Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

TTK4235 - Tilpassede datasystemer
Vår 2024

Øving 1
Introduksjon til Linux

Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang
2024	Terje Haugland Jacobsson Tord Natlandsmyr

1 Introduksjon - Praktisk rundt filene

I denne øvingen får dere utlevert en `.c`-fil i `source`-mappen. Tabellen under lister opp filen som kommer med i `source`-mappen samt litt informasjon om dere skal endre på filen eller om dere skal la den bli i løpet av øvingen.

Filer	**Skal filen(e) endres?**
`source/main.c`	Ja
`Main/*`	Nei
`.github/*`	Nei

2 Introduksjon - Praktisk rundt øvingen

Denne øvingen gir en innføring i Linux, slik at resten av labopplegget fremover ikke foregår i et helt ukjent miljø. Seksjon 4 gir en innføring i grunnleggende kommandoer som er nyttige i Linux, og som trengs for å fullføre oppgavene som følger i seksjon 5 . Dere får godkjent øving 1 ved at dere demonstrerer foran en studass at dere har gjort alle oppgavene i seksjon 5 . I tillegg har to appendikser blitt lagt til (se appendiks A og B) i tilfelle det er noen som enten har lyst til å installere Linux på egen maskin med dual booting, eller jobbe på et Unix-liknende omgivelse for Windows med Cygwin.

3 Introduksjon - Litt om GNU/Linux

Linux (Linux kernel) ble først utviklet av Linus Torvalds på 90-tallet da han studerte ved Universitetet i Helsinki. Linus ble interessert i utvikling av operativsystemer gjennom sine studier, men ble frustrert av datidens lisensbegrensninger på tilgjengelige operativsystemer. I 1991 begynte han å utvikle kjernen (kernel) til det som skulle bli til Linux-kjernen som vi kjenner den i dag. En operativsystemkjerne fungerer som et bindeledd mellom maskinvare og programmer. Den er ansvarlig for å starte systemet og å tildele ressurser til programmer. En kjerne er i seg selv ikke særlig nyttig uten resten av operativsystemet, slik som enhetsdrivere, brukerprogrammer, kompilatorer, osv. Heldigvis eksisterte allerede GNU-prosjektet, som også hadde som mål å tilby fri (som i frihet)[1] programvare, og i grunn bare manglet en velfungerende kjerne. Ved å kombinere Linux-kjernen med GNU-programvare ble Linux til et fullverdig operativsystem som mange i dag omtaler som "GNU/Linux".

GNU/Linux kommer i dag som flere ulike varianter, eller distribusjoner, avhengig av hvilken programvare som brukes. Noen av de mest populære distribusjonene er Debian, Fedora Linux og Arch Linux. Datamaskinene dere skal bruke på lab bruker Ubuntu, som er en variant av Debian med et mer begynnervennlig brukergrensesnitt.

Et av kjerneprinsippene til både GNU- og Linux-prosjektet er at programvare, samt tilhørende kildekode, skal være åpent tilgjengelig og respektere brukerens friheter til å bruke, endre og (re)distribuere den. Dette gjelder også utviklingen av Linux. Hvem som helst kan bidra til Linux-kjernen, og kun et fåtall utviklere jobber med å vedlikeholde prosjektet profesjonelt. Projektenes dugnadsånd har vært en stor faktor i GNU/Linux sin utvikling, og holdningen har spredd seg til å bli en hjørnestein i kulturen til dagens programvareutviklere. Dette er også en av flere grunner til at programvareutviklere foretrekker å bruke Linux både til daglig bruk og profesjonelt. GNU/Linux brukes i dag av Android, ChromeOS, majoriteten av tjenere på internett og en stor andel av alle tilpassede datasystemer. SpaceX sin Falcon 9, NASA sin Perseverance og Tesla sine biler er bare noen få av alle nevneverdige Linux-brukere.

4 Innføring - Grunnleggende kommandoer

Denne seksjonen dekker stort sett det som kreves for å bruke Linux i de kommende øvingene/labene. Ubuntu er varianten/distribusjonen som er installert på Sanntidssalen, men alle kommandoene i denne seksjonen gjelder for enhver distribusjon av Linux-baserte operativsystemer.

Ubuntu har i likhet med Windows et grafisk brukergrensesnitt, men det er stort sett raskere å bruke en terminal når bruken er kjent. Man kan åpne terminalen ved å trykke `Ctrl + Alt + T` på tastaturet, eller ved å åpne *Dash* (øverst til venstre)

[1] "Free as in freedom, not as in beer" - Richard Stallman, Grunnlegger av GNU Prosjektet

og søke etter `terminal`.

Terminalen åpner seg i hjemmemappen, og prompten ser slik ut (avhengig av hvem som har gjort endringer på maskinen før, kan den være annerledes):

```
student@Ubuntu:~$
```

Tildesymbolet (~) forteller brukeren at man er i hjemmemappen, og `$` forteller at man er en vanlig bruker - i motsetning til *root* som er en allmektig bruker (symboliseres med `#`).

