Make 기반 빌드 시스템

계산기 프로그램 예제를 통한 Makefile 개념 이해

파일의 목록은 아래와 같다.

```
$ ls
add.c calc.c malt.c numbers.h sub.c
```

원래 우리가 사용하던 방법으로 사용을 한다면 아래와 같다.

```
$ gcc -g -c add.c
$ gcc -g -c calc.c
$ gcc -g -c mult.c
$ gcc -g -c sub.c
```

각 gcc 명령의 -c 옵션은 오브젝트 파일을 만드는 옵션이며, -g 옵션은 디버깅을 활성화하는 옵션이다.
위의 명령어를 실행을 한다면 디렉토리에 다음과 같이 추가된다.

```
$ ls
add.c calc.c malt.c numbers.h sub.c
add.o calc.o malt.o sub.o
```

디렉토리의 내용을 주의 깊게 살펴보면 각 소스코드 파일에 대응하는 오브젝트 파일이 하나씩 생성된 점을 확인 할 수 있다. 단, number.h 헤더 파일에 대응하는 오브젝트 파일은 생성되지 않았지만, 나머지 소스코드 파일에서 이미 포함하고 있음을 명심하자. 빌드 시스템의 용어 측면에서 설명하자면 이것은 “각 소스코드 파일이 numbers.h 헤더 파일에 의존한다.” 라고 한다.

계산기 프로그램을 최종적으로 빌드하기 위해 다음과 같이 모든 오브젝트 파일을 하나의 실행 파일로 링크 시킨다.

```
$ gcc -g -o calculator add.o calc.o mult.o sub.o
$ ls
add.c calc.c malt.c numbers.h sub.c
add.o calc.o malt.o sub.o calculator
```

이것으로 계산기 프로그램의 빌드에 관한 모든 절차가 완료됐다.

여러 가지 이유로 빌드 시스템에서 종속성 그래프는 매우 중요하다. 그래프를 바탕으로 빌드 시스템에 관련된 파일 들 뿐만 아니라 그 파일들 간의 종속성 관계도 한눈에 파악할 수 있기 때문이다. GNU Make와 같은 빌드 도구는 컴파일을 수행할 때 종속성 그래프를 이용해 최적화된 컴파일을 수행한다. 예를 들어 add.o에서 발생한 화살표는 add.c와 numbers.h 를 향하고 있는데, 이는 add.c와 numbers.h는 add.o를 컴파일하는 데 필요한 소스 파일임을 뜻한다. 게다가 이 두 소스 파일 중 하나라도 수정된다면 add.o는 다시 컴파일돼야 한다. 반대로 add.o을 컴파한 후에 이들 파일에 변경한 사항이 없다면 다시 컴파일할 필요가 없다.

이 개념을 바탕으로 에제 프로그램의 종속성 그래프를 어떻게 GNU Make환경에 맞게 작성하는지를 알아보자

간단한 Makefile 작성

```
 1 calculator: add.o calc.o mult.o sub.o
 2     gcc -g -o calculator add.o calc.o mult.o sub.o
 3
 4 add.o: add.c numbers.h
 5     gcc -g -c add.c
 6
 7 calc.o: calc.c numbers.h
 8     gcc -g -c calc.c
 9
10 mult.o: mult.c numbers.h
11     gcc -g -c mult.c
12
13 sub.o: sub.c numbers.h
14     gcc -g -c sub.c
```

컴파일 명령을 사용할 수 있게 컴파일할 파일을 나열하고 그들 간의 종속성을 나타내는 평문를 사용해야만 한다. 그리고 GNU Make 도구는 직관적인 번역 기능을 제공한다. 위의 코드는 소프 파일이나 오브젝트 파일과 동일한 디렉터리에 있다는 것을 가정했다.

위의 Makefile의 각 섹션은 새로운 규칙이 기술되어 있다. 1번 행에는 calculator가 의존하는 모든 파일의 목록이 기술돼있다. 2번 행에는 오브젝트 파일들로부터 calculator를 생성하기 위한 유닉스 명령이 기술돼 있다.

4번 행에는 add.o가 add.c와 numbers.h에 의존한다는 점이 기술돼 있으며, 마찬가지로 5번 행에는 add.o를 생성하기 위한 유닉스 명령이 기술돼 있다. 나머지 행들도 이와 비슷하게 작성되어 있다.

> 💡 Makefile에서 사용되는 모든 유닉스 명령은 스페이스 바로 공백을 만들면 에러가 발생하므로 모든 것을 tab으로 공백을 만들어야 한다.

