.0 : f1.c f1.h

@echo dependency

[makefile]

%.o : %.c

@echo $@ $^

@echo hello

결과:

dependency

3. headerfile dependency

gcc 옵션 중에 makefile용 규칙을 만들어주는 전처리기 관련 옵션(-M)을 사용하면, 자동으로 헤더 파일을 추출할 수 있다.

-MM : stdio.h와 같은 시스템 헤더 파일을 제외한 나머지 헤더 파일에 대한 의존 관계를 출력해준다.

4. pattern match

A pattern rule contains the character ‘%’ (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule. The target is a pattern for matching file names; the ‘%’ matches any nonempty substring, while other characters match only themselves.

For example, ‘%.c’ as a pattern matches any file name that ends in ‘.c’. ‘s.%.c’ as a pattern matches any file name that starts with ‘s.’, ends in ‘.c’ and is at least five characters long. (There must be at least one character to match the ‘%’.) The substring that the ‘%’ matches is called the stem.

‘%’ in a prerequisite of a pattern rule stands for the same stem that was matched by the ‘%’ in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration and all of its prerequisites (after pattern substitution) must name files that exist or can be made. These files become prerequisites of the target.

Thus, a rule of the form

%.o : %.c ; recipe…

specifies how to make a file n.o, with another file n.c as its prerequisite, provided that n.c exists or can be made.

[확장자가 .c인 파일을 .o로 변경하는 룰을 지정하는 예]

%.o : %.c

gcc –c $<

정교하게 작성된 makefile 예제

1 ###############################################################

2 # 빌드 환경 및 규칙 설정

3 ###############################################################

4 NASM32 = nasm

5 GCC32 = x86\_64-pc-linux-gcc -c -m32 -ffreestanding

6 LD32 = x86\_64-pc-linux-ld -melf\_i386 -T ../elf\_i386.x -nostdlib -e main -Ttext 0x10200

7 OBJCOPY32 = x86\_64-pc-linux-objcopy -j .text -j .data -j .rodata -j .bss -S -O binary

8

9 OBJECTDIRECTORY = Temp

10 SOURCEDIRECTORY = Source

11

12

13 ###############################################################

14 # 빌드 항목 및 빌드 방법 설정

15 ###############################################################

16

17

18 all : prepare Kernel32.bin

19

20

21 prepare:

22 mkdir -p $(OBJECTDIRECTORY)

23

24 $(OBJECTDIRECTORY)/EntryPoint.bin: $(SOURCEDIRECTORY)/EntryPoint.s

25 $(NASM32) -o $@ $<

26

27 dep:

28 @echo === make dependency file ===

29 make -C $(OBJECTDIRECTORY) -f ../makefile InternalDependency

30 @echo === depency search complete ===

31

32 ExecuteInternalBuild: dep

33 make -C $(OBJECTDIRECTORY) -f ../makefile Kernel32.elf

34

35 $(OBJECTDIRECTORY)/Kernel32.elf.bin: ExecuteInternalBuild

36 $(OBJCOPY32) $(OBJECTDIRECTORY)/Kernel32.elf $@

37

38 Kernel32.bin: $(OBJECTDIRECTORY)/EntryPoint.bin $(OBJECTDIRECTORY)/Kernel32.elf.bin

39 cat $^ > $@

40

41 clean:

42 rm -f \*.bin

43 rm -f $(OBJECTDIRECTORY)/\*.\*

44

45 ###############################################################

46 # make에 의해 다시 호출되는 부분, Temp 디렉터리를 기준으로 수행됨

47 ###############################################################

48

49 CENTRYPOINTOBJECTFILE = main.o

50 CSOURCEFILES = $(wildcard ../$(SOURCEDIRECTORY)/\*.c)

51 ASSEMBLYSOURCEFILES = $(wildcard ../$(SOURCEDIRECTORY)/\*.asm)

52 COBJECTFILES = $(subst main.o, , $(notdir $(patsubst %.c,%.o,$(CSOURCEFILES))))

53 ASSEMBLYOBJECTFILES = $(notdir $(patsubst %.asm,%.o,$(ASSEMBLYSOURCEFILES)))

54

55 # .c 파일을 .o 파일로 바꾸는 규칙 정의

56 %.o: ../$(SOURCEDIRECTORY)/%.c

57 @echo $@ $^

58 $(GCC32) -c $<

59

60 # .asm 파일을 .o 파일로 바꾸는 규칙 정의

61 %.o: ../$(SOURCEDIRECTORY)/%.asm

62 @echo %.o %.c

63 $(NASM32) -f elf32 -o $@ $<

64

65 InternalDependency:

66 $(GCC32) -MM $(CSOURCEFILES) > dependency.dep

67

68 Kernel32.elf: $(CENTRYPOINTOBJECTFILE) $(COBJECTFILES) $(ASSEMBLYOBJECTFILES)

69 $(LD32) -o $@ $^

70

71

72 #ifeq (dependency.dep, $(wildcard dependency.dep))

73 include dependency.dep

74 #endif

해당 makefile의 flow을 분석하면

ifeq문 false -> all -> target -> mkdir –p $(OBJECTDIRECTORY) -> kernel32.bin -> $(OBJECTDIRECTORY)/EntryPoint.bin -> ExecuteInternalBuild -> dep -> @echo === Make dependency File === -> make –C $(OBJECTDIRECTORY) –f .../makefile InternalDependency (makefile 다시 실행) -> ifeq문 false -> InternalDependency -> $(GCC32) –MM $(CSOURCEFILES) -> dependency.dep -> @echo === dependency search complete === -> make –C $(OBJECTDIRECTORY) –f ../makefile Kernel32.elf -> ifeq문 true -> include dependency.dep -> $(CENTRYPOINTOBJECTFILE) ->

%.0: ../$(CSOURCEDIRECTORY/%.c or dependency.dep의 dependency -> $(GCC32) –c $< (dependency.dep에는 명령어가 없기 때문에) -> $(COBJECTFILES) ->%.0: ../$(CSOURCEDIRECTORY/%.c or dependency.dep의 dependency -> $(GCC32) –c $< (dependency.dep에는 명령어가 없기 때문에) -> $(ASSEMBLYOBJECTFILES) -> $.o:../$(SOURCEDIRECTORY)/%.asm -> $(NASM32) –f elf32 –o $@ $< -> $(LD32) –o $@ $^ -> cat $^ > $@