TTK4235 - Tilpassede datasystemer
Vår 2021

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Øving 3
Kompilering med GNU make

Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang

I Introduksjon - Praktisk rundt øvingen

GNU make er et automatisk byggeverktøy som kan gjøre store prosjekter med håndterlig. Automatisk bygging er spesielt nyttig dersom det tar lang tid å gjenkompilere hver eneste fil hver gang en fil endres. Med et automatisk byggeverktøy, kan man i praksis definere et forhåndsbestemt hierarki av hvilken filer som avhenger av hvilke andre filer. Med et hierarki, kan et automatisk byggeverktøy selektivt kompilere kun de filene som har endret seg og filene som er avhengig av de endrede filene.

I praksis har mange programmeringsspråk sine egne byggeverktøy: `gb` for golang, `rake` for ruby, `mix` for elixir, eller `rebar` for erlang. Fordelen med verktøy som *GNU make* (fra nå av kalt `make`) og `ninja` er at de ikke er bundet til et spesifikt språk. Dette gjør `make` veldig allsidig som fører til at `make` kan bygge hva som helst, så lenge brukeren kan kommandoene som skal kalles og i hvilken rekkefølge.

Introduksjon II gir en innføring i bruk av `make`. Oppgavene finner dere i seksjon 1 . I tillegg er det inkludert mer informasjon i Appendiks A som ikke er obligatorisk, men er nyttig for heis-labben for en mer elegant bruk av `make`. For mer informasjon om `make`, ligger manualen (utgitt av Free Software Foundation) ute på https://www.gnu.org.

II Introduksjon - Grunnleggende `make`

`make` bygger prosjektet utifra en "Makefile" ("makefile" er også godkjent, men har lavere prioritet enn "Makefile"). Denne filen definerer et forhåndsbestemt hierarki og består av et sett med regler - som enten er en målfil (en fil som skal bygges), eller et generelt mål (en oppgave som skal utføres, uavhengig av om sluttproduktet er en fil). Alle regler følger samme mønster:

```
mål : ingredienser
	oppskrift
```

II .1 Generell virkemåte

På starten av regelen er det definert et sluttprodukt ("mål"). For å genere målet, forteller man `make` at en rekke ingredienser, spesifisert etter kolonet, er nødvendig. Dersom alle disse ingrediensene er å oppdrive, vil `make` følge oppskriften spesifisert under. Om en eller flere ingredienser mangler, vil `make` forsøke å finne mål som kan bygge dem. **Viktig at det bare er ett eneste tabulatorinnrykk mellom starten av linjen og stegene i oppskriften!**.

Et eksempel på en regel for å genere filen "main.o", kan sees her:

```
main.o : main.c constants.h
	gcc -c main.c -o main.o
```

Denne snutten leses slik: "Filen `main.o` avhenger av filene `main.c` og `constants.h`". For å bygge `main.o`, kalles `gcc -c main.c -o main.o`. Om hverken `main.c` eller `constants.h` har endret seg siden `make` sist bygde `main.o`, vil `make` ikke gjøre noe.

I motsetning til reelle mål, har man også oppgaver som skal fullføres, men som i seg selv ikke produserer en håndfast fil (veldig vanlig å definere disse reglene med deklarasjonen `.PHONY`):

```
.PHONY: clean
clean :
	rm -f *.o
```

Deklarasjonen `.PHONY` er egentlig ikke nødvendig så lenge man ikke har en fil som er kalt "`clean`" i prosjektet. Det er uansett anbefalt å bruke `.PHONY` for leselighet. I tillegg til at denne regelen ikke produserer en håndfast fil, så er ikke `clean` avhengig av noen filer, fordi den ikke spesifiserer noe bak kolonet. Akkurat denne kommandoen er spesielt nyttig, ettersom den fjerner alle objektfilene som har blitt kompilert.

II .2 Nøstede regler

Det er vanlig å nøste regler. Et eksempel kan sees under:

```
taco : ost.o lefse.o saus.o protein.o
	gcc -o taco ost.o lefse.o saus.o protein.o

ost.o : ost.c
	gcc -c ost.c
```

```
lefse.o : lefse.c
	gcc -c lefse.c

saus.o : saus.c
	gcc -c saus.c

protein.o : protein.c
	gcc -c protein.c
```

Dette er en regel for å bygge en tacosimulator "`taco`", som avhenger av en rekke filer (`ost.o, lefse.o, saus.o, protein,o`) som må kompileres før den den kjørebare filen i seg selv kan bygges. Hvis ikke alle objektfilene er til stede, vil `make` fortsette å lete nedover for å finne en regel for å bygge det som mangler. I dette tilfellet vil `make` prøve å bygge `taco` først. Dersom `ost.o` ikke finnes vil `make` kompilere `ost.o` ved å lete nedover i koden. Deretter vil `make` fortsette med `taco`-regelen.