모든 소스 파일과 makefile이 동일한 디렉토리에 있다고 가정하고, 유닉스 쉘의 gmake 명령을 실행해 모든 소스 파일을 컴파일한다.

GNU Make는 먼저 Makefile을 조사하고 나서 종속성 그래프를 생성한다. 그러고 나서 이 종속성 그래프를 바탕으로 실행할 명령을 결정한다. 그리고 GNU Make는 calculator 실행 파일을 만들기 전에 모든 오브젝트 파일이 존재하는지 검사한 후 실행할 명령의 순서를 결정한다.

빌드 도구에서 다음으로 중요한 개념은 증분 빌드(incremental build)이다. GNU Make는 맹목적으로 빌드하는 대신 오브젝트 파일이 존재하는 지 아니면 컴파일해야 하는지를 판단하기 위해 사전 분석을 한다. 따라서 처음 빌드를 수행한 다음 GNU Make를 호출하면 이전보다 빌드 작업량이 현저히 줄어든다.

```
$ gmake
gmake: 'calculator' is up to date.
```

위 경우 GNU Make는 파일의 타임 스탬프를 기준으로 자신이 생성한 모든 파일이 소스 파일보다 더 최근에 생성됐는지 판단한다. 생성된 모든 파일이 소스 파일보다 최신 상태이면 아무런 작업도 하지 않는다.

특정 소스 파일이 수정됐다면 해당 파일의 타임스탬프가 자동으로 갱신된다. 그 결과 GNU Make는 오브젝트 파일이 더는 최신이 아니며, 갱신할 필요가 있다고 판단하고, add.o와 calculator를 다시 컴파일 한다.

```
$ gmake
gcc -g -c add.c
```

```
gcc -g -o calculator add.o calc.o mult.o sub.o
```

만약 numbers.h 헤더 파일이 수정이 되었다면 모든 소스코드 파일이 numbers.h에 의존성이 존재하기 때문에 모든 오브젝트 파일과 calculator 실행 파일이 컴파일 된다.

마지막으로, Makefile에서 각 오브젝트 파일에 대한 규칙이 개별적으로 기술돼 있지 않다면 증분 빌드는 실행되지 않는다. 다음 코드와 같이 규칙의 오른쪽에 모든 소스 파일을 열거하면 GNU Make는 소스 파일 중 하나가 수정되더라도 모든 소스파일을 처음부터 다시 컴파일한다. 하지만 직관적으로 효율적이지 않다고 생각될 수 있다.

```
1 calculator: add.c calc.c mult.c sub.c numbers.h
2     gcc -g -o calculator add.c calc.c mult.c sub.c numbers.h
```

2번 행은 이전보다 간결하게 하나의 실행 명령만을 포함하고 있을 뿐이지만, 컴파일러가 호출될 때마다 모든 소스 파일을 컴파일한다. 이전 예제와 비교를 한다면, 위 경우 한 번의 명령으로 모든 소스 파일을 컴파일하고 그들을 링크시켜 실행 파일을 생성한다.

이제, 어떻게 하면 Makefile을 최적화할 수 있는지를 생각해보자.

결국 각 소스 파일과 오브젝트 파일의 대응을 개별적인 규칙으로 기술하는 것이 비효율적일 수 밖에 없다.

Makefile의 간결화

빌드 시스템도 기존 소스와 마찬가지로 중복적인 구문이 안 좋다는 것을 알고 있다. 그래서 Makefile을 좀 더 수월히 만들기 위해 GNU Make는 일반적인 작업에서 기본 내장 규칙을 제공한다.

> 💡 Makefile에서는 별도의 지정 없이도 a.c파일을 a.o파일로 컴파일하는 기능이다.

위의 예제를 더 간결하게 다시 작성 할 수 있다.

```
1 calculator: add.o calc.o mult.o sub.o
2     gcc -g  -o  calculator  add.o   calc.o  mult.o  sub.o
3
4 add.o calc.o mult.o sub.o: numbers.h
```

1번 행과 2번 행은 이전 Makefile의 내용과 동일하지만 나머지 대부분은 제거된다. GNU Make는 오브젝트 파일이 동명의 소스 파일에 의존하는 점을 이미 알고 있다. 단지 모든 오

브젝트 파일이 numbers.h 헤더 파일에 의존하는 점을 명시적으로 기술하면 된다.

