HALLYM UNIVERSITY

Operating Systems LAB 4

Wonpyo Kim

skykwp@gmail.com

Outline

- Substance
  - Makefile II
  - GDB

Macro

- 매크로는 특정한 코드를 간단하게 표현한 것이다. Makefile 에서 사용되는 매크로는 그 사용법이 비교적 간단하기 때문에 금방 익힐 수 있다.

- 매크로의 정의는 프로그램을 작성할 때 변수를 지정하는 것 처럼 한다.

```
OBJS = main.o tables.o rows.o ops.o

test : $(OBJS)
    gcc -o test $(OBJS)
```

- 위와 같이 매크로는 복잡한 코드를 간단하게 표현한 것이다.

Pre-defined Macro

- 예전부터 개발을 주도하던 사람들은 make 유틸리티를 만들면서 매크로를 몇 가지 미리 정해두었다. 여기서, "**make -p**" 라고 입력하면 미리 설정되어있는 모든 값(매크로, 환경변수 등)이 스크롤 된다.

- 많이 쓰이는 미리 정해진 매크로 들의 예

```
ASFLGS =
AS = as
CFLAGS =
CC = cc
CPPFLAGS =
CXX = g++
LDFLAGS =
LD = ld
MAKE_COMMAND = make
```

- 열거된 매크로는 극히 일부분이다. 하지만, 프로그램을 작성할 때 가장 많이 사용되는 매크로 들이다. 이들 매크로는 사용자에 의해 재정의가 가능하다.

Pre-defined Macro

- 예를 들어 gcc 의 옵션 중에 디버그 정보를 표시하는 "-g" 옵션을 넣고 싶다면, 아래와 같이 재정의 한다.

```
CFLAGS = -g
```

- 다음은 Pre-defined Macro를 사용하여 이전 과제의 Makefile을 만든 예이다.

```
CC = gcc
CFLAGS = -g -c
OBJECTS = main.o tables.o rows.o ops.o
SRCS = main.c tables.c rows.c ops.c
TARGET = lab03_kwp


$(TARGET) : $(OBJECTS)
    $(CC) -o $(TARGET) $(OBJECTS)

clean :
    rm -rf $(OBJECTS) $(TARGET)

main.o : main.h main.c
tables.o : tables.h tables.c
rows.o : rows.h rows.c
ops.o : ops.c
```

- 여기서, 한 가지 다른점은 .c 파일을 .o 파일로 바꾸는 부분의 명령이 없다. 하지만, 정상적으로 컴파일된다.

- 이유는 Pre-defined Macro들은 make 의 내부에서 이용되기 때문이다. 따라서, CFLAGS 의 설정만 추가해주면 make 가 알아서 컴파일을 한다.

Suffix Rule

- 확장자 규칙은 파일의 확장자를 보고 적절한 연산을 수행시키는 규칙이다. 예를들어 .c 파일은 C 소스코드, .o 파일은 목적파일(Object file)을 말한다.

```
.SUFFIXES : .c .o
```

- **.SUFFIXES** 라는 매크로는 make 에게 해당 확장자를 주의깊게 처리하라고 명시한다고 생각하면 된다.

```
.SUFFIXES : .c .o

OBJECTS = main.o tables.o rows.o ops.o
SRCS = main.c tables.c rows.c ops.c

CC = gcc
CFLAGS = -g -c

TARGET = lab03_kwp

$(TARGET) : $(OBJECTS)
    $(CC) -o $(TARGET) $(OBJECTS)

clean :
    rm -rf $(OBJECTS) $(TARGET)

main.o : main.h main.c
tables.o : tables.h tables.c
rows.o : rows.h rows.c
ops.o : ops.c
```

- .SUFFIXES : .c .o 라고 명시했기 때문에 make 내부에서는 .c 를 컴파일해서 .o 파일을 만들어내는 루틴이 자동적으로 동작하게 된다.