4 .1 `pwd`

Første kommando som kan være lurt å vite om er `pwd`. `pwd` brukes for å få en oversikt over hvilken mappe man befinner seg i. Om man kaller `pwd` på sanntidssalen, burde man få noe som ligner på dette:

```
student@Ubuntu:~$ pwd

/home/student
```

4 .2 `ls`

Videre, for å vise innholdet i mappen man befinner seg i kan man bruke kommandoen `ls`. Om man kaller `ls` fra hjemmemappen vil man typisk se disse mappene:

```
student@Ubuntu:~$ ls

Documents Downloads Pictures Music
Public Videos Desktop Templates
```

Dette er de samme mappene som kan ses i Ubuntu sin filutforsker (åpnes ved å trykke ikonet til venstre, eller ved å søke `nautilus` i *dash*). I tillegg kan man inspisere innholdet i en spesifikk mappe ved å kalle `ls mappenavn`.

Typisk med kommandoer i Linux er at man kan legge til flagg i kommandoene for å endre oppførselen til kommandoen. Et flagg som er nyttig for `ls` er `-l`. Ved bruk av `-l`, listes det opp hvilke typer filer som befinner seg i mappen, hvem som har brukerrettigheter, hvem som eier filene, filstørrelse, sist modifikasjonsdato, og navn. For eksempel:

```
student@Ubuntu:~$ ls -l

total 48
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Documents
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Downloads
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Pictures
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Music
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Public
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Videos
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Desktop
drwxr-xr-x 2 student student 4096 Nov 18 01:49 Templates
```

Formatet som `ls -l` produserer, er illustrert i figur 1. Den første bokstaven (`d`) betyr at filen er en mappe[2]. Deretter følger noen bokstaver som forteller hvem som har rettighetene til å endre filen. En `r` betyr at man har leserettigheter, en `w` betyr at man har skriverettigheter, og en `x` betyr at man kan kjøre filen.

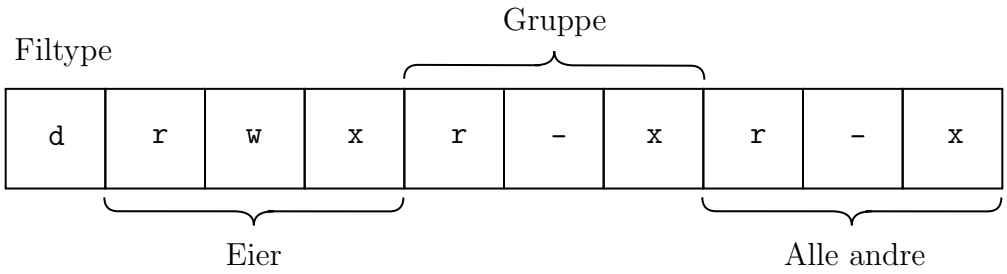


Figure 1: Filegenskaper fra `ls -l`.

I eksempelet i figur 1 kan eieren gjøre hva han eller hun vil, mens gruppen som eier filen og alle andre kan gjøre alt bortsett fra å endre på filen (mangel på `w`).

Etter denne filbeskrivelsen kommer et tall (`2` i eksempelet). Dette er antall *hardlenker* filen har, og har med hvordan filen er lagret på. Deretter følger navn på eieren av filen (`student` i eksempelet), og gruppen som eier filen (også `student` i eksempelet - disse trenger ikke være like). Tallet `4096` som følger hver linje i eksempelet er antall byte som filen okkuperer - men ettersom mapper egentlig bare er pekere som forteller hvilke filer den inneholder, er dette tallet bare størrelsen på pekeren - altså ikke størrelsen på det mappen inneholder. Til slutt kommer en timestamp av når filen sist ble endret, og navnet på filen.

4 .3 `cd`

Kommandoen `cd` er en av de viktigste kommandoene i Linux og blir brukt for å navigere inn- og ut fra mapper. Eksempelvis kaller man `cd Downloads` om man ønsker å bevege seg inn i mappen `Downloads`.

[2]I Linux er mapper også betraktet som filer.

For å bevege seg et nivå opp bytter man ut mappenavnet med to punktum, slik at `cd ..` blir kommandoen. For eksempel om man er i mappen `Downloads` vil `cd ..` navigere brukeren til hjemmemappen.

I Linux har to punktum blitt definert som foreldremappen, mens ett punktum definerer mappen som brukeren befinner seg i. Dette betyr at kommandoen `cd .` flytter brukeren inn i mappen som man allerede befinner seg i.

`cd -` er en annen viktig kommando. `cd -` brukes for å navigere brukeren tilbake til der man sist var.