코드를 더욱 읽기 쉽게 하려고 심볼 이름(Symbol name)을 사용하면 더욱 최적화 된다. GNU Make는 프로그래밍 언어에서와 같이 Makefile에 변수도 사용할 수 있게 제공된다.

아래는 변수가 사용된 예제를 나타낸다.

```
 1 SRCS=add.c calc.c mult.c sub.c
 2 OBJS=$(SRC:.c=.o)
 3 PROG=calculator
 4 CC=gcc
 5 CFLAGS=-g
 6
 7 $(PROG):$(OBJS)
 8           $(CC) $(CFLAGS) -o $@ $^
 9
10 $(OBJS): numbers.h
```

1번 행에는 SRCS 변수에 모든 소스코드 파일이 정의됐다. 2번 행에는 OBJS변수에 편리한 GNU Make 구문으로 SRCS 변수에 있는 모든 소스 파일명에서 .c를 .o로 대체한 것이 정의 됐다. 따라서 OBJS 변수는 모든 오브젝트 파일의 목록이 된다.

3번 행에는 PROG 변수에 실행 파일명이 정의됐다. 4번 행에는 CC 변수에 컴파일러 도구 이름이 정의됐다. 이 값들이 Makefile에서 여러 번 참조될 경우 특정한 곳에 정의해두면 아주 효율적이다.

5번 행에는 CFLAGS 변수에 디버깅 정보 포함 옵션이 부여됐다. 암묵적으로 소스 코드 파일로 부터 오브젝트 파일을 생성하는 이전 예제에서는 CFLAG 변수를 사용하지 않았던 점을 유념하자

7~8번 행은 변수로 대체됐을 뿐이다. $@과 $^는 Makefile에서 사용되는 매크로이다.

$@ : 규칙의 왼쪽에 기재된 $(PROG), 즉, calculator를 가리키며, 타겟이라고 불린다.

$^ : 규칙의 오른쪽에 기재된 $(OBJS)를 가리키며, 모든 오브젝트를 의미한다.

빌드 타겟 추가

빌드 시스템은 프로그램을 컴파일하는 외에도 많은 일을 할 수 있다. 사실 빌드 시스템은 입력 파일로부터 출력 파일이 생성되는 과정상에서 어떤 일도 처리할 수 있다. 심지어 파일을 제거하거나 한 위치에서 그 외의 위치로 복사하는 기능도 포함한다.

C언어 빌드 시스템에서 가장 보편적인 작업으로는 빌드 트리를 클리닝, 타겟 머신에 실행 파일을 설치하는 일을 들 수 있다.

clean 타겟은 소프트웨어를 컴파일할 때 생성된 모든 파일을 제거하는 일을 주된 목적으로 하며, install 타겟은 타겟 머신의 표준 바이너리 경로로 실행 파일을 복사하는 일을 주된 목

적으로 한다.

```
 1 SRCS = add.c calc.c mult.c sub.c
 2 OBJS = $(SRC:.c=.o)
 3 PROG = calculator
 4 CC = gcc
 5 CFLAGS = -g
 6 INSTALL_ROOT=/usr/local
 7
 8 $(PROG):$(OBJS)
 9           $(CC) $(CFLAGS) -o $@ $^
10
11 $(OBJS): numbers.h
12
13 clean:
14     rm -f $(OBJS) $(PROG)
15
16 install: $(PROG)
17     cp $(PROG) $(INSTALL_ROOT)/bin
```

위의 예제에서 13, 16번 행이 추가가 된 것이다. 이전 예제와 달리 규칙 오른쪽에 입력 파일 목록이 없지만, 이는 표준 GNU Make규칙 중 하나이다. 입력 파일의 타임스탬프를 확인하지 않고 이 타겟은 항상 실행된다.

위의 타겟중 clean이 실행이 되면 .o파일과 calculator파일이 사라진다. 즉, 아래와 같은 기능을 한다.

```
rm -f add.o calc.o mult.o sub.o calculator
```

이 규칙에서 GNU Make는 rm명령의 실행을 회피할 방법이 없다. 파일의 타임 스탬프를 확인함으로써 중복 실행을 회피해 왔지만, 타임 스탬프 정보를 찾을 수 없기 때문이다. 소스 파일과 오브젝트 파일의 타임스탬프를 비교하는 이전 규칙과 대조를 이룬다.

