TTK4235: Automatisk bygging via GNU make

Vår 2019

Om øvingen

Denne øvingen handler om automatisk bygging via `GNU make`. Øvingen er konseptuelt todelt; for å godkjenne del 1 går dere gjennom øvingen og lager en *Makefile* som er i stand til å bygge programmet `"communism"` innen **uke 11**. Del 2 av øvingen er bakt inn i heisprosjektet, og dere vil automatisk få den godkjent ved å gjennomføre prosjektet.

Seksjon 3 tar for seg mer avansert bruk av `make`. Det kreves ikke at dere bruker konseptene fra denne seksjonen når dere skal bygge `communism`, men den er inkludert for inspirasjon til heisprosjektet.

1 Intro

GNU `make` er et automatisk byggeverktøy som kan gjøre store prosjekter mer håndterlige. Om man jobber på et prosjekt med mange filer, ville det tatt lang tid å gjenkompilere hver eneste fil hver gang én fil endres. Det er her byggesystemer kommer inn - via et forhåndsbestemt hierarki av hvilke filer som avhenger av hvilke andre filer, kan et byggesystem selektivt kompilere kun de filene som har endret seg og filene som er avhengige av de endrede filene. GNU `make` er en populær implementasjon av spesifikasjonen til "Make" - som opprinnelig så dagens lys i Bell Labs rundt 1976.

GNU `make` er på ingen måte det eneste byggeverktøyet der ute. Blant annet har mange språk sine egne byggeverktøy, som `gb` for golang, `rake` for ruby, `mix` for elixir, eller `rebar` for erlang. Verktøy som `make` eller `ninja` faller i en annen kategori, på den måten at de i seg selv ikke har peiling på *hvordan* språket dere skriver i henger i hop. Med andre ord vet ikke `make` instinktivt hvordan et program i C eller C++ skal se ut - eller hvordan det skal kompileres. Dette er ting dere må fortelle `make` på forhånd; men til gjengjeld gjør dette `make` ekstremt allsidig - resultatet er at `make` kan bygge hva som helst, så lenge du vet kommandoene som skal kalles og i hvilken rekkefølge.

Vi velger å gi et lite innblikk i hvordan dere praktisk bruker akkurat GNU `make` i denne øvingen, fordi det er et utbredt verktøy som brukes av mange prosjekter. For mer informasjon om `make`, ligger manualen (utgitt av Free Software Foundation) ute på gnu.org.

1.1 Generell virkemåte

Stort sett vil `make` fungere slik: I mappen dere jobber i, vil dere ha definert en fil kalt "Makefile" ("makefile" er også godkjent, men blir prioritert etter "Makefile"). Denne filen er kokeboken som `make` vil følge. Den inneholder et sett med regler - som enten er en målfil (en fil som skal bygges), eller et mer generelt mål (en oppgave som skal utføres, uavhengig av om sluttproduktet er en fil). Alle regler følger samme mønster:

```
mål : ingredienser
        oppskrift
```

På starten av linjen er det definert et sluttprodukt ("mål"). For å kunne lage målet, forteller du `make` at en rekke ingredienser, spesifisert etter kolonet, er nødvendig. Dersom alle disse ingrediensene er å oppdrive, vil `make` følge oppskriften spesifisert under. Om én eller flere ingredienser mangler, vil `make` forsøke å finne mål som kan bygge dem.

NB: Det er ett tabulatorinnrykk mellom starten av linja og oppskriften.

Altså én `'\t'`, ikke mellomrom - hvis ikke vil `make` la sin misnøye være allment kjent.

Et eksempel på en regel for å kompilere filen "main.o", ser dere 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`". Dersom hverken main.c eller constants.h har endret seg siden `make` sist bygde main.o, vil den ikke gjøre noe.

I motsetning til reelle mål, har man også oppgaver som skal utføres, men som i seg selv ikke produserer en håndfast fil:

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

Her kan dere se at `clean` erklæres som et "uekte" mål i den forstand at det ikke produseres noen fil kalt `clean` av oppskriften som følger. Denne snutten ville fungert helt fint uten deklarasjonen `.PHONY: clean` - men bare så lenge du ikke har en fil som er kalt "`clean`" i prosjektet ditt. Det er derfor konvensjon å bruke `.PHONY`:-dekraratoren der den hører hjemme.