4 .4 `mkdir`

En nyttig kommando for videre bruk er `mkdir` (*make directory*). Dette er en kommando som brukes for å opprette en mappe. Om man vil opprette en mappe kalt `demo` og bevege seg inn i den kan man eksempelvis kalle:

```
student@Ubuntu:~$ mkdir demo
student@Ubuntu:~$ cd demo
```

4 .5 `touch`

Kommandoen `touch` angir endrings- og tilgangstider for filer. Hvis en fil ikke finnes, opprettes den med standardtillatelser.

```
student@Ubuntu:~$ touch newfile.c
student@Ubuntu:~$ ls

Documents Downloads Pictures Music
Public Videos Desktop Templates newfile.c
```

4 .6 `file`

En annen nyttig kommando er `file`. `file` brukes for å skaffe seg informasjon om en fil. For eksempel:

```
student@Ubuntu:~$ file Downloads
Downloads: directory

student@Ubuntu:~$ file main.c
main.c: C source, ASCII text
```

4 .7 `cat`

Gitt nå at man har lyst til å ta en titt på innholdet i `main.c` som man allerede vet er en `c`-fil på grunn av kommandoen `file`. Da kan man bruke kommandoen

`cat`. `cat` tar innholdet i en fil og skriver det ut i terminalen. For eksempel:

```
student@Ubuntu:~$ cat main.c

#include <stdio.h>

int main(){
    printf("Welcome to TTK4235 - Embedded Systems\n");
    printf("Please be aware of the fact that most animals, especially
        ferrets, are not allowed in Sanntidssalen.\n");
    printf("Smaller birds, e.g. pigeons, may in some very rare cases be
        permitted.\n");
    return 0;
}
```

4 .8 `man`

`man`, kort for *manual page* er den viktigste kommandoen i Linux. Denne kommandoen brukes for å lære om bruken av en hvilken som helst kommando. Et relevant eksempel er kommandoen `man ls` som brukes for å generere en liste over hvilke flagg som `ls` støtter.

I tillegg er det også mulig å kalle `man man` for å få enda mer informasjon om hvordan man kan bruke `man`. Dersom dette blir gjort, kan man se at `man` grupperer manualene i 9 kategorier - hvor kategori 3 er bibliotekkall.

Disse kategoriene kan brukes for å spesifisere hvilke kommandoer man vil ha mer informasjon om, dersom det er flere programmer eller bibliotekkall med samme navn. For eksempel har man programmet `printf`, som man kan få informasjon om ved å kalle `man printf`. Om man derimot vil ha dokumentasjon på funksjonen med samme navn i `C`, kan man kalle `man 3 printf`.

4 .9 `sudo`

Som vanlig bruker kan man ikke bruke alle kommandoer. Noen kommandoer er forbeholdt brukere med ekstra rettigheter (*root*). Om man får `permission denied` når man kaller en kommando, kan man midlertidig eskalere brukerrettighetene til *root*-rettigheter ved å kalle kommandoen med `sudo` foran (kort for **su**per**u**ser **do**).

4 .10 `apt`

Til slutt er det verdt å vite hvordan man installerer nye programmer og pakker. De fleste Linux-distribusjoner bruker en pakkemanager for å håndtere installerte programmer. På denne måten har man en sentralisert løsning for å installere og oppdatere programvarer.

I Ubuntu bruker man `apt` som pakkemanager. For eksempel kan man installere programmeringsspråket `Ruby` sin interpreter ved å kalle `sudo apt install ruby`. Informasjon om hva programvaren `ruby` inneholder kan fås ved å kalle `apt show ruby`.

For å oppdatere alle installerte programvarer til nyeste versjon som Ubuntu har tilgjengelig kaller man `sudo apt update`, etterfulgt av `sudo apt upgrade`. Kommandoen `update` oppdaterer listen over tilgjengelig programvarer som kan oppdateres, mens kommandoen `upgrade` installerer de nyeste oppdateringene. For mer informasjon om hvilke kommandoer `apt` støtter kan man kalle `man apt`.

En ting som er verdt å vite om er at Ubuntu tester ut nye pakker en stund før de legges til oversikten som `apt` har tilgang til. Med andre ord kan det ta en stund før nyeste versjon av programvare kommer til Ubuntu.

4 .11 `nano`

Når man jobber med GNU/Linux er det ikke alltid gitt at man kan bruke tekstredigeringsprogrammer som *Word*, *Visual Studio Code* eller *Notepad++*. Derfor er det lurt å kunne bruke verktøy som fungerer *uten* et grafisk brukergrensesnitt. `nano` er et enkelt tekstredigeringsvektøy som fungerer i terminalen.

```
student@Ubuntu:~$ nano main.c
```

Nederst i terminalvinduet vil du se enkelte kommandoforslag som `^X` for *Exit*. Kontraintuitivt betyr `^` at vi skal bruke `Ctrl`. Dette betyr at for å gå ut av `nano` må vi taste `Ctrl` og `X` etter hverandre. For å forkaste eller lagre endringene du har gjort i en fil må du deretter taste `Y` for "Yes" eller `N` for "No".

