

제 2 장

프로그래밍 개발 환경

ACS30021
고급 프로그래밍

나보균 (bkna@kpu.ac.kr)

컴퓨터 공학과
한국산업기술 대학교

학습 목표

□ 팀 단위 프로그램 개발 환경

- ✓ ar
- ✓ lib.so/lib.a 차이점
- ✓ nm
- ✓ makefile
- ✓ version control
- ✓ autoconf
- ✓ automake

2.1 라이브러리

- ❑ 라이브러리
 - ✓ 미리 컴파일된 오브젝트들의 집합 – 함수와 변수
- ❑ 정적 (archive library)
 - ✓ 정적 링크 – 컴파일 시 링크
- ❑ 공유 (shared library)
 - ✓ 동적 링크 – 바이너리 코드가 실행되는 순간에 링크
- ❑ 예:
 - ✓ `gcc -c file1.c file2.c like.c`
 - ✓ `gcc -o like file1.o file2.o like.o`

```
// file1.c

#include <stdio.h>

void func1 ()
{
    printf ("Hello func1\n");
}

void func2 ()
{
    printf ("Hello func2\n");
}
```

```
// file2.c

#include <stdio.h>

void func3 ()
{
    printf ("Hello func3\n");
}
```

```
// like.c

#include <stdio.h>

extern void func1();
extern void func2();
extern void func3();

int main ()
{
    func1 ();
    func2 ();
    func3 ();

    return 0;
}
```

라이브러리 기호표 검색(Library symbol table)

❑ nm

- ✓ ELF 목적 파일 포맷으로 만들어진 파일의 내부 심볼 출력
- ✓ nm libfunc.a 의 결과:
 - 첫째 필드 - 메모리에 로드될 때 심볼(함수나 변수)의 주소
 - 둘째 필드 - 심볼의 종류
 - 셋째 필드 - 심볼의 이름

❑ ranlib

- ✓ 정적 라이브러리 안에 인덱스 생성
- ✓ 함수 간 의존성 독립

❑ readelf -a libmyfunc.so.0.0.0

- ✓ ELF 포맷의 바이너리 파일 속성(object code에는 coef와 elf 형식 존재)
- ✓ 이 라이브러리가 필요로 하는 다른 라이브러리들 soname의 정보 표시

❑ ldd 실행파일명

- ✓ 실행 파일이 요구하는 공유라이브러리 목록을 출력

❑ objcopy

- ✓ 목적파일을 다른 목적파일로 복사
- ✓ ELF헤더가 제거된 순수 바이너리 파일 생성
- ✓ 부트로더 생성시 이용

라이브러리 작성법 – 정적 라이브러리 (lib???.a)

- ❑ libmyfunc.a를 생성하기 위해
 - ✓ gcc -c file1.c file2.c
 - ✓ ar rscv libmyfunc.a file1.o file2.o
 - ar 명령의 옵션
 - s - 아카이브 파일에 인덱스 생성
 - s 옵션이 없으면 ranlib libmyfunc.a 명령으로 인덱스 생성
- ❑ 실행파일 완성
 - ✓ gcc -o like like.c -L./ -lmyfunc
- ❑ ar t libfunc.a: 라이브러리에 있는 파일 리스트 출력
- ❑ ar rus libfunc.a obj_1 obj_2: libfunc.a에 obj_1 obj_2 추가

라이브러리 작성법 – 공유 라이브러리 (lib???.so)

- ❑ libmyfunc.so를 생성하기 위해
 - ✓ `gcc -shared -fPIC -o libmyfunc.so file1.c file2.c`
- ❑ 라이브러리를 포함하는 실행파일 작성
 - ✓ `gcc -o like like.c -L./ -lmyfunc`
- ❑ 개인적 생성 동적 라이브러리 적재
 1. LD_LIBRARY_PATH 환경변수
 - ✓ ~/.bash_profile 에 다음 추가
 - export
LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOME/class/ACS30021/practice/ch02/lib
 2. /etc/ld.so.conf 파일에 지정 후 ldconf 실행
 - ✓ 리눅스 Kernel 2.6 이후부터 /etc/ld.so.conf.d/*.conf 파일 형식으로 설정
 - ✓ 프로그램이 실행될 때 기본적으로 /lib과 /usr/lib에서 so파일을 검색

라이브러리 작성법 – 공유 라이브러리 (lib???.so)

1. gcc -fPIC -c file1.c file2.c

- ✓ -fPIC : 컴파일러에게 위치에 관계없이 수행되는 코드로 컴파일
- ✓ 프로세스의 가상 주소 공간 안의 다른 지역에 로드될 수 있도록 코드를 컴파일
- ✓ 실행 될 때 그 위치가 정해지기 때문에 가상 공간 안에 고정된 위치를 가지면 안 된다.

2. gcc -shared -W1, -soname,libmyfunc.so.0 \n -o libmyfunc.so.0.0.0 file1.o file2.o

- ✓ -shared : file1.o와 file2.o를 링크할 때 정적라이브러리와 동적라이브러리가 동시에 존재하면 동적라이브러리를 사용 (참조: 정적 라이브러리 -static 옵션)
- ✓ -W1 : 이 옵션 뒤에 오는 옵션들을 링커에게 gcc 컴파일러를 거치지 않고 직접 전달
 - -W1, -soname,libmyfunc.so은 링커에게 -soname libmyfunc.so로 전달
 - -soname : 동적 링크 (/lib/ld-linux.so.2)가 공유라이브러리를 찾을 때 libmyfunc.so.0.0.0를 찾는게 아니라 libmyfunc.so.0를 이용