I utgangspunktet er det viktig å spesifisere hvilken av reglene som skal kalles. Dersom man kaller `make` fra kommandolinjen, vil `make` finne den første regelen som ikke starter med et punktum, og så forsøke å utføre den. Alle andre regler vil bli ansett som hjelperegler for toppmålet.

Det er hovedsakelig to måter å overstyre denne oppførselen på: Først og fremst kan man manuelt spesifisere hvilken regel `make` skal behandle, ved å kalle eksempelvis `make tiles.o`. Den andre måten er å spesifisere en variabel kalt `.DEFAULT_GOAL` i makefilen. I følgende eksempel vil `production` bygges, med mindre man eksplisitt ber om `debug`, selv om `debug` er definert før `production`:

```
.DEFAULT_GOAL := production

debug : main.c
	gcc main.c -O0 -g3

production : main.c
	gcc main.c -O3
```

hvor `-O0 -g3` er vanlige flag som blir gitt til `gcc` for å kompilere `main.c`. `-O0` reduserer tiden det tar for å kompilere koden og gjør at debugging produserer forventede resultater, mens `-g3` brukes for å produsere dubbing informasjon. I noen situasjoner kan det være lurt å ikke kompilere `gcc` med `-O0` fordi det fjerner muligheten til å optimalisere ytelsen til koden.

II .3 Variabler i regler

For å ikke ha unødvendig *boilerplate* i reglene, er det vanlig å definere variabler som blir substitutert inn med dollartegnet (`$`). Et eksempel kan sees under:

```
OBJ = ost.o lefse.o saus.o protein.o

taco : $(OBJ)
	gcc -o taco $(OBJ)
```

Konvensjonen er å definere variabler med store tegn (kommer fra shellscripting), men det er opp til brukeren om denne konvensjonen følges eller ei.

II .3.1 De to variantene variabler

`make` har to forskjellige varianter av variabler - *rekursive* og *enkle*. En *rekursiv* variabel ekspanderer til hva enn variabelen referer. For eksempel, vil `default` i kodesnutten under, skrive ut variabelen `KAKE`, som referer variabelen `TYPE`, som igjen referer variabelen `SJOKOLADE`, som til slutt inneholder strengen "`brownie`". I dette tilfellet vil output være strengen `"brownie"` dersom man kaller `make`.

```
KAKE = $(TYPE)
TYPE = $(SJOKOLADE)
SJOKOLADE = "brownie"

default:
	echo $(KAKE)
```

Problemet med *rekursive*-variabler er at man ikke kan skrive koder som dette:

```
CFLAGS = $(CFLAGS) -O0 -g3
```

Denne kodesnutten vil føre til en evig løkke. For å realisere denne type oppførsel kan man bruke *enkle* variabler. Enkle variabler settes med enten `:=` eller `::=`[1]:

```
X := "sjokolade"
Y := "$(X)kake"
X := "gulrotkake"
```

Når `make` kommer over denne kodesnutten, finner den verdien av variablene, og bruker så disse verdiene i resten av byggeprosessen. Denne koden er derfor ekvivalent med denne:

```
Y := "sjokoladekake"
X := "gulrotkake"
```

[1] *GNU make* støtter begge, og de er ekvivalente, men *POSIX* definerer kun `::=`

II .3.2 Måter å sette variabler

I tillegg til at `make` har to forskjellige variabelvarianter, er det mange mulige måter å sette verdiene på disse på. Om man ønsker at en variabel får en verdi, men bare hvis den ikke allerede er definert, bruker man `?=`. For å legge til ledd i en variabel, kan `+=` brukes. For å kjøre et shellscript og tilegne resultatet til en variabel, brukes `!=`.

`make` har i tillegg støtte for å avdefinere en variabel med deklarasjonen `undefine`. `make` kan også definere multilinjevariabler slik:

```
define LINES =
"Linje en"
$(LINJE_TO)
endef
```

II .3.3 Spesielt lange variabellister

Dersom man har spesielt lange variabellister, er det mulig å bruke "\" for å signalisere at linjen ikke ender selv ved et linjeskift. Sett nå at `taco` bestod av flere filer enn `ost`, `lefse`, `saus` og `protein` kan man definere `OJB` som:

```
OBJ = ost.o lefse.o saus.o protein.o tacokrydder.o\
       avokado.o rømme.o bønner.o nachos.o mais.o\
       paprika.o løk.o olje.o
```

II .4 Infererte regler

GNU make har en stor fordel når det kommer til kode som er skrevet med C eller C++. `make` vil anta at en fil kalt `kardemommete.o` avhenger av en tilsvarende `kardemommete.c`-fil. Videre vil `make` anta at kompilatorflagget "`-c`" brukes for å genere objektfiler.