- 다음은 make 에서 기본적으로 제공하는 확장자 리스트이다.
- .out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch, .web, .sh, .elc, .el

Suffix Rule

- 이전 페이지의 예제를 조금 바꾸면 아래와 같다. include 경로를 추가하고 .c.o 부분을 임의로 확장자 규칙을 만들어 주었다.

```
.SUFFIXES : .c .o

OBJECTS = main.o tables.o rows.o ops.o
SRCS = main.c tables.c rows.c ops.c
CC = gcc
CFLAGS = -g -c
INC = -I/home/kwp/.toy/lab4

TARGET = lab03_kwp

$(TARGET) : $(OBJECTS)
    $(CC) -o $(TARGET) $(OBJECTS)

.c.o :
    $(CC) $(INC) $(CFLAGS) $(SRCS)

clean :
    rm -rf $(OBJECTS) $(TARGET)

main.o : main.h main.c
tables.o : tables.h tables.c
rows.o : rows.h rows.c
ops.o : ops.c
```

```
kwp@A1409-OSLAB-01:~/.toy/lab4$ make
gcc -I/home/kwp/.toy/lab4 -g -c main.c tables.c rows.c ops.c
gcc -o lab03_kwp main.o tables.o rows.o ops.o
```

- INC 라는 매크로를 추가하여 컴파일할 때 이용하도록 하였다.

Macro Substitution

- 매크로의 내용을 조금 바꾸어야 할 때, 치환 기능을 사용하면 편리하다.
  - $(MACRO_NAME:OLD=NEW) 와 같은 형태로 작성한다.

```
OBJS = main.o tables.o rows.o ops.o
SRCS = $(OBJS:.o=.c)
```

- 이렇게 바꿔주면 좀 더 간결한 코드가 작성된다. 즉, OBJS 내에 있는 .o 를 .c 로 바꾼다 라는 의미가 된다.

Automatic Dependency

- Makefile을 만들 때 target, dependency, command 가 연속적으로 정의해야한다. 그런데, 파일이 10개가 넘어갈 경우 힘든 상황이 발생한다.

- 이렇게 반복적인 작업을 대신해주는 gccmakedep 이라는 유틸리티가 존재한다. 이 유틸리티는 어떤 파일의 의존 관계를 자동으로 조사하여 Makefile 뒷 부분에 자동으로 붙여준다.

- 대부분 gccmakedep 은 기본적으로 설치되지 않는다. 간단하게 보려면 gcc -M XX.c 를 입력하면 된다.

```
.SUFFIXES : .c .o
CFLAGS = -g

OBJS = main.o tables.o rows.o ops.o
SRCS = $(OBJS:.o=.c)

test : $(OBJS)
    $(CC) -o test $(OBJS)

dep :
    gccmakedep $(SRCS)
```

- 왼쪽은 현재 Makefile의 전체내용이다.

- 작성 후 make dep 을 수행한다.

Automatic Dependency

- 이와 같이 자동으로 의존관계를 작성해준다. 이후 make 를 실행해주면 정상적으로 컴파일된다.

```
.SUFFIXES : .c .o
CFLAGS = -g

OBJS = main.o tables.o rows.o ops.o
SRCS = $(OBJS:.o=.c)

test : $(OBJS)
    $(CC) -o test $(OBJS)

dep :
    gccmakedep $(SRCS)


# DO NOT DELETE
main.o: main.c /usr/include/stdc-predef.h /usr/include/stdio.h \
 /usr/include/x86_64-linux-gnu/bits/libc-header-start.h \
 /usr/include/features.h /usr/include/x86_64-linux-gnu/sys/cdefs.h \
 /usr/include/x86_64-linux-gnu/bits/wordsize.h \
 /usr/include/x86_64-linux-gnu/bits/long-double.h \
 /usr/include/x86_64-linux-gnu/gnu/stubs.h \
 /usr/include/x86_64-linux-gnu/gnu/stubs-64.h \
 /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h \
 /usr/include/x86_64-linux-gnu/bits/types.h \
 /usr/include/x86_64-linux-gnu/bits/typesizes.h \
 /usr/include/x86_64-linux-gnu/bits/types/__FILE.h \
 /usr/include/x86_64-linux-gnu/bits/types/FILE.h \
 /usr/include/x86_64-linux-gnu/bits/libio.h \
 /usr/include/x86_64-linux-gnu/bits/_G_config.h \
 /usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h \
 /usr/lib/gcc/x86_64-linux-gnu/7/include/stdarg.h \
 /usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
 /usr/include/x86_64-linux-gnu/bits/sys_errlist.h /usr/include/stdlib.h \
 /usr/include/x86_64-linux-gnu/bits/waitflags.h \
```