① 또는 위의 1과2를 합쳐서

gcc -shared -fPIC -o libmyfunc.so.0.0.0 file1.c file2.c

② ln -s libmyfunc.so.0.0.0 libmyfunc.so.0

ln -s libmyfunc.so.0.0.0 libmyfunc.so

③ ldconf

- ✓ 공유라이브러리의 위치를 /etc/ld.so.conf 파일에 추가
- ✓ /etc/ld.so.cache 파일 갱신

실습: file1.c file2.c file3.c를 라이브러리(.a와 .so)로 만들어 실행

```
// file1.c

#include <stdio.h>

void func1 ()
{
    fprintf (stdout, "Function 1\n");
}
```

```
// file2.c

#include <stdio.h>

void func2 ()
{
    fprintf (stdout, "Function 2\n");
}
```

```
// file3.c

#include <stdio.h>

void func3 ()
{
    fprintf (stdout, "Function 3\n");
}
```

```
// fun.h

#ifndef __FUN_H
#define __FUN_H

extern void func1();
extern void func2();
extern void func3();

#endif
```

```
// main.c

#include "fun.h"
#include <stdio.h>

int main (int argc, char **argv)
{
    fprintf (stdout, "main 함수:: \n");
    func1 ();
    func2 ();
    func3 ();

    return 0;
}
```


실습:

1. 만약 `libmyfunc.a` 와 `libmyfunc.so`가 함께 있다면?
 - ✓ 컴파일러는 `libmyfunc.so`를 선택
 - ✓ 명시적 표현은 `-shared` 옵션 사용
2. `libmyfunc.a`를 선택하고 싶다면?
 - ✓ `-static` 추가
3. 실행 파일 `main`을 각각 `main_so`, `main_a`로 생성
 - ✓ 파일 크기는?

라이브러리

□ 특성

	정적 라이브러리 (Archive Library)	공유 라이브러리 (Shared Library)
확장자	.a	.so, .sl(HP), .dll (MS)
라이브러리 로직 변경 시	*.a 교체 후 재 컴파일 후 재실행	*.so 교체 후 컴파일 없이 재실행
작성 방법	ar -rscv	gcc -shared -fPIC -o
실행 속도	상대적 빠름	상대적 느림
	실행 프로세스마다 라이브러리 들이 포함되어 중복적으로 메모 리 공간 소비	<ul style="list-style-type: none">프로세스들이 라이브러리 코드 를 포함하지 않고 커널 공간에 공유하여 필요 시 접근printf() 로 화면에 “Hello!” 라고 출력하는 프로그램을 하나의 서버에 여러 사람이 동시에 실행 할 때 메모리 공간 절약

2.2 Make 정의

□ 정의

- ✓ 파일 관리 유틸리티
- ✓ 일종의 셸 스크립트
- ✓ 각 파일간의 종속 관계를 파악하여 Makefile에 기술된 대로 명령을 실행
- ✓ 각 파일에 대한 반복적 명령을 자동화

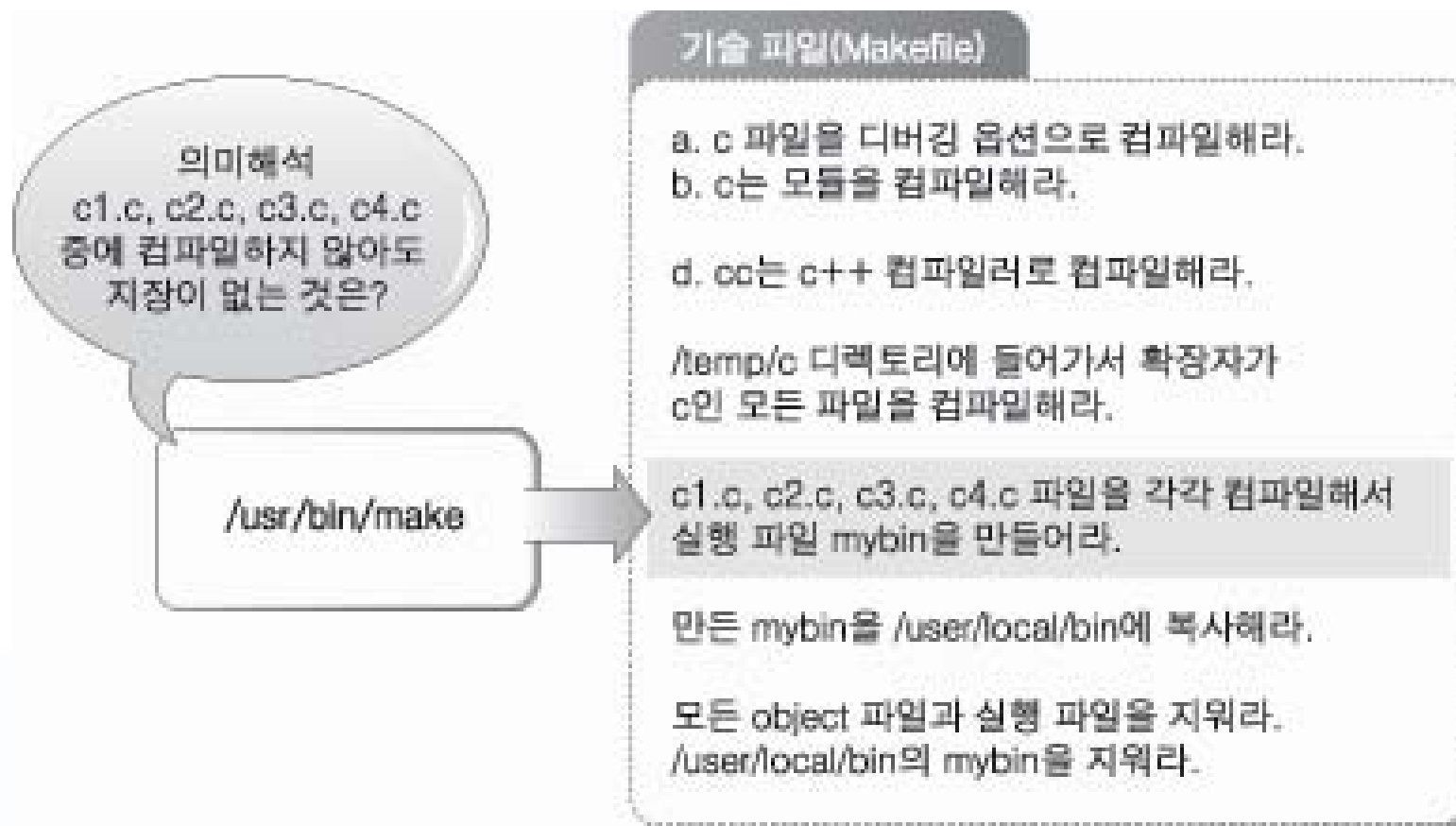
□ 프로젝트 관리

- ✓ 팀 단위 프로그램 개발
- ✓ 개발 중 팀원의 교체
- ✓ 팀원의 경쟁 그룹으로 이동
- ✓ 소스의 유출
- ✓ 설치 과정의 일목요연한 이해