install 타겟은 규칙 오른쪽에 파일명이 기술돼 있으므로 GNU Make는 install 타겟이 호출될때 자동으로 calculator프로그램 전체가 최신으로 갱신되는 것을 보장한다. 17번 행의 cp 명령은 실행 파일을 /usr/local/bin (INSTALL_ROOT 변수) 디렉터리로 복사한다.

```
$ gmake install
gcc -g -c -o add.o add.c
gcc -g -c -o calc.o calc.c
gcc -g -c -o mult.o mult.c
gcc -g -c -o sub.o sub.c
gcc -g -o calculator add.o calc.o mult.o sub.o
cp calculator /usr/local/bin
```

install target이 두 번째로 호출 될 때에는 몇 가지 작업을 수행한지만, 그 작업량이 첫번째 호출보다 많지는 않다.

```
$ gmake install
cp calculator /usr/local/bin
```

GNU Make는 calculator 실행 파일이 모든 오브젝트 파일보다 최신의 타임스탬프를 갖고 있다고 판단하면 calculator 실행 파일을 다시 컴파일하지 않는다. 하지만 install이라는 파일이 없으므로 GNU Make는 clean 타겟에서와 같이 cp 명령을 매번 호출한다.

프레임워크 사용

대부분의 빌드 시스템에서의 일반적인 관행은 프레임워크를 만드는 일이다. 즉, 빌드 시스템의 모든 부분이 소프트웨어 개발자가 일일이 신경 쓰지 않아도 될 만큼 별도의 파일들로 관리되는 것이다. 이를 바탕으로 개발자는 주로 관심을 두는 소스 파일 목록과 컴파일 옵션에 집중할 수 있다. 그 결과 대부분의 소프트웨어 개발자는 복잡한 프레임워크를 읽거나 이해할 필요 없이 효율적으로 개발 작업을 할 수 있다.

예를 들어 다음 Makefile은 일반적인 소프트웨어 개발자가 이해하는 데 필요한 정보만을 제공한다.

```
1 SRCS = add.c calc.c mult.c sub.c
2 PROG = calculator
3 HEADERS = numbers.h
4
5 include framework.mk
```

1~3번 행은 가장 기본적인 정보를 제공한다. 즉, 컴파일될 소스 파일 목록과 실행 파일명, 헤더 파일 목록만 기술하고 있다. 이는 간단한 프로글매을 컴파일하는데 필요한 모든 정보를 포함하고 있으므로 소프트웨어 개발자라면 누구든지 관심을 둔다.

5번 행에서는 프레임워크 파일을 포함하는 구문이 있다. 이 프레임워크 파일에는 GNU Make 규칙과 그 밖의 고급 정의가 기술돼 있다.

```
 1 OBJS = $(SRC:.c=.o)
 2 CC = gcc
 3 INSTALL_ROOT=/usr/local
 4
 5 ifdef DEBUG
 6 CFLAGS = -O -g
 7 else
 8 CFLAGS = -O
 9 endif
```

```
10
11 $(PROG):$(OBJS)
12           $(CC) $(CFLAGS) -o $@ $^
13
14 clean:
15     rm -f $(OBJS) $(PROG)
16
17 install: $(PROG)
18     cp $(PROG) $(INSTALL_ROOT)/bin
```

위의 어디에도 소프 파일명과 실행 파일이 기술돼 있지 않다. 이들은 프레임워크에 기술돼 있지 않고 Makefile에 기술돼 있다는 점을 유념하길 바란다. 이 프레임워크에서 주의깊게 봐야할 부분은 5~9행이다. 5번 행에서 DEBUG심볼이 존재하는지 테스트하는데 이 DEBUG 심볼은 프레임워크 파일을 호출하는 Makefile이나 유닉스 커맨드라인에서 설정될 수 있다.

예를 들어 calculator 프로그램의 표준 빌드는 컴파일 시 -O(최적화) 옵션을 이용한다.

```
$ gmake
gcc -O -c -o add.o add.c
gcc -O -c -o calc.o calc.c
gcc -O -c -o mult.o mult.c
gcc -O -c -o sub.o sub.c
gcc -O -o calculator add.o calc.o mult.o sub.o
```

반면에 커맨드라인에서 gmake실행 시 DEBUG 변수 값을 1로 설정하면 컴파일 시 -g 옵션이 활성화된다.

```
$ gmake DEBUG=1
gcc -O -g -c -o add.o add.c
gcc -O -g -c -o calc.o calc.c
gcc -O -g -c -o mult.o mult.c
gcc -O -g -c -o sub.o sub.c
gcc -O -g -o calculator add.o calc.o mult.o sub.o
```