Multiple Target

- 우리는 여지껏 단 한 개의 실행파일 만을 위해 작성해왔다. 하지만, 여러 개의 파일을 한 번에 생성할 수 있다.

```
.SUFFIXES : .c .o
CC = gcc
CFLAGS = -g

OBJS = main.o tables.o rows.o ops.o
SRCS = $(OBJS:.o=.c)

all : test1 test2 test3

test1 : $(OBJS)
    $(CC) -o test1 $(OBJS)
test2 : $(OBJS)
    $(CC) -o test2 $(OBJS)
test3 : $(OBJS)
    $(CC) -o test3 $(OBJS)

dep :
    gccmakedep $(SRCS)
```

- all 이라는 키워드를 정의하고 실행파일이 될 이름들을 적어주었다. 그 밑은 명령들이며 이전과 내용은 같다.
- make all 을 통해 수행을 할 수 있다. 결과는 test1, test2, test3 이 생성된다.

Recursive Make

- 규모가 큰 프로그램들은 파일들이 하나의 디렉토리에 있지 않은 경우가 많다. 여러 개의 서브시스템이 전체 시스템으로 통합된다고 가정하면, 각 서브 시스템에 Makefile이 존재한다.

- 따라서, 여러 개의 Makefile을 동작시킬 수 있다.

```
.SUFFIXES : .c .o
CC = gcc
CFLAGS = -g

all : LAB3_task LAB4_task

LAB3_task :
    cd lab3 ; $(MAKE)

LAB4_task :
    cd lab4 ; $(MAKE)
```

- 위 예제는 이전에 작성했던 과제들의 Makefile 을 상위 폴더에서 실행하도록 하는 것이다.

Recursive Make

```
kwp@A1409-OSLAB-01:~/.toy$ make
cd lab3 ; make
make[1]: Entering directory '/home/kwp/.toy/lab3'
gcc -c main.c
gcc -c exchange.c
gcc -c get_data.c
gcc -c smallest.c
gcc -c select_sort.c
gcc -c print_data.c
gcc -o lab02_kwp main.o exchange.o get_data.o smallest.o select_sort.o print_data.o
make[1]: Leaving directory '/home/kwp/.toy/lab3'
cd lab4 ; make
make[1]: Entering directory '/home/kwp/.toy/lab4'
gcc -g   -c -o main.o main.c
gcc -g   -c -o tables.o tables.c
gcc -g   -c -o rows.o rows.c
gcc -g   -c -o ops.o ops.c
gcc -o test1 main.o tables.o rows.o ops.o
gcc -o test2 main.o tables.o rows.o ops.o
gcc -o test3 main.o tables.o rows.o ops.o
make[1]: Leaving directory '/home/kwp/.toy/lab4'
```

- 위와 같이 디렉토리를 순환하면서 지정된 각 폴더에 대해 make를 각각 수행하는 것을 볼 수 있다.

Outline

- Substance
  - Makefile II
  - **GDB**

GDB

- 버그(bug)라는 용어는 미국의 한 대형 컴퓨터에 벌레가 붙어 시스템을 정지시킨 이후로 사용되었다.

- 프로그램에 있어서는 컴파일하거나, 링크하는 작업을 하는 동안 출력하는 문법 오류, 논리 오류, 런타임 오류와는 다르게 컴파일러, 링커가 찾아내지 못하는 오류를 버그라고 한다.

- 이 버그를 제거하는 작업을 디버깅(debugging)이라 한다.

- GDB는 대표적인 디버깅 툴로 오류가 발생한 위치를 찾아주어 오류를 쉽게 잡을 수 있다.

GDB

- 실행에 앞서 다음과 같은 버그 프로그램을 하나 작성한다. bug1.c

```
1
2 #include <stdio.h>
3
4 int sum(int i);
5
6 void main(void)
7 {
8     printf("%d\n", sum(10));
9 }
10
11 int sum(int i)
12 {
13     return i+sum(i-1);
14 }
15
```

GDB

- gdb를 사용하기 위해서는 디버깅에 필요한 여러 정보를 만들어 주는 "-g" 옵션을 사용하여 컴파일 해야한다.

```
kwp@A1409-OSLAB-01:~/student/lab4/gdb$ gcc -g bug1.c -o bug1
kwp@A1409-OSLAB-01:~/student/lab4/gdb$ ls
bug1  bug1.c
```

- GDB의 실행은 다음과 같다.