Make와 Makefile의 관계

가정:

1. 하나의 실행 프로그램을 완성
2. 1만개의 소스 파일들로 구성
3. 디버깅 중 매번 1만개의 소스 파일을 컴파일?



2.2.1 Make의 기본 동작

kpu.h	file1.c
<pre>#ifndef __KPU_H #define __KPU_H #include <stdio.h> int func1 (); int func2 (); #endif //__KPU_H</pre>	<pre>#include "kpu.h" int func1 () { printf ("Hello!"); return 0; }</pre>
main.c	file2.c
<pre>#include "kpu.h" int main (int argc, char **argv) { func1 (); func2 (); return 0; }</pre>	<pre>#include "kpu.h" int func2 () { printf (" Dept. of Cybernetics"); return 0; }</pre>

헤더파일 종속 관계

- ❑ 전처리를 이용
- ❑ 중복 포함되는 것을 방지
- ❑ 예,
 - ✓ file1.c에 kpu.h 포함
 - ✓ file2.c에 kpu.h 포함
 - ✓ main.c에 kpu.h 포함
 - ✓ 하지만,
 - ✓ gcc -o prog main.c file1.c file2.c로 컴파일 할 경우
 - ✓ 심볼 테이블에 의거 한번만 포함 됨
- ❑ 전역변수가 헤더에 포함된 경우
 - ✓ 참조: 실습 코드

```
#ifndef __KPU_H
#define __KPU_H

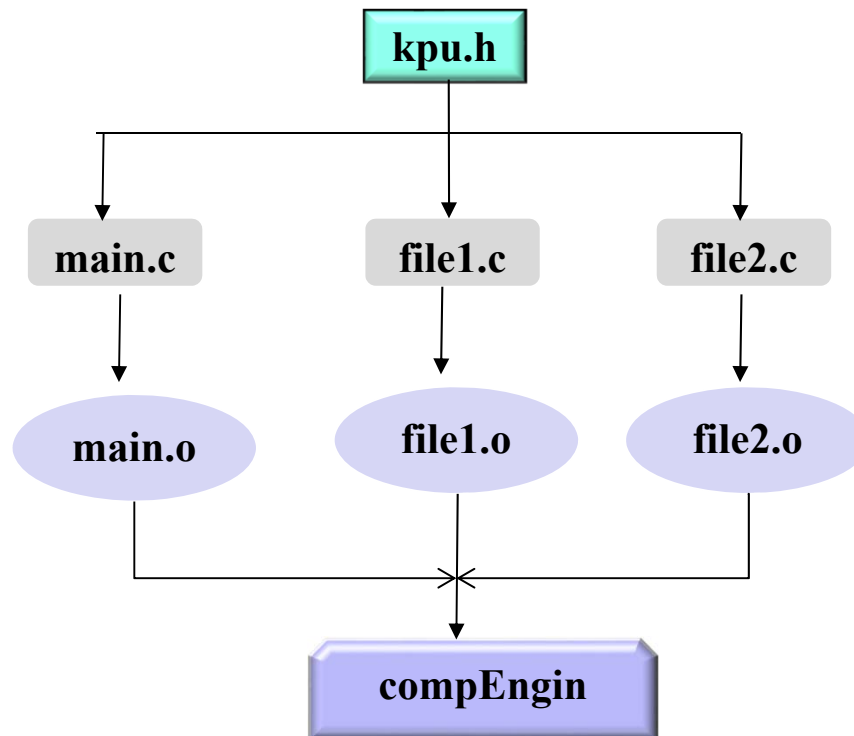
#include <stdio.h>

int func1 ();
int func2 ();

#endif // __KPU_H
```

파일종속 구조

- 실행 파일 compEngin을 만들기 위한 소스 파일들
 - ✓ kpu.h, main.c, file1.c, file2.c



컴파일

□ compEngin을 컴파일 하기 위한 Makefile

```
all : compEngin
```

```
compEngin : main.o file1.o file2.o
```

```
gcc -Wall -o compEngin main.o file1.o file2.o
```



```
main.o : main.c
```

```
gcc -Wall -c -o main.o main.c
```

```
file1.o : file1.c
```

```
gcc -Wall -c -o file1.o file1.c
```

```
file2.o : file2.c
```

```
gcc -Wall -c -o file2.o file2.c
```


make 실행 과정

1. 현재 디렉터리 내에 있는 기술파일(makefile)인 GNUmakefile, makefile, Makefile의 순서로 파일을 검색
2. Makefile 내에서 처음 오는 타겟을 해석
 - ✓ all : compEngin
3. all 타겟을 만들기 위한 종속항 compEngin의 현재 디렉터리에 존재 여부 확인
 - ✓ 처음 컴파일 할 경우 compEngin 파일이 존재하지 않음
 - ✓ 존재하더라도 소스파일이 수정된 시간 보다 앞서면 무시
 - ✓ 위의 경우가 아닌 경우 – 실행 종료
4. compEngin : main.o file1.o file2.o
 - ✓ 종속항의 첫 번째 파일 main.o의 존재 여부 검사
 - 없거나 소스파일 수정 전 생성되었으면, main.o : main.c 항 실행
 - 그 외 – file1.o의 존재 여부 검사로 다음 종속항의 존재 여부 검사 및 컴파일 여부 심사
 - ✓ 반복적 *.o의 존재 여부 검사
5. 위의 단계가 끝나면 compEngin을 만들기 위한 컴파일 실행