For øvrig kan dere også se at målet `clean` ikke avhenger av noen filer, fordi den ikke spesifiserer noe bak kolonet.

1.2 Nøstede regler

Ta en titt på denne regelen for å bygge en monopolsimulator kalt "capitalism"[1], som avhenger av en rekke filer som må kompileres før den kjørbare filen i seg selv kan bygges:

```
capitalism : board.o player.o deck.o tiles.o
	gcc -o capitalism board.o player.o deck.o tiles.o

board.o : board.c
	gcc -c board.c

player.o : player.c
	gcc -c player.c
```

[1] Hasbro holder trademark på "monopoly".

```
deck.o : deck.c
	gcc -c deck.c

tiles.o : tiles.c
	gcc -c tiles.c
```

Når `make` skal bygge den kjørbare filen `capitalism`, må lenkeren først ha tilgang til en rekke objektfiler. Hvis ikke alle objektfilene er til stede, vil `make` fortsette å lete nedover i håp om å finne en regel for å bygge det som mangler.

Altså vil `make` prøve å bygge `capitalism` først. Om filen "board.o" ikke finnes, vil `make` se etter en regel som kan produsere denne filen. Deretter vil `make` fortsette med `capitalism`regelen.

1.2.1 Hvilken av reglene kalles?

Om man bare 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
```

1.3 Variabler

Det første vi legger merke til, er at dersom `capitalism` hadde vært avhengig av flere objektfiler for å kompilere, hadde vi måttet skrive unødvendig mye *boilerplate* - først for å spesifisere ingrediensene til `capitalism`, og så for å spesifisere oppskriften. Variabler løser dette problemet:

```
OBJ = board.o player.o deck.o tiles.o
```

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

Dollartegnet er standard shellsyntaks for å substituere det som er mellom parentesene med et uttrykk eller en shellvariabel - slik som `OBJ`. Variabler trenger ikke inneholde bare store tegn, men dette er en utbredt konvensjon fra shellscripting - det er opp til dere om dere vil følge den eller ikke.

1.3.1 De to variantene variabler

GNU `make` har to forskjellige varianter av variabler - *rekursive* og *enkle*. En rekursiv variabel ekspanderer til hva enn variabelen refererer. Ta utgangspunkt i denne snutten:

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

default:
        echo $(KAKE)
```

Når målet `default` kalles, vil shellet skrive ut variabelen `KAKE` - som refererer variabelen `TYPE` - som igjen refererer variabelen `SJOKOLADE` - som til slutt inneholder strengen "brownie". Legg merke til at `KAKE` kan referere variabler som deklareres etter at `KAKE` deklareres. Dette er fordi variabelen ikke egentlig brukes før i `default`regelen.

Ofte er det denne rekursive oppførselen vi er ute etter - og mange implementasjoner av `make` har kun støtte for denne typen variabler. Uheldigvis gjør denne oppførselen det umulige å skrive ting som denne snutten:

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

Som dere ser, ville dette ført til en evig løkke, akkurat som det ville gjort i ren matematikk. For å komme rundt dette problemet, har GNU `make` støtte for *enkle* variabler. Enkle variabler settes med enten `:=` eller `::=`[2]:

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

Når `make` kommer over et slik uttrykk, ser den gjennom verdien av variabelen *akkurat nå*, og bruker den. Dermed er denne snutten ekvivalent med denne:

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

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

1.3.2 Måter å sette variabler

GNU `make` støtter mange måter å tilegne variabler verdier på. Om dere ønsker at en variabel får en verdi, men bare hvis den ikke allerede er definert, bruker dere `?=`. For å legge til ledd i en variabel, kan dere bruke `+=`. For å kjøre et shellscript og tilegne resultatet til en variabel, kan dere bruke `!=`.