4 .12 Kommentar til Tekstredigering

Det finnes veldig mange populære programmer for tekst- og filredigering for GNU/Linux. Noen av dem er `vim`, `neovim`, `emacs`, `gedit`, `sublime-text`, `notepadqq`, `VSCode`[3] og `nano`. Disse varierer både i grensesnitt og brukervennlighet. For små endringer i filer kan alternativer som `nano` og `vim` være gode programmer. Dersom du ikke har noe tidligere erfaring med GNU/Linux anbefaler vi at du bruker enten VSCode eller open-source programmet VSCodium[4] for mer omfattende endringer. Disse programmene er *source-code editors* skreddersydde for programvareutvikling. Videre i øvingsopplegget kommer vi til å bruke `VSCode` som eksempel på hvordan vi kan bruke verktøyene vi lærer om i et utviklingsmiljø.

5 Oppgaver (100 %) - Grunnleggende Linux

a Naviger til rotmappen (`cd /`), og bruk `ls` og `cd` for å komme tilbake til egen hjemmemappe uten å bruke `cd -`.

[3] Kort for *Visual Studio Code*, laget av Microsoft
[4] VSCode er basert på VSCodium

b I Linux er det mange filer som starter med et punktum i navnet. Ulempen med punktum i navnet er at disse ikke blir vist i filutforskeren eller av `ls` med mindre man spesifikt ber om det. Bruk `man` til å finne hvilket flagg som `ls` krever for å vise alle filene i en mappe.

c Linux kan *pipe* output fra en kommando inn i en annen kommando. Et eksempel på det er `cat main.c | less`, hvor returverdien av `main.c` blir gitt som input til kommandoen `less`. Lær mer om `less` med `man` og prøv `cat main.c | less` med `main.c` i `source`-mappen.

d Filutforskerprogrammet som kommer med Ubuntu heter *nautilus*, og kan startes fra terminalen ved å kalle `nautilus`. Prøv å kalle `nautilus . &` hvor punktum blir brukt som argument, mens `&` blir brukt for at terminalen skal starte *nautilus* i bakgrunnen. Bruk `man` til å finne hvilke flagg som kreves for å avslutte `nautilus`.

e Opprett en ny fil med `touch` og *pipe* innholdet fra `main.c` inn i den nye filen. Bekreft at du har kopiert innholdet fra `main.c` over til den nye filen ved å bruke tekstredigeringsverktøyet `nano`.

f Erstatt en av tekststrengene i `printf` med *Linux er lett!* i filen du opprettet i oppgave **e**. *Hint:* Bruk `nano`.

g Oppgrader alle pakkene på datamaskinen på sanntidssalen med `apt`.

A Appendiks - Installasjon av Linux

Det er mulig å installere Linux på egen maskin i tillegg til å bruke maskinene på Sanntidsalen. Dette gjøres enten ved å slette operativsystemet man har nå for så å installere Linux, eller gjennom en ordning kalt *dual booting*. Sistenevnte er spesielt populært og betyr ganske greit at man har to eller flere operativsystemer installert på en datamaskin, så velger man hvilket operativsystem man vil *boote* inn ved oppstart.

Installasjonsprosessen er litt forskjellig fra maskin til maskin, og Linux-variant, men her er en generell beskrivelse av installasjons-prosessen.

A.1 Last ned operativsystemet

Det første man trenger er et *image* av operativsystemet man vil installere. Om dette er første gang man er borti Linux før, er Ubuntu anbefalt. Fra https://ubuntu.com/download kan man laste ned nyeste LTS-utgave (**L**ong **T**erm **S**upport). Dette kommer som en `iso`-fil som må overføres til et oppstartsbart medium - typisk en minnepenn.

A.2 Kopier `iso`-filen til et oppstartsmedium

For å transformere minnepennen til et oppstartsmedium bruker man enten programmer som UNetbootin dersom man kommer fra Windows eller Mac. Om man har tilgang på en annen Linux-maskin kan også kommandoen `dd` brukes:

1. Koble minnepennen i datamaskinen og kall `lsblk` for å se hva minnepennen ble registrert som - typisk som `sdb`.
2. Kall `dd if=~/Downloads/ubuntu[...]amd64.iso of=/dev/sdb bs=4M status=progress`. Om isofilen ligger i en annen mappe enn i `Downloads` endrer man på `if`-adressen. Tilsvarende endrer man på `of`-adressen om minnepennen ble registrert som noe annet enn `sdb`.

A.3 Start fra oppstartsmediet

Etter at man har laget et oppstartsmedium, kan man starte datamaskinen fra denne. Hvordan dette gjøres, varierer sterkt, men ofte kommer man inn i maskinens *boot meny* ved å trykke `F12`, `F8`, eller `F2` mens maskinen *booter*.

På nyere utgaver av Windows kan maskinen også ha *UEFI*-beskyttelse, som først må skrus av før man kan starte fra et oppstartsmedium som ikke først er godkjent av Microsoft. Linux er *ikke* godkjent av Microsoft, så vi må skru av *UEFI*-beskyttelsen.

A.4 Installasjon av Ubuntu med dual booting

Når man først får startet fra oppstartsmediumet, er det bare å prøve ut systemet som man vil, eller installere ved å følge *wizarden* som dukker opp.