2.2.2 Makefile의 작성 방법

- 기본 구조
 1. 매크로 정의
 2. 명령절
 - 규칙문과 명령문으로 구성
- 매크로는 규칙문과 명령문에서 문장 실행 시 치환 됨
- 규칙문의 구성
 - ✓ 타겟 : 종속항1 종속항2 종속항3
 - ✓ 종속항이 없는 규칙문도 가능 – 더미 타겟
- 명령문의 구성
 - ✓ 탭으로 시작
- 공란은 무시됨
- '#' - 주석문
- '\' - 수식이 길면 '\'로 다음행에 계속 이어 표현 가능
- 명령문의 명령어에는 `cc`, `ar`, `mv`, `rm` 등 모든 쉘 명령어 가능

Makefile 기본 구조

CC = gcc

← 매크로 정의

target1 : dependency1 dependency2

← 규칙문 1

command1

← 명령문 1

command2

← 명령문 2

target2 는 기초 실험 프로그램입니다.

← 주석문

target2 : dependency3 dependency4 dependency5

← 규칙문 2

command3


← 명령문 3

command4

← 명령문 4

명령문

- 명령의 시작은 반드시 탭 문자(\t)로 시작
 - ✓ 한 개 이상의 탭 문자 가능

```
target1 : dependency1 dependency2
          command1
           command2
          탭 문자
```

- 공란은 무시

```
CC = gcc

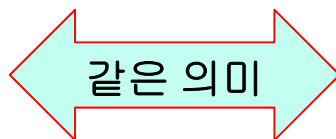
target1 : dependency1 dependency2
          command1
```

- ‘#’를 만나면 개행문자까지 주석문으로 무시

```
# target2 는 기초 실험 프로그램입니다.  
target2 : dependency3 dependency4 dependency5  
          command3  
          command4
```

- 문장의 연결

```
target2 : dependency3 dependency4 dependency5
```



```
target2 : dependency3 dependency4 \  
          dependency5
```

□ 명령문의 명령어

```
target1 : dependency1 dependency2 dependency 3
cp -f file1 file2
mv target1 /usr/share/bin
ar libdep.a dependency1 dependency2
ranlib libdep.a
gcc -o target1 dependency 3 dependency 2
```

□ 위 명령문 중

ar libdep.a dependency1 dependency2

ranlib libdep.a

은

ar rscv libdep.a dependency1 dependency2

로 축약하여 사용 가능

2.2.3 매크로 사용

- ❑ 사용자 정의 변수에 특정한 문자열을 정의하고 표현
- ❑ 보다 일관되고, 이식성 높아지고, 융통성 증가
- ❑ 표현

CC = gcc

- ✓ 의미 – CC는 makefile 파일 내부에서 \$(CC) 또는 \${CC}로 표기
- ✓ 같은 소스를 타겟 아키텍처에 맞게 크로스 컴파일 하고자 할 때

CC = arm-linux-gcc

매크로 작성의 기본 규칙

- ❑ 매크로는 모두 대문자로 구성
 - ✓ TARGET (o), Target (x), TargetProg (x)

- ❑ '=' 를 포함하는 하나의 문장

```
NAME = string
```

- ✓ = 의 좌측에 매크로, 우측에 정의 문자열

- ❑ 여러 행을 사용하여 정의 하고자 할 때

```
CFLAGS = -D_KERNEL__ -DMODULE -W -Wall \
        -DNO_DEBUG
```

- ✓ 위의 문장은 아래와 동일

```
CFLAGS = -D_KERNEL__ -DMODULE -W -Wall -DNO_DEBUG
```

- ❑ 매크로 정의 수식에 매크로 포함 가능

```
CFLAGS = -I./include $(INCL)
```


매크로 작성의 기본 규칙

❑ 매크로 참조

```
CC = gcc
TARGET = starcraft

CFLAGS = -DLINUX -Wall

SRCS = main.c abc.c alpha.c

$(TARGET) : $(SRCS) $(TARGET).c
```

- ✓ '\$' 와 소괄호 (()) 나 중괄호 ({ }) 로 둘러 쌓여 사용
- ✓ 다른 문자열과 혼합되어 사용 가능
 - \$(TARGET).c

❑ 정의되지 않은 매크로에 대한 참조는 NULL과 같음

```
NAME =
CFLAGS = -DLINUX -DX11 -Wall -D$(NAME)
```

← 위는 다음과 같은 의미 →

```
NAME =
CFLAGS = -DLINUX -DX11 -Wall
```

매크로 작성의 기본 규칙

□ 중복된 정의는 최후 정의 값이 유효

```
GAME = tetris  
GAME = solitaire  
GAME = starcraft  
TARGET = ${GAME}      # TARGET 은?
```

- ✓ GAME = tetris 와 GAME = solitaire 는 무시
- ✓ GAME = starcraft 만 정의
- ✓ 그러므로 TARGET = \${GAME} 는 TARGET = starcraft 와 동일

□ 주석문 – ‘#’로 시작

```
CFLAGS = -I./include # 현재 디렉터리에 존재하는 include
```

매크로 작성의 기본 규칙

□ 대입 연산자

CFLAGS = -Wall \$(INCL)	# 재귀적 확장 매크로
INCL = -I/usr/share/include	
LIBDIR := -DLINUX \$(LIBS)	# 단순 확장 매크로
LIBS = -L/usr/bin	
CFLAGS += -I\$(LIBS)	# 정의수식 추가
CPPFLAGS ?= -DCLASS	# 조건적 매크로 정의

✓ '='