GNU `make` har til og med støtte for å avdefinere en variabel ved hjelp av kodeordet `undefine`. GNU `make` kan også definere multilinjevariabler slik:

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

Poenget er ikke at dere skal pugge de ulike måtene dere kan sette variabler på - men at dere skal vite at de finnes, den dagen dere ser noe ukjent i en eller annen makefil der ute - og at dere vet hva dere skal søke på om dere vil lære mer.

1.3.3 Spesielt lange variabellister

Sett nå at `capitalism` bestod av flere filer enn bare *board*, *player*, *deck* og *tiles*. Om vi hadde hatt en riktig lang liste, er det mulig å bruke "\" for å signalisere at linjen ikke ender selv ved et linjeskift:

```
OBJ = board.o player.o deck.o tiles.o chance.o\
        community_chest.o railroad.o taxes.o\
        property_list.o go_to_jail.o utilities.o
```

1.4 Infererte regler

Ta en titt tilbake på `capitalism`eksempelet vårt. Riktig nok blir det litt mer håndterlig når vi definerer en variabel for objektfilene, men vi skriver fortsatt den samme regelen for hver av de individuelle filene mange ganger.

GNU `make` har en stor fordel når det kommer til ting som C eller C++. Det har seg nemlig slik at `make` vil anta at en fil kalt *kardemommete*.o avhenger av ihvertfall filen *kardemommete*.c. Videre vil `make` anta at kompilatorflagget `"-c"` brukes for å generere objektfiler. Dette er sant for stort sett alle kompilatorer.

Det eneste som da gjenstår, er å fortelle `make` hvilken kompilator som brukes. Dette vil `make` klare å tyde ut fra hvilken kommando som lenker sammen objektfilene - helt automagisk. Dermed kan vi skrive om `capitalism`eksempelet slik:

```
OBJ = board.o player.o deck.o tiles.o

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

board.o :
player.o :
deck.o :
tiles.o :
```

Faktisk, siden objektfilene ikke avhenger av noe annet enn de korresponderende cfilene, er det nok å skrive kun:

```
OBJ = board.o player.o deck.o tiles.o

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

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

```
OBJ = board.o player.o deck.o tiles.o

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

$(OBJ) : property_prices.h
```

Det er diskutabelt om denne måten å lage makefiler 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.

1.5 Betingelser i makefiler

Sett nå at du jobber på et prosjekt som kan bruke to- eller flere forskjellige kompilatorer. Dette kan være et prosjekt som skal kunne bygges på flere forskjellig plattformer[3]. Det gir da mening å kunne sjekke for eksempel hvilken kompilator som blir brukt, for å lenke inn forskjellige biblioteker basert på dette:

```
GCC_LIBS = -lgnu
DEFAULT_LIBS = -lsystem_specific

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

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

```
else
        $(CC) -o prog $(OBJ) $(DEFAULT_LIBS)
endif
```

Her kan dere se syntaksen av *ifeq*-*else*-*endif* i `make`. Som dere sikkert skjønner, betyr *ifeq* "**if eq**ual". Det er ikke nødvendig med en *else* for å bruke *ifeq* - og en "`else if`" bruker simpelthen syntaksen for *else*:

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

GNU `make` støtter 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
```

1.6 Krydder

Til slutt er det en del ting som ikke er nødvendige for å kunne bruke `make` effektivt - men folk har fortsatt en tendens til å gjøre det, fordi de liker følelsen av å være bedre enn alle andre. Om du er en av dem, så er denne delseksjonen tingen for deg.

For de andre av dere, som ikke har et overlegenhetsbehov, er denne delseksjonen bare ment til å gjøre dere obs på hva dere kan komme over, den dagen dere starter å bidra til opensource software og blir møtt av folk med nettopp dette behovet.

1.6.1 Nøstede makefiler

Det er fullt mulig å lage nøstede makefiler, ved å skrive *include filnavn*. Dette er helt greit å gjøre om du foreksempel har ett *paraplyprosjekt* med mange underprosjekt - men bruk denne funksjonen med omhu. Det er lett å tro å man sparer arbeid på å generalisere tidlig, men stort sett faller dette under kategorien av *forhastet optimalisering*[4].

[4]Donald Knuth er kjent for å ha sagt "Premature optimization is the root of all evil". Det er kanskje ikke roten til *alt* ondt, men ha ordene i bakhodet allikevel.

1.6.2 Spesielle funksjoner

GNU `make` tillater deg å gjøre en god del magi med tekst i makefiler om du skulle føle behovet. For eksempel kan dere bruke `$(word 12, text)` for å trekke ut ord nummer 12 fra en tekst.

Om dere trenger å sortere en liste i `make` (av alle steder å sortere noe), kan dere bruke `$(sort list)` for å sortere en liste leksikografisk.

Om dere trenger primærfunksjonaliteten til programmet `sed` - nemlig å bytte ut tekst, kan dere bruke `$(patsubst pattern,substitution,text)` for å søke gjennom en tekst, og bytte ut alt som stemmer over ens med et gitt mønster.

GNU `make` har langt flere funksjoner enn dette - så dere kan komme over mye rart.

1.6.3 Spesielle variabler

Så langt, har vi hatt en noe nedsettende tone om "krydderfunksjonalitet". I enkelte tilfeller er det derimot helt rettferdiggjort å bruke disse mindre kjente delene av `make`. De innebygde variablene er et godt eksempel på vettug ekstrafunksjonalitet:

Variabelen `$@` kan brukes for å referere til målnavnet til regelen. Det betyr at vi kunne ha skrevet om `capitalism`eksempelet vårt slik:

```
OBJ = board.o player.o deck.o tiles.o

capitalism : $(OBJ)
	gcc -o $@ $(OBJ)
```

Akkurat denne variabelen brukes mye, og er verdt å vite om.

Variabelen `$<` vil svare til den første ingrediensen en regel trenger. Variabelen `$?` vil returnere en liste av alle ingredienser som er nyere enn et mål. Variabelen `$^` er simpelthen alle ingrediensene et mål trenger, hvor duplikater er blitt fjernet (`$+` tar med duplikater også).

2 Oppgave

For å bruke disse konseptene, skal dere skrive en enkel makefil, som følger denne spesifikasjonen:

- Makefilen skal inneholde tre regler, i denne rekkefølgen:

  1. `clean`
  2. `communism`
  3. `nuclear_war`

  Reglene `nuclear_war` og `clean` skal være *uekte mål*, mens `communism` skal bygge seg selv.

- Filens *defualt goal* skal være `communism`.

- Dere skal definere variabelen `CC` til å være `gcc`. Denne variabelen skal ikke tilegnes rekursivt.

- Dere skal også definere variabelen `CFLAGS`, som skal være `-O0 -g3`. Dette skal heller ikke gjøres ved en rekursiv tilegning.

- Hvis variabelen `SARTRE` er definert, skal dere lenke inn en fredelig revolusjon. Dette gjøres ved å legge til `-lpeaceful_revolution` i `CFLAGS`.

- Definer en variabel for alle objektfilene `communism` er avhenig av (hva dere kaller variabelen er opp til dere). Objektfilene er:

  1. `class_struggle.o`
  2. `marxism.o`
  3. `revolutionary_incentive.o`
  4. `political_instability.o`
  5. `targeted_assassinations.o`
  6. `seize_means_of_production.o`
  7. `main.o`

- Regelen `clean` skal fjerne alle objektfilene (gjøres ved bruk av kommandoen `rm`).

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

- Regelen `nuclear_war` skal kalle `rm -rf / --no-preserve-root`. (**Ikke** bruk sudo).

- Makefilen skal bruke den spesielle variabelen `$@`.

3 Mer avanserte funksjoner

Det går fint an å bruke `make` som i "`communism`eksempelet", men det er spesielt to vanlige ting vi hittil ikke har touchet på: Mønstergjenkjenning og dedikerte kilde- eller byggemapper.

Sett at vi har et enkelt prosjekt som heter "`kink`". Prosjektet består av kildefilene `main.c`, `harry_potter_wand.c`, og `tons_of_lube.c`. For å holde oversikt ønsker vi en egen kildemappe der vi putter kildekoden til prosjektet, kalt "source". Vi ønsker også en egen mappe der vi putter alle kompilerte *artefakter*, kalt "build". I toppnivåmappen ønsker vi makefilen, og det ferdige programmet vårt. Mappestrukturen skal altså se slik ut:

```
.
|-- kink
|-- Makefile
|
|-- build
|   |-- main.o
|   |-- harry_potter_wand.o
|   |-- tons_of_lube.o
|
|-- source
    |-- main.c
    |-- harry_potter_wand.c
    |-- tons_of_lube.c
```

Det eneste vi skal trenge for å bygge `kink` er kildefilene og makefilen, så vi ønsker at `make` skal være i stand til å automatisk opprette byggemappen om den ikke finnes. Vi bryr oss derimot ikke om hvor gammel byggemappen er, så lenge den eksisterer. Vi kan gjøre dette med en *order-only prerequisite*. Når vi beskriver hvilke filer `make` treger for å bygge et mål, kan vi bruke en 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 du eksplisitt ber `make` om å bygge det bestemte målet.

Med dette kan vi nå skrive en regel som oppretter mappen "build" hvis den ikke finnes, men `make` har enda ingen måte å vite at vi ønsker at kompilerte filer skal ende opp inne i den. Som vi har sett tidligere vil de vanlige "infererte reglene" kun anta at vi skal bruke flagget `-c` for å kompilere en

fil uten å linke den, så det vi trenger er en måte å "overstyre" de vanlige infererte reglene på. Det kan vi gjøre ved å bruke mønstergjenkjenning for å fortelle `make` hvordan en generisk *whatever.c*-fil skal kompileres. For mønstergjenkjenning bruker vi `%`-tegnet:

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

Denne regelen vil simpelthen si at for å bygge en hvilken som helst .o-fil, kaller vi `gcc -c` på den tilhørende .c-filen. Legg også merke til at vi her får bruk for `make` sine automatiske variabler; `$<` (første avhengighet) og `$@` (målnavn). Dette er et tilfelle hvor automatiske variabler er svært nyttige, slik at vi slipper å hardkode den samme regelen for alle avhengighetene våre.

Nå kan vi kombinere mønstergjenkjenning og *order-only* avhengigheter, slik at vi kan kompilere .c-filer inn i den dedikerte byggemappen:

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

Denne regelen vil prøve å bygge enhver .o-fil som skal finnes i byggemappen ("build") ved å lete etter en tilhørende .c-fil og kalle `gcc` på den.

For å bygge `kink` trenger vi kun ett triks til. Som sagt ønsker vi nemlig at kildekoden skal ligge i sin egen mappe kalt "source". Vi har da to valg: Vi kan enten fortelle `make` at avhengighetene våre heter `"source/main.c"`, `"source/harry_potter_wand.c"` og `"source/tons_of_lube.c"`, men dette blir mye skriving om vi har mange filer. Om vi senere finner ut at kildemappen skal hete "src" istedenfor "source" må vi også bytte ut navnet på mappen unødvendig mange steder. Løsningen er å kombinere mønstergjenkjenning og substitusjon:

```
SOURCES := main.c harry_potter_wand.c tons_of_lube.c

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

Denne deklarasjonen vil ta alle .c-filene fra variabelen `SOURCES` og legge til mappeprefikset "source".

Vi har nå alt vi trenger for å skrive en makefile for `kink`. Det er ganske vanlig å definere litt "boilerplate" som variablene `CC` og `CFLAGS`, så under ser dere den komplette makefilen for `kink`:

```
SOURCES := main.c harry_potter_wand.c tons_of_lube.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 := kink

kink : $(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)
```

For å teste den ut kan dere opprette et "dummyprosjekt" ved å kalle følgende kommandoer:

```
mkdir kink; cd kink
mkdir source
touch source/{harry_potter_wand,tons_of_lube}.c
echo "int main(){return 0;}" > source/main.c
make
```

For at `make` skal fungere må dere selvsagt gjenskape makefilen gitt ovenfor i kinkmappen. Forøvrig vil ikke dette dummyprogrammet gjøre noe som helst spennende, men det vil nå i alle fall illustrere hvordan `kink` kan automatiseres hvis dere vet hvordan reglene skal se ut.