B Appendiks - Cygwin

Cygwin er et alternativ til dual booting av Linux dersom man vil fortsatt jobbe på en Window maskin. Dette er en stor samling av GNU og Open-Source verktøy som gir en funksjonalitet som likner veldig på en Linux-distribusjon på Windows.

B.1 Last ned Cygwin

For å installere Cygwin, må man først laste ned installasjonsprogrammet ved å gå inn på Cygwin sin nettside (https://www.cygwin.com/). Installasjonsprogrammet heter `setup-x86.exe` (for 32 bits Windows) eller `setup-x86_64` (for 64 bits Windows). Det kan være lurt å lagre installasjonsprogrammet, ettersom den kan brukes til å oppdatere Cygwin senere.

B.2 Installasjon av Cygwin

For å installere Cygwin, må man kjøre installasjonsprogrammet. Selve installasjonsprosessen er veldig enkel, det eneste man trenger å passe på er å velge **Install from Internet**. Etter dette er det bare å trykke **Next**, **Next**, **Next** og **Next**. Deretter har man muligheten til å velge distribusjonsstedet. I teorien kan man også bare presse **Next** på dette steget. Når man har gjort dette, kommer det opp et vindu. For å installere pakkene man trenger, kan man enten søke etter pakken (se figur 2), eller så kan man bare lete nedover ved å ekspandere de ulike kategoriene. For å informere Cygwin om å installere en pakke, må man dobbelttrykke på **Skip**, helt til man får en versjon opp (i figur 2 installer man `make` ved at man inkluderer `make version 4.3.1`). Deretter trykker man **Next** og **Finish**.

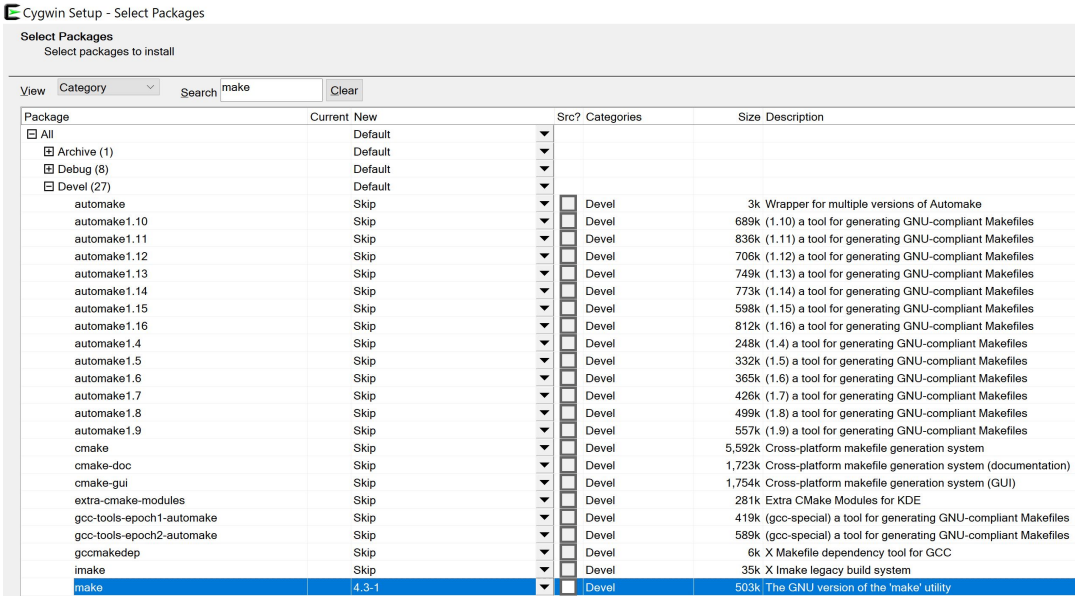


Figure 2: Eksempel på installasjon av pakken `make` i Cygwin.

B.3 Oppdatering av pakker etter installasjon

Det er fort gjort å glemme å installere en pakke. Da kjører man simpelthen installasjonsprogrammet igjen. Like enkelt er det å oppgradere pakker. Installasjonsprogrammet holder styr på det man allerede har installert, og sammenlikner det med det som ligger på det distribujsonsstedet som man velger.

TTK4235 - Tilpassede datasystemer
Vår 2024

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Øving 2
Versjonskontroll med Git

Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang
2024	Terje Haugland Jacobsson Tord Natlandsmyr

I Introduksjon - Praktisk rundt filene

I denne øvingen får dere ikke utlevert noen `.c` eller `.h`-filer.

II Introduksjon - Praktisk rundt øvingen

Versjonskontroll er en måte å ta *bilder* av en fil på, slik at man kan se hvordan filen har endret seg, og når endringene ble gjort. Programmet `git` er et utbredt verktøy for versjonskontroll. Versjonskontroll er spesielt viktig når man jobber flere i lag på de samme filene. `git` ble opprinnelig utviklet av Linus Torvalds i 2005 for å gjøre det lettere for flere programvareutviklere å samarbeide på Linux-kjernen samtidig.