Dette kan brukes for å simplifisere `taco`-eksempelet. Det eneste som gjenstår er å fortelle `make` hvilken kompilator som brukes. Dette vil `make` klare å tyde ut fra hvilken kommando som lenker sammen objektfilene - automatisk. `taco`-eksemplet kan dermed simplifiseres til:

```
OBJ = ost.o lefse.o saus.o protein.o

taco : $(OBJ)
       gcc -o taco $(OBJ)

ost.o :
lefse.o :
saus.o :
protein.o :
```

Siden objektfilene ikke avhenger av noe annet enn de korresponderende `.c`-filene, er det nok å skrive:

```
OBJ = ost.o lefse.o saus.o protein.o

taco : $(OBJ)
       gcc -o taco $(OBJ)
```

Om alle objektfilene viser seg å avhenge av verdiene som er definert i `råvare_pris.h` kan dette beskrives enkelt slik:

```
OBJ = ost.o lefse.o saus.o protein.o

taco : $(OBJ)
       gcc -o taco $(OBJ)

$(OBJ) : råvare_pris.h
```

Det er diskutabelt om denne måten å lage `make`-filer er å foretrekke, siden det ikke lenger er helt klart hva som skjer - men til syvende og sist er det rett og slett et spørsmål om personlig smak.

II .5 Betingelser i regler

Akkurat som i vanlige programmeringsspråk, kan man bruke betingelser for å teste en betingelse før en handling/regel blir utført. Betingelser kan være veldig nyttige, spesielt om man har et prosjekt som skal kunne bygges på forskjellige plattformer[2]. Et eksempel på dette kan sees under:

```
GCC_LIBS = -lgnu
DEFAULT_LIBS = -lsystem_specific

ifeq ($(CC), gcc)
       $(CC) -o prog $(OBJ) $(GCC_LIBS)
else
       $(CC) -o prog $(OBJ) $(DEFAULT_LIBS)
endif
```

hvor `ifeq` betyr **if eq**ual. Det er ikke nødvendig med en `else` for å bruke `ifeq` - og en `else if` bruker syntaksen for `else`:

```
ifeq ($(CC), gcc)
       LIBS = $(GCC_LIBS)
else ifeq ($(CC), clang)
       LIBS = $(CLANG_LIBS)
else
```

[2]Om man først skal støtte flere plattformer, er nok verktøyet `cmake` verdt å ta en titt på

```
        LIBS = $(DEFAULT_LIBS)
endif
```

I tillegg støtter *GNU make* også andre tester enn `ifeq`: eksempelvis `ifneq` for test av ulikhet, `ifdef` for å teste om noe er definert, eller `ifndef` for å teste om noe ikke er definert. For `ifdef` og `ifndef` tar operatoren kun ett argument, ikke to:

```
ifdef $(USE_SYSTEM_LIBS)
        LIBS += -lsystem_specific
endif
```

1 Oppgave (100%) - Grunnleggende make

Deres oppgave er å skrive en enkel makefil for å bygge den utleverte koden. Makefilen skal ha følgende spesifikasjoner:

a Makefilen skal inneholde to regler, i denne rekkefølgen:

(a) `clean`

(b) `taco`

Regelen `clean` skal være *uekte mål*, mens `taco` skal bygge seg selv.

b Filens *default goal* skal være `taco`.

c Definer variabelen `CC` til å være `gcc`. Denne variabelen skal ikke tilegnes rekursivt.

d Definer variabelen `CFLAGS` til å være `-O0 -g3`. Denne variabelen skal ikke tilegnes rekursivt.

e Definer en variabel for alle objektfilene `taco` er avhengig av (hva dere kaller variabelen er opp til dere). Objektfilene er:

(a) `taco_krydder.o`

(b) `taco_saus.o`

(c) `taco_lefse.o`

(d) `protein.o`

(e) `grønnsak.o`

(f) `drikke.o`

(g) `main.o`

f Regelen `clean` skal fjerne alle objektfilene (**Hint:** kommandoen `rm`).

g Regelen `taco` skal bygge programmet `taco` ved å lenke sammen objektfilene. Dere skal bruke variablene `CC` og `CFLAGS`, samt objektvariabelen dere definerte.

h Makefilen skal bruke den spesielle variabelen `$@`

Når dere er ferdige, skal dere bygge den utleverte koden med makefilen for en læringsassistent. For å kjøre filen kan dere bruke `./taco <navn>` hvor navnet er input til programmet. Når dere får til dette, og kan kjøre programmet, er dere klare for godkjenning.