- 정의수식에 포함된 모든 매크로를 치환
- 위 표에서 **CFLAGS = -Wall \$(INCL)** 는 **CFLAGS = -Wall -I/usr/share/include** 와 동일
- 반복 스캔

✓ ':='

- 정의문에 있는 매크로가 문장 전에 정의되어 있으면 포함;
- 위 표에서 **LIBS** 매크로는 **LIBDIR**보다 뒤에 정의 되므로 **LIBDIR** 매크로에 불포함
- 단일 스캔

✓ '+='

- 정의 수식을 추가하여 매크로 확장
- **CFLAGS += -I\$(LIBS)** 은 **CFLAGS = -Wall -I/usr/share/include -L/usr/bin** 과 동일

✓ '?='

- 현 매크로가 전에 정의 되지 않은 경우만 유효
- **CPPFLAGS** 가 이 문장 앞에 정의 되어 있지 않기에 **CPPFLAGS ?= -DCLASS** 는 **CPPFLAGS = -DCLASS**와 같다.

매크로 사용 시 주의 사항

- 구분자로서 따옴표 불필요

MACRO = “string”

CFLAGS = \$(MACRO)

CFLAGS = string 이 아닌 CFLAGS = “string”로 인식

- 매크로 이름에 ‘:’, ‘=’, ‘#’, 탭 사용 금지

✓ Makefile 에서 탭 문자는 명령행을 의미

- 매크로 정의 후 매크로 호출

\$(NAME)

NULL 과 같다.

NAME = HONG

- ‘\’, ‘<’, ‘>’와 같은 쉘 메타 문자 사용 자제

✓ Makefile의 다른 용도와 혼란 야기

내부 정의 매크로

- 쉘 명령으로 내부 정의 매크로 확인 가능
 - ✓ `make -p | grep ^[:alpha:]*[:space:]*=[:space:]`

매크로	의미	매크로	의미
AR	아카이브 관리 프로그램	AS	어셈블러
CC	C 컴파일러	CXX	C++ 컴파일러
CPP	C 전처리기	RM	<code>rm -f</code> 의 의미
CFLAGS	C 컴파일러 플래그	CXXFLAGS	C++ 컴파일러 플래그
CPPFLAGS	C 전처리기 플래그	LDFLAGS	링크 플래그
LFLAGS	Lex 플래그	ASFLAGS	어셈블러 플래그
ARFLAGS	ar 플래그		

자동 매크로

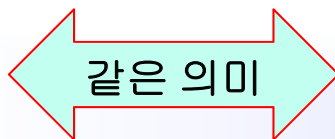
- 내부정의된 매크로지만 **make -p** 명령으로 확인 불가

매크로	의미	매크로	의미
\$?	현재의 타겟 보다 최근에 수정된 종속항 리스트	\$^	현재 타겟의 종속항 리스트
\$<	타겟보다 이후에 수정된 종속항 리스트 (확장자 규칙에서만 사용)	\$*	종속항 리스트에서 확장자를 제외한 파일이름들 (확장자 규칙에서만 사용)
\$@	현재의 타겟 이름	\$%	라이브러리 모듈의 .o 파일

```

TARGET = starcraft
SRCS = main.c vdo.c odo.c gui.c

$(TARGET) : $(SRCS)
gcc -o $(TARGET) $(SRCS)
    
```



```

TARGET = starcraft
SRCS = main.c vdo.c odo.c gui.c

$(TARGET) : $(SRCS)
gcc -o $@ $?
    
```

C 컴파일러 옵션

❑ man cc

- ✓ -D: define; 전처리기 등의 새로운 조건이나 변수를 정의
- ✓ -I: include; 헤더 파일이 위치한 디렉터리
- ✓ -L: 라이브러리 파일이 위치한 디렉터리
- ✓ -O: optimization. 예, -O2
- ✓ -W: warning. -Wall 은 모든 경고 무시하라는 의미
- ✓ -g: debugging 가능
- ✓ -l(소문자 L): 라이브러리 이름

CC = gcc

TARGET = starcraft

CFLAGS = -DLINUX -Wall

LIBS = -L/usr/lib -lm

SRCS = main.c abc.c alpha.c

\$(TARGET) : \$(SRCS) \$(TARGET).c

\$(CC) \$(CFLAGS) -o \$@ \$? \$(LIBS)

실습:

- ❑ \$HOME/class/ACS30021/ch2/lib/src/ 에 func1.c func2.c func3.c 와 Makefile 저장
- ❑ libmyfunc.so와 libmyfunc.a를 만들어 \$HOME/class/ACS30021/ch2/lib에 배치
- ❑ \$HOME/class/ACS30021/ch2/ 에 main.c 와 Makefile 저장
- ❑ \$HOME/class/ACS30021/ch2/include 에 헤더 파일 보관
- ❑ main_so 와 main_a 실행파일 완성

라이브러리 - \$HOME/ \$HOME/class/ACS30021/ch2/lib/src/

```
TARGET = libmylib.a libmylib.so

CC      = gcc

WDIR    = ../..
LDIR    = ${WDIR}/lib

SHARED  = -shared -fPIC

CFLAGS  = -Wall -O3 -D_REENTRANT
CFLAGS += -I./include -I${WDIR}/include

SRCS    = $(wildcard *.c)
OBJS    = $(SRCS:.c=.o)

#=====
all: ${TARGET} clean
#=====
libmylib.a : $(OBJS)
    ar -rscv $@ $?
    mv $@ ${LDIR}

libmylib.so : ${SRCS}
    $(CC) ${SHARED} $(CFLAGS) -o $@ $? $(LIBS)
    mv $@ ${LDIR}

#-----
clean:
    rm -rf core *.o a.out

#=====
```