  $ gdb 실행파일이름

GDB

- 실행하면 다음과 같은 화면을 볼 수 있다.

```
[kwp@A1409-OSLAB-01:~/student/lab4/gdb$ gdb bug1
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bug1...done.
(gdb)
```

- 종료는 quit 를 입력하면 된다.

GDB

- 먼저 GDB 에서 사용되는 명령은 한 글자를 알고있을 때 <TAB> 키로 찾을 수 있다. 기본적으로 help 를 입력하면 명령어를 참조할 수 있다.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

GDB

- 먼저 작성한 bug1.c 의 내용을 살펴볼 수 있다. list 명령을 입력한다. 범위는 [시작줄], [끝줄]로 줄 수 있다.

```
(gdb) list 1, 20
1
2       #include <stdio.h>
3
4       int sum(int i);
5
6       void main(void)
7       {
8               printf("%d\n", sum(10));
9       }
10
11      int sum(int i)
12      {
13              return i+sum(i-1);
14      }
15
(gdb)
```

GDB

- 실행은 run 명령을 통해 가능하다.

```
(gdb) run
Starting program: /home/kwp/student/lab4/gdb/bug1

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554676 in sum (i=<error reading variable: Cannot access memory at address 0x7fffff7feffc>) at bug1.c:12
12      {
(gdb)
```

- 오류의 내용은 세그멘테이션 오류가 발생하여 SIGSEGV 시그널을 받아 프로그램이 중단된 것을 알 수 있다. 이 오류는 메모리에 잘못접근했을 때 발생한다.

- 또한, bug1.c 의 12번 라인에서 발생한 것을 볼 수 있다.

GDB

- GDB에서는 명령라인 인수를 넣을 수도 있다. 다음의 bug2.c 파일을 작성한다.

```
1
2 #include <stdio.h>
3
4 void main(int argc, char **argv)
5 {
6     int i;
7
8     for (i = 0; i < argc; i++)
9         printf("%s\n", argv[i]);
10 }
11
```

- bug2.c 는 특별한 버그는 없다. 여기서, argc 는 입력인자의 갯수, argv 는 입력인자 값을 갖는다.

- -g 옵션을 이용해 컴파일한다.

GDB

- 다음과 같이 run 으로 실행하되, 입력 인자로 "hello operating systems"를 주었다.

```
(gdb) run hello operating systems
Starting program: /home/kwp/student/lab4/gdb/bug2 hello operating systems
/home/kwp/student/lab4/gdb/bug2
hello
operating
systems
[Inferior 1 (process 4169) exited with code 04]
(gdb)
```

- bug2.c 의 실행에서 중단점 설정과 그 지점의 값을 보려면 break 명령과 whatis, print 명령을 사용한다. 소스코드를 수정하는 것이 아니다.

GDB

- 소스 상에서 printf가 있는 라인 9번을 중단점으로 선언하였다. 이후 프로그램을 다시 실행하면 아래와 같이 멈추는 것을 볼 수 있다.

```
(gdb) break 9
Breakpoint 1 at 0x652: file bug2.c, line 9.
(gdb) run hello operating systeems
Starting program: /home/kwp/student/lab4/gdb/bug2 hello operating systeems
Breakpoint 1, main (argc=4, argv=0x7fffffffe3a8) at bug2.c:9
warning: Source file is more recent than executable.
9                       printf("%s\n", argv[i]);
(gdb) whatis i
type = int
(gdb) print i
$1 = 0
(gdb) print argc
$2 = 4
(gdb) print argv
$3 = (char **) 0x7fffffffe3a8
```

- whatis 는 변수의 타입을 print 는 변수의 값을 볼 수 있다.

GDB

- 현재 프로그램은 중단점으로 인해 중단되어 있으므로, 계속하려면 continue 명령을 입력하면 된다.

```
(gdb) print i
$2 = 0
(gdb) continue
Continuing.
/home/kwp/student/lab4/gdb/bug2

Breakpoint 1, main (argc=4, argv=0x7fffffffe3a8) at bug2.c:9
9                       printf("%s\n", argv[i]);
(gdb) print i
$3 = 1
```

GDB

- 현재 설정된 중단점의 리스트를 알고싶다면, info break 명령을 입력한다.

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555554652 in main at bug2.c:9
        breakpoint already hit 2 times
(gdb) delete 1
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

- 중단점의 삭제는 delete [Num] 으로 할 수 있다. 위에서의 라인 9를 말하는 것이 아니라 info break 상의 Num을 말하는 것이다.