Denne øvingen er ment som en introduksjon til praktisk bruk av `git`. For å illustrere de mest brukte delene av `git` vil øvingen være strukturert som en walk-through og består av å skrive enkel `C`-kode, som skal versjonskontrolleres (introduksjon III). Oppgaven finner dere i seksjon 1 og handler om at dere skal vise at dere mestrer `git` til en studass for å få godkjent øvingen.

II .1 Avhengigheter

Denne øvingen krever at man har `git` på datamaskinen før man begynner. Datamaskinene på Sanntidssalen har `git` installert, men det kan også være greit å skaffe

det selv også, ettersom det er et nyttig verktøy. Dette kan gjøres via https://git-scm.com, eller via en pakkebehandler om operativsystemet har en[1].

I tillegg til `git`, trenger man en `C`-kompilator og tekstbehandler for å kompilere og skrive `C`-kodene vi kommer til å versjonskontrollere. Denne oppgaveteksten kommer til å bruke `gcc`, men en hvilken som helst `C`-kompilator vil fungere (datamaskinene på Sanntidssalen har allerede `gcc` installert).

III Introduksjon - Grunnleggende Kommandoer

III .1 Oppsett

Åpne en terminal og skriv inn `git --version`. Dette er en enkel test for å sjekke om `git` er riktig installert og ligger i `PATH`-variabelen til operativsystemet.

Før `git` kan brukes må man gjøre noen enkle konfigurasjoner. Først og fremst må `git` ha litt informasjon om brukeren. Dette må bare gjøres en gang, og består av to kommandoer:

```
student@Ubuntu:~$ git config --global user.name "Linus Torvalds"
student@Ubuntu:~$ git config --global user.email "linux@stud.ntnu.no"
```

Flagget `--global` fører til at `git` lagrer navn og email i filen `~/.gitconfig`. Om man jobber på en datamaskin andre bruker - slik som i heisprosjektet, er det lurt å konfigurere `git` lokalt [2] (**VIKTIG!**). Det gjøres fra et allerede opprettet `git`-repository, ved å kalle de samme kommandoene uten `--global`-flagget inne i mappen til `git`-repositoriet:

```
student@Ubuntu:~$ git config user.name "Linus Torvalds"
student@Ubuntu:~$ git config user.email "linux@stud.ntnu.no"
```

For ekstra brukervennlighet, har `git` muligheten til å definere aliaser for lange kommandoer som ofte brukes. For eksempel brukes kommandoen `git checkout` ofte slik at mange liker å aliase denne til `git co`. For denne øvingen definerer man aliaset `lg` (betyr det samme som `git log --all --oneline --graph --decorate`):

```
student@Ubuntu:~$ git config --global alias.lg "log --all --oneline
    --graph --decorate"
```

Akkurat dette aliaset er spesielt nyttig for videre bruk av `git`.

[1] apt, yum, pacman etc for Linux. Homebrew for mac

[2] Dette er spesielt viktig når man bruker tjenester for å lagre repositories på nett, f.eks. Github.

III .2 `git init`

`git` er basert på at man har en *oppbevaringsmappe* kalt repository. Om man ønsker å skrive kode i en mappe kalt `demo`, og at `git` skal følge med koden, kan man skrive følgende fra terminalen:

```
student@Ubuntu:~$ mkdir demo
student@Ubuntu:~$ cd demo
student@Ubuntu:~$ git init

Initialized empty Git repository in /home/student/demo/.git/
```

`mkdir` (***m**ake **dir**ectory*) vil opprette mappen `demo`, og `cd` (***c**hange **d**irectory*) vil flytte brukeren inn i den. Til slutt vil kommandoen `git init` opprette en skjult mappe med navn `.git` inne i demo-mappen, som `git` vil bruke for å holde styr på filene i mappen.

III .3 `git status, git add, git commit`

Dere har nå en tom `git`-mappe (`demo`). Kall kommandoen `git status`. Hvis dere ikke har gjort noen endringer i mappen så langt, vil dere få tilbake en melding som dette:

```
student@Ubuntu:~$ git status

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Stort sett er `git` veldig behjelpelig med å fortelle brukeren hva som foregår. Om man lurer på hva som skjer, er utskriften fra `git` oftest et godt svar.

Opprett nå en fil kalt `main.c` i demo-mappen, og skriv inn det følgende med en hvilken som helst tekstbehandler:

```
#include <stdio.h>

int main(){
        return 0;
}
```

Dersom dere nå lagrer denne filen, og kjører `git status` på nytt burde dere se noe slikt eller liknende avhengig av hvilken versjon av `git` dere har:

```
student@Ubuntu:~$ git status
```

```
On branch master

No commits yet

Untracked files:
    (use "git add <file>..." to include in what will be committed)
          main.c

nothing added to commit but untracked files present (use "git add" to
    track)
```

Denne utskriften forteller oss at vi nå har en ny fil i `demo`-mappen ved navn `main.c`, som `git` foreløpig ikke bryr seg om. Kall nå `git add main.c`, etterfulgt av `git status` for å få opp denne meldingen:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git status

On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:    main.c
```