\$HOME/class/ACS30021/ch2/

```
TARGET = main_lib_so main_lib_a
WDIR   = .

CC      = gcc

CFLAGS = -Wall -O3 -D_REENTRANT
CFLAGS += -I${WDIR}/include

LDIR = -L./lib
LIBS = $(LDIR) -lmylib

SRCS  = $(wildcard *.c)
OBJS  = $(SRCS:.c=.o)

#=====
all: ${TARGET} clean
#=====
main_lib_so : $(SRCS)
              $(CC) $(CFLAGS) -o $@ $? $(LIBS)

main_lib_a : $(SRCS)
              $(CC) -static $(CFLAGS) -o $@ $? $(LIBS)

#-----
clean:
        rm -f core *.o a.out

#=====
```

2.2.4 확장자 규칙(Suffix rules)

- 확장자가 가지는 규칙에 기초하여 사용자가 내리는 명령을 해석하여 컴파일을 자동화
 - 1. 내부 정의 확장자 규칙
 - 2. 확장자 규칙 재정의
 - 3. 와일드카드 매칭과 대입참조

2.2.4 확장자 규칙(Suffix rules)

□ 내부 정의 확장자 규칙

✓ make -p

```
%o: %f
    $(COMPILE.F) $(OUTPUT_OPTION) $<
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
%.o: %.cc
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
%.o: %.C
    $(COMPILE.C) $(OUTPUT_OPTION) $<
%.o: %.cpp
    $(COMPILE.cpp) $(OUTPUT_OPTION) $<
```

COMPILE.xx 각 소스들의 컴파일러로 지정

OUTPUT_OPTION은 -o \$@ 으로 지정

확장자 규칙 재정의

❑ 다음의 makefile의 결과는?

```
OBJS = memo.o. calendar.o main.o
```

```
all : diary
```

```
diary : $(OBJS)
```

```
$(CC) -o $@ $?
```

- ✓ diary를 생성하기 위해 make는 종속항을 검사하여 각 종속항을 타겟으로 설정
- ✓ diary는 memo.o에 의존하므로 memo.o를 생성하기 위한 규칙을 검사

```
OBJS = memo.o calendar.o main.o
```

```
.SUFFIXES : .o .c # 내부정의 확장자 규칙에서 .o와 .c를 발견하면 아래 명령 수행
```

```
%.o : %.c # %는 $(OBJS)에서 일치하는 확장자를 제외한 파일명 의미
```

```
# 즉, memo calendar, main
```

```
$(CC) -DDEBUG -c -o $@ $< # $?가 아닌 $< 사용됨
```

확장자 규칙 재정의

```
all : diary
```

```
diary : $(OBJS)
```

```
$(CC) -o $@ $?
```

자동 매크로

- ✓ 위에서 “%.o:%.c”는 “.c.o:”로 쓰여도 같다.

확장자 규칙 – 와일드카드 매칭, 대입참조 기법

- ❑ SRCS 매크로에서 `$(wildcard *.c)`는 현재 디렉터리에서 `*.c`와 파일명이 일치하는 파일을 찾아 공백문자로 구분하여 매크로 `SRCS`에 종속항 리스트로 대입
- ❑ `$(SRCS: .c=.o)`를 통해 확장자가 `.o`인 파일들 생성

```
SRCS = $(wildcard *.c)
OBJS = $(SRCS: .c=.o)

all : diary
diary : $(OBJS)
        $(CC) -o $@ $?
```

- ❑ 함수 사용에 의한 확장

```
OBJS = $(patsubst %.c, %.o, $(wildcard *.c))

all : diary
diary : $(OBJS)
        $(CC) -o $@ $?
```

2.2.5 특수 타겟

❑ 더미 타겟 (Dummy target)

```
OBJS = $(patsubst %.c, %.o, %(wildcard *.c))
```

```
all : diary
```

```
diary : $(OBJS)
```

```
$(CC) -o $@ $?
```

```
clean :
```

```
rm -rf *.o diary
```

- ✓ 일반적으로 타겟은 새로 생성될 파일인데 반해 더미 타겟은 타겟이 생성되지 않는 개념

2.2.6 명령어 사용 규칙

❑ 명령행에 적용

❑ 예1:

```
echo :  
    @echo "echo test!"
```

- ✓ make는 셸을 이용하여 새로운 프로세스를 생성하여 `</bin/sh -c echo "echo test!">` 를 실행 (@은 셸명령의 입력에 대한 에코 기능 중지 명령)

❑ 예2:

```
del:  
    cd ./backup          # 현 프로세스가 실행  
    rm -rf *             # 새 프로세스가 현재 디렉터리에서 실행
```

- ✓ 그러므로

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

- ✓ 그러나 ‘;’ 는 앞의 명령 실패와 무관하게 뒤 명령을 실행
- ✓ ‘&&’ 는 앞의 명령이 성공일때만 다음 명령 실행

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


명령어 사용 규칙

❑ .SILENT :

- ✓ .SILENT 타겟을 만들기 위해 등록된 종속항에 수행되는 명령어들에게 에코 기능 제거
- ✓ 화면에 명령어 실행 과정 미 출력

❑ -

- ✓ 오류에 무관하게 **makefile** 내 명령 실행
- ✓ 명령어 앞에 '-' 추가

cat :

-cat file.txt

❑ .IGNORE:

- ✓ 오류와 무관하게 **makefile** 전체 실행

명령어 사용 규칙

□ 명령행에 쉘 변수 참조

```
OBJS = $(patsubst %.c, %.o, $(wildcard *.c))
```

```
all : diary
```

```
diary : $(OBJS)
```

```
$(CC) -o $@ $?
```

```
cp -f diary $$$(HOME)
```

```
#사용자의 홈 디렉터리로 diary 복사
```

```
clean :
```

```
rm -rf *.o diary
```

✓ \$\$ (변수명) 의 형식

- 쉘 변수나 쉘 스크립트 사용 시
- \$\$ (쉘 변수)