A Appendiks - Mer avanserte funksjoner

Oppgaven dere nå løste var gjort med "brute force". Dette funker, men det finnes andre elegante løsninger. Disse løsningene avhenger på mønstergjennskjenning og dedikerte kilde- eller byggemapper.

Sett at man har et prosjekt som heter `pizza`. Prosjektet består av kildefilene `main.c, pizza_bread.c, pizza_sauce.c` og `pizza_topping.c`. For å holde oversikt er det lurt å ha en dedikert kildemappe der man lagrer kildekoden til prosjektet (vanligvis blir denne mappen kalt "source"). I tillegg er det også ønskelig å ha en mappe for alle kompilerte *artefakter* (vanligvis blir denne mappen kalt "build").

I toppnivåmappen har man make-filen og det ferdige programmet:

```
pizza
├── Make-fil
├── build
│   ├── main.o
│   ├── pizza_bread.o
│   ├── pizza_sauce.o
│   └── pizza_topping.o
└── source
    ├── main.c
    ├── pizza_bread.c
    ├── pizza_sauce.c
    └── pizza_topping.c
```

Det eneste man trenger for å bygge `pizza` er kildefilene (`source`) og make-filen. Det er derfor ønskelig å kunne automatisk opprette byggemappen (`build`) om den ikke finnes. Dette kan gjøres med en *order-only prerequisite*. Når man beskriver hvilke filer `make` trenger for å bygge et mål, kan man bruke vertikal pipe ("`|`") for å fortelle `make` at avhengigheten kun trenger å eksistere:

```
target : dependency_1 dependency_2 | order_only_1 order_only_2
        [commands to build target]
```

Alle avhengigheter som kommer etter `|`-tegnet vil kun bygges dersom de enten ikke

allerede finnes, eller om man eksplisitt ber `make` om å bygge det bestemte målet. Dette er nyttig for å automatisk opprette byggemappen om den ikke finnes.

For å lage en regel om at kompilerte filer skal legges i byggemappen må man overstyre de infererte reglene ved å bruke mønstergjenkjenning. Mønstergjenkjenning brukes for å lage en generisk regel for hvordan en `.c`-fil skal kompileres. For mønstergjenkjenning, brukes tegnet `%`-tegnet. Et eksempel på grunnleggende mønstergjenkjenning kan ses under:

```
%.o : %.c
	gcc -c $< -o $@
```

hvor `$@` og `$<` er automatiske variabler. `$@` referer til målet som blir generert ved å kjøre regelen, mens `$<` referer til første avhengighet i regelen. Akkurat denne regelen definerer byggeprosessen slik at en hvilken som helst `.o`-fil blir generert ved å kompilere den tilsvarende `.c`-filen med `gcc`. Mønstergjenkjenning og *order-only* avhengigheter kan kombineres elegant for å automatisk kompilere `.c`-filene inn i den dedikerte byggemappen:

```
build/%.o : %.c | build
	gcc -c $< -o $@
```

For å gjøre prosessen mer lettvint, er det greit å definere alle avhengighetene (`main.c pizza_bread.c pizza_sauce.c pizza_topping.c`) i en dedikert regel som referer til variablene. Dette gjøres ved å kombinere mønstergjenkjenning og substitusjon:

```
SOURCES := main.c pizza_bread.c pizza_sauce.c pizza_topping.c

SRC := $(SOURCES:%c=source/%c)
```

Denne deklarasjonen vil ta alle .c filene fra variabelen `SOURCES` og legge til mappeprefikset "source". Dermed trenger man ikke å skrive alle avhengighetene (`main.c pizza_bread.c pizza_sauce.c pizza_topping.c`) for hver enkel fil, men man kan bare bruke variabelen `SOURCES`. Den komplette make-filen for `pizza` består av følgende kode:

```
SOURCES := main.c pizza_bread.c pizza_sauce.c pizza_topping.c

BUILD_DIR := build

OBJ := $(SOURCES:%.c=$(BUILD_DIR)/%.o)

SRC_DIR := source
SRC := $(SOURCES:%.c=$(SRC_DIR)/%.c)

CC := gcc
CFLAGS := -O0 -g3 -Wall -Werror
```

```
.DEFAULT_GOAL := pizza

pizza : $(OBJ)
    $(CC) $(OBJ) -o $@

$(BUILD_DIR) :
    mkdir $(BUILD_DIR)

$(BUILD_DIR)/%.o : $(SRC_DIR)/%.c | $(BUILD_DIR)
    $(CC) -c $< -o $@

.PHONY : clean
clean:
    rm -rf $(.DEFAULT_GOAL) $(BUILD_DIR)
```