2.2.7 프로젝트 관리

- ❑ 분산 디렉터리 파일들의 **make** 수행
 - ✓ 전체 경로 지정
 - ✓ **VPATH**
 - ✓ 재귀적 **make** 사용
- ❑ 조건적 실행
 - ✓ 종속관계 행에
 - **ifeq ~ else ~ endif**
 - **ifneq ~ else ~ endif**
- ❑ 함수의 사용
 - ✓ 문자열 처리 함수
 - ✓ 파일 관련 함수
 - ✓ 반복을 위한 함수

분산 디렉터리 파일들의 **make** 수행

- ❑ \$HOME/diary 디렉터리
- ❑ 각각 main, memo, calendar 서브 디렉터리 존재
- ❑ 각 서브 디렉터리에 Makefile 과 소스 존재
- ❑ diary 디렉터리 Makefile

```
root@os:/temp/book/make
[root@os make]# tree
.
|-- Makefile
|-- calendar
|   |-- Makefile
|   `-- calendar.c
|-- include
|   `-- diary.h
|-- main
|   |-- Makefile
|   `-- main.c
`-- memo
    |-- Makefile
    `-- memo.c

4 directories, 8 files
[root@os make]#
```

경로지정에 의한 **make**

❑ \$HOME/diary/*/Makefile

```
# $HOME/diary/main/Makefile

OBJS = $(patsubst %.c, %.o, $(wildcard *.c))

all : $(OBJS)

clean :
    rm -rf *.o
```

```
# $HOME/diary/memo/Makefile

OBJS = $(patsubst %.c, %.o, $(wildcard *.c))

all : $(OBJS)

clean :
    rm -rf *.o
```

```
# $HOME/diary/calendar/Makefile

OBJS = $(patsubst %.c, %.o, $(wildcard *.c))

all : $(OBJS)

clean :
    rm -rf *.o
```

경로지정에 의한 **make**

❑ \$HOME/diary/Makefile

```
# $HOME/diary/Makefile

OBJS = memo/memo.o calendar/calendar.o main/main.o
TARGET = diary

all : $(TARGET)
$(TARGET) : MAIN MEMO CALENDAR
            $(CC) $(CFLAGS) -o $@ $(OBJS)

MAIN :
        cd main && $(MAKE) all    # make -C main과 동일

MEMO :
        cd memo && $(MAKE) all

CALENDAR :
        cd calendar && $(MAKE) all

clean :
        rm -rf $(TARGET)
        cd main ; $(MAKE) clean
        cd memo ; $(MAKE) clean
        cd calendar ; $(MAKE) clean
```

VPATH 매크로

- ❑ **make**가 소스파일을 찾을 때 현재 디렉터리뿐 아니라 **VPATH** 매크로에 등록된 디렉터리까지 검색
- ❑ 각 서브디렉터리에 **Makefile**이 없어도 됨
- ❑ 단, **다른 디렉터리라 하더라도 파일명이 동일하면 안됨**

```
VPATH = memo calendar main
```

```
OBJS = memo.o calendar.o main.o
```

```
CFLAGS = -I./include
```

```
diary : $(OBJS)  
        $(CC) -o $@ $?
```

```
clean :  
        -rm -rf *.o diary
```

재귀적 **make** – 셸 스크립트

- ❑ **DIRS** 매크로에 서브디렉터리를 등록
- ❑ 셸 스크립트에 의한 일괄적 **make** 명령

- ❑ 최상위 **Makefile**에서 지정한 매크로를 서브 디렉터리의 **Makefile**에 전달
- ❑ 전달 하고자 하는 매크로 앞에 **export** 키워드 추가
- ❑ 모든 매크로 전달 – **export** 키워드 단독으로 추가

```
# $HOME/diary/*/Makefile
```

```
OBJS = $(patsubst %.c, %.o, $(wildcard *.c))  
CFLAGS = -I../include
```

```
all : $(OBJS)  
        cp -f $^ ../
```

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

```
# $HOME/diary/Makefile
```

```
DIRS = memo calendar main  
OBJS = memo.o calendar.o main.o  
TARGET = diary  
export CC = gcc
```

```
all : objs  
        $(CC) -o $(TARGET) $(OBJS)
```

```
objs :  
        @for dir in $(DIRS); do \  
        make -C $$dir || exit $? ; \  
        done
```

```
clean :  
        @for dir in $(DIRS) ; do \  
        make -C $$dir clean ; \  
        done  
        rm -rf $(TARGET)
```


실습:

```
// file1.c

#include <stdio.h>

void func1 ()
{
    fprintf (stdout, "Function 1\n");
}
```

```
// file2.c

#include <stdio.h>

void func2 ()
{
    fprintf (stdout, "Function 2\n");
}
```

```
// file3.c

#include <stdio.h>

void func3 ()
{
    fprintf (stdout, "Function 3\n");
}
```

```
// fun.h

#ifndef __FUN_H
#define __FUN_H

extern void func1();
extern void func2();
extern void func3();

#endif
```

```
// main.c

#include "fun.h"
#include <stdio.h>

int main (int argc, char **argv)
{
    fprintf (stdout, "main 함수:: \n");
    func1 ();
    func2 ();
    func3 ();

    return 0;
}
```

실습:

- 현재 작업 디렉터리에
 - 서브 디렉터리 upath, vpath, sh 생성
 - upath 디렉터리에 d1, d2, d3 디렉터리 생성
 - d1/file1.c, d2/file2.c d3/file3.c 복사
 - vpath, sh 디렉터리에도 동일하게 수행
- 1. 경로 지정에 의한 분산 make 방법 수행
- 2. vpath 방법에 의한 make 수행
- 3. shell script 방법에 의한 make 수행

분산 makefile

- 각각의 d1, d2, d3 디렉터리에서 libd1.a, libd2.a, libd3.a를 생성하여 ../..에 저장
- 현재 작업 디렉터리에 있는 main.c파일과 ./lib/에 있는 라이브러리 파일들을 가지고 실행파일 main 생성

```
TARGET = main
```

```
WDIR = .
```

```
DIRS = lib/src/d1 lib/src/d2 lib/src/d3
```

```
export CC = gcc
```

```
CFLAGS = -Wall -O3 -D_REENTRANT
```

```
CFLAGS += -I${WDIR}/include
```

```
LFLAGS = -L./lib -L./
```

```
LIBS = -ld1 -ld2 -ld3
```

```
SRCS = $(wildcard *.c)
```

```
OBJS = $(SRCS:.c=.o)
```

```
#=====
```

```
all: archives ${TARGET} clean
```

```
#=====
```

```
${TARGET}: $(SRCS)
```

```
$(CC) $(CFLAGS) -o $@ $? $(LFLAGS) $(LIBS)
```

```
#-----
```

```
archives:
```

```
@for dir in $(DIRS); do \
```

```
$(MAKE) -C $$dir || exit $? ; \
```

```
done
```

```
#-----
```

```
clean:
```

```
@for dir in $(DIRS); do \
```

```
$(MAKE) -C $$dir clean ; \
```

```
done
```

```
rm -f core *.o a.out
```

```
#=====
```

분산 makefile

□ 라이브러리 생성용 makefile

```
TARGET = libd1
TARGET1 = $(TARGET).a
TARGET2 = $(TARGET).so

CC      = gcc

WDIR   = ../../..
LDIR   = ${WDIR}/lib

SHARED = -shared -fPIC

CFLAGS = -Wall -o32 -O2 -D_REENTRANT
CFLAGS += -I./include -I${WDIR}/include

LFLAGS = -lm
LFLAGS +=

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
#=====
all: ${TARGET1} clean
#=====
${TARGET1} : ${OBJS}
    ar -rscv $@ $?
    mv $@ ${LDIR}

${TARGET2} : ${SRCS}
    $(CC) ${SHARED} $(CFLAGS) -o $@ $? $(LFLAGS)
    mv $@ ${LDIR}
#-----
clean:
    rm -f core *.o a.out
#=====
```

다수의 **makefile** 존재

- 한 디렉터리에 여러 **makefile**을 저장하여야 할 경우
 - ✓ **File_name.mk** 로 각각의 **makefile** 이름 설정
 - ✓ **\$ make -f File_name.mk** 로 **makefile** 실행

조건적 실행

- ❑ C 언어의 if {~ }else {} 에 해당
- ❑ ifeq ~ else ~ endif
- ❑ ifneq ~ else ~ endif
- ❑ ifdef ~ else ~ endif
- ❑ ifndef ~ else ~ endif

```
# ifeq

all :
ifeq $(CC), gcc
    @echo "C 컴파일러는 GNU gcc "
else
    @echo "C 컴파일러는 $(CC)"
endif
```

```
# ifneq

all :
ifneq $(CC), gcc
    @echo "C 컴파일러는 $(CC)"
else
    @echo "C 컴파일러는 GNU gcc"
endif
```

함수

□ \$(함수명 인자들)

- ✓ 공백이나 탭 문자로 함수명과 함수인자 구분
- ✓ 여러 개의 함수 인자 존재 – 콤마, (,), 로 구분

□ 셀 명령 함수

□ 문자열 처리 함수

□ 파일이름 관련 함수

□ 반복을 위한 함수

함수 - 셸 명령 함수

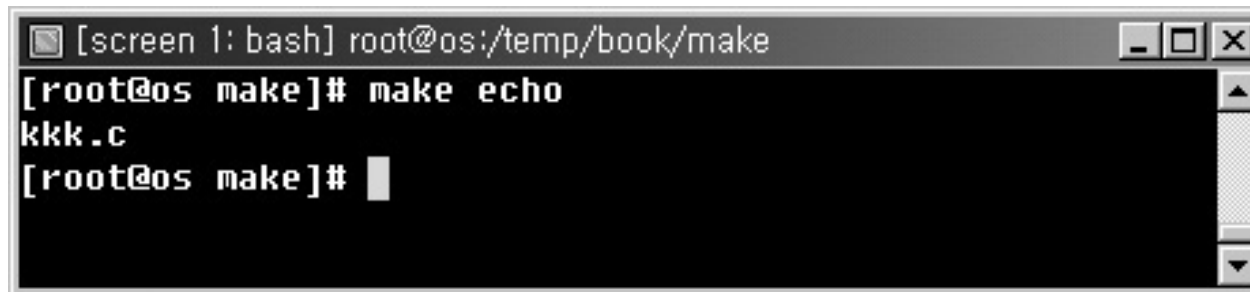
❑ \$(shell 셸명령어)

```
SRCS = $(shell ls *.c)
```

```
echo :
```

```
    @echo $(SRCS)
```

❑ 실행 결과



```
[screen 1: bash] root@os:/temp/book/make
[root@os make]# make echo
kkk.c
[root@os make]#
```


함수 – 문자열 처리 함수

- ❑ **\$(subst 검색문자, 변경문자, 대상 문자열)**
 - ✓ 대상 문자열에서 검색문자를 발견하면 변경문자로 치환
- ❑ **\$(patsubst 패턴, 변경 문자열, 대상 문자열)**
 - ✓ 대상 문자열에서 패턴을 발견하면 변경문자열로 치환
 - ✓ 패턴 기호로 %기호는 공백과 탭을 제외한 모든 문자 의미

```
STR = $(patsubst %.c, %.o, memo.c main.c ABCD)
```

```
echo:
```

```
    @echo $(STR)
```

- ❑ **\$(매크로명: 패턴=변경문자열)**

```
MACRO = memo.c main.c ABCD
```

```
STR = $(MACRO:%.c=%.o)
```

```
echo :
```

```
    @echo $(STR)
```

함수 – 반복처리 함수

- 현재 디렉터리에서 패턴과 일치하는 파일 리스트 추출
 - ✓ `$(wildcard *.c)` – 현재 디렉터리에 있는 모든 .C 파일의 리스트를 반환
 - ✓ `$(wildcard test.*)` – 파일명이 **test**인 모든 파일 반환

레포트

❑ autoconf

❑ automake