Dette betyr at `git` har lagt til `main.c` i sitt *staging area*, som er stedet før `git` *tar et bilde* av mappen. Dersom man nå skriver `git commit -m "added main.c"` etterfulgt av `git lg`[3] vil man se noe slikt:

```
student@Ubuntu:~$ git commit -m "added main.c"

[master (root-commit) 7cb84fb] added main.c
 1 file changed, 5 insertions(+)
 create mode 100644 main.c

student@Ubuntu:~$ git lg

* 7cb84fb (HEAD -> master) added main.c
```

Det siste vi ser på denne linjen er en stjerne. Denne stjernen representerer et *bilde* som `git` har tatt for oss. De sju heksadesimale tallene som følger etter er et utsnitt av en hash på 40 bokstaver, som `git` bruker for å identifisere *bilder* (eller *commits*). Denne hashen er spesielt viktig, om man vil inspisere tidligere *bilder* som `git` har lagret.

[3] Dette er aliaset vi tidligere opprettet og defineres som `git log --all --oneline --graph --decorate`

I parentesen står det (`HEAD -> master`), som betyr at hodepekeren til `git` peker på akkurat dette *bildet*, som befinner seg på grenen `master`. Til slutt står det `added main.c` som er *commit*-meldingen vi ga dette bildet, for å beskrive hva vi har gjort.

III .4 Terminologi og `gits` indre

Figur 1 viser arbeidflyten som kan forventes med `git`,

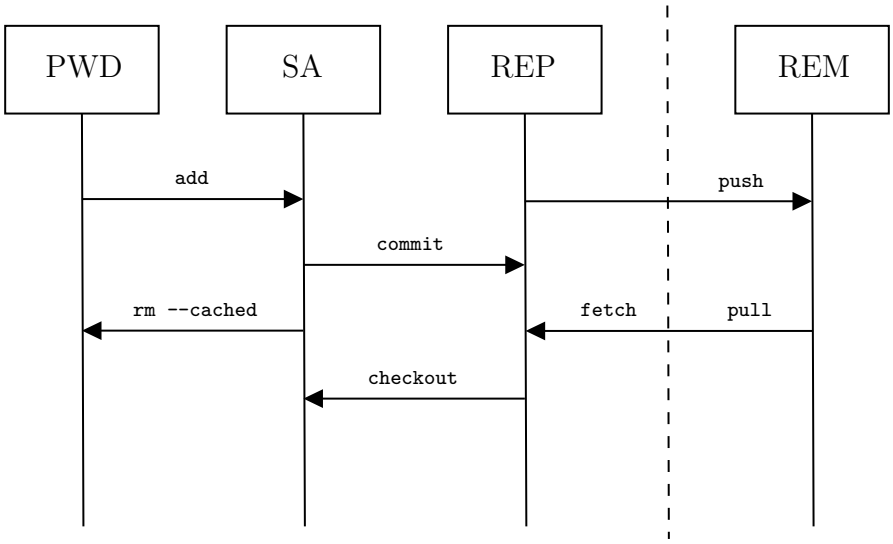


Figure 1: Arbeidsflyt i `git`.

hvor:

- **PWD**: *Current working directory*, dette er mappen som `git` holder styr på.
- **SA**: *Staging area*, her legges filer `git` skal *ta bilde av*, før de legges til i historikken.
- **REP**: *Repository*, dette er all historien `git` kjenner til. Bilder som er tatt av tidligere versjoner av filer legges til her, som en ny stjerne i en graf som representerer alt som har skjedd hittil.
- **REM**: *Remote*, dette er stort sett en ekstern server (men kan være en annen lokal mappe), dit `git` vil dytte lokale endringer, og ta endringer gjort av andre fra.

Til nå har vi vært innom **PWD**, **SA**, og **REP** ved at vi har laget en enkel `C`-kode (`main.c`) som vi har tatt *bilde* av med `git`. Vi kommer ikke til å sette opp eksterne servere, så vi kommer ikke borti **REM**[4]. For å få mer kunnskap om `git`, skal vi nå bygge videre på den lokale `git`-grafen vår.

III .5 `git diff, git checkout`

Endre `main.c` til å inneholde dette:

```
#include <stdio.h>

int main(){
        printf("Hello world\n");
        return 0;
}
```

Kjør deretter `git status`. Dere vil nå se:

```
student@Ubuntu:~$ git status

On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   main.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Dette forteller oss at `git` vet at `main.c` har endret seg, men fordi vi ikke har lagt den til i *staging area*, vil ikke `git` ta et bilde av den.

[4] I praksis brukes ofte Github eller Gitlab som ekstern server for lagring av repositories.

For å få en oversikt over hva som har endret seg siden sist, kan man bruke kommandoen `git diff`. Dersom vi nå kaller `git diff main.c` vil vi få:

```
student@Ubuntu:~$ git diff main.c

@@ -1,5 + 1,6 @@
 #include <stdio.h>\newline
 int main(){
+       printf("Hello world\n");
        return 0;
}
```

hvor et pluss-tegn representerer en linje som har blitt lagt til, mens et minus-tegn representerer en linje som har blitt tatt bort. `git diff` er spesielt viktig dersom man har gjort mange endringer på en gang, og ikke husker hva man har gjort.

Dersom dere nå kjører `git add main.c` og `git commit -m "classic example code"`, etterfulgt av `git lg` vil dere se:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "classic example code"

[master df7e5b2] classic example code
 1 file changed, 1 insertion(+)

student@Ubuntu:~$ git lg

* df7e5b2 (HEAD -> master) classic example code
* 7cb84fb added main.c
```

Dette betyr at vi nettopp tok et nytt bilde av `main.c`, og at vi la denne til øverst i *historikktreet*. Den *gamle* koden som ikke gjorde noe ligger fortsatt i `git`s minne (med hashen `7cb84fb`), men grenen kalt `master` (og også vår hodepeker) peker til den nye `printf`-koden vi akkurat skrev som har fått hashen `df7e5b2`.

Det som gjør `git` veldig nyttig er at det er mulig å få tidligere *bilder*, ved å hoppe tilbake i historikk-grafen. Dersom dere nå kjører `git checkout 7cb84fb`[5] vil dere få en melding som sier at dere er i `detached HEAD state`. Her kan man leke med den gamle koden som `git` har tatt vare på. Om man nå kaller `git checkout master` kommer man tilbake til den nye koden.

III .6 `git branch, git merge`

Når dere jobber på samme kodebase, kommer versjonskonflikter til å oppstå. Dette kan lett bli håndtert med `git branch` og `git merge`.

[5] Hashsummen deres kan være annerledes enn oppgaveteksten. Hashen til den gamle koden kan fås ved å kjøre `git lg`.

Kall først `git branch other`, etterfulgt av `git checkout other`. Dette vil lage en ny gren, kalt `other`, og hoppe til den. Denne grenen skal simulere at dere er to som jobber på samme kode. Dette er så vanlig at `git` har en innebygd kommando for akkurat dette: `git checkout -b <grennavn>`.

Om dere nå kaller `git lg`, vil dere se følgende:

```
student@Ubuntu:~$ git checkout -b other

Switched to branch other

student@Ubuntu:~$ git lg

* df7e5b2 (HEAD -> other, master) classic example code
* 7cb84fb added main.c
```

Nå har vi to grener som begge peker til den nyeste koden, men vi er på grenen `other`, og ikke `master`. La oss si at vi nå endrer på `main.c` slik:

```
#include <stdio.h

int main(){
        printf("Hello world\n");
        printf("...and Mars\n");
        return 0;
}
```

Kjør så `git add main.c` og `git commit -m "greet mars as well"`. Dersom dere nå kjører `git lg` får dere:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "greet mars as well"

[other eb331fb] greet mars as well
 1 file changed, 1 insertion(+)

student@Ubuntu:~$ git lg

* eb331fb (HEAD -> other) greet mars as well
* df7e5b2 (master) classic example code
* 7cb84fb added main.c
```

Her ser vi at grenen `master` fortsatt ligger på koden med `"Hello world"`, mens grenen `other` ligger på koden med `"...and Mars"`.

Sett nå at dere er to som jobber i par, og at en har skrevet `world`-versjonen av koden og en har skrevet `Mars`-versjonen av koden. Dersom den som har skrevet `world`-versjonen nå skriver (bruk `git checkout master` for å bytte tilbake til

`master`-grenen):

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
    if(1 > 0){
            return 1;
    }
    return 0;
}
```

Om man nå kjører `git add main.c`, `git commit -m "assert truth"`, og `git lg` får vi en historikkgraf som ser slik ut:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "assert truth"

[master 835af8f] assert truth
 1 file changed, 3 insertions(+)

student@Ubuntu:~$ git lg

* 835af8f (HEAD -> master) assert truth
| * eb331fb (other) greet mars as well
|/
* df7e5b2 classic example code
* 7cb84fb added main.c
```

Dette representerer at dere var enige på committen med hash `df7e5b2`, men at dere deretter har divergert til hver deres gren (nye `world`-grenen har fått hashen `835af8f`, mens `Mars`-grenen har fått hashen `eb331fb`). Dersom vi nå kaller `git merge other`, for å sammenslå `other`-grenen inn i `master`-grenen vil `git` klage med en feilmelding som burde seg noe slikt ut:

```
student@Ubuntu:~$ git merge other

Auto-merging main.
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result
```

For å få mer informasjon, kan man kalle `git status`:

```
student@Ubuntu:~$ git status

On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

```
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   main.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Her er `git` veldig behjelpelig og forteller brukeren nøyaktig hva som foregår: Vi holder på med en sammenslåing, men `git` vil ikke fullføre, fordi både `master` og `other` har endret på `main.c`.

For å fikse dette problemet kan man gjøre 2 ting:

1. Vi kan fikse konflikten ved at man endrer koden slik at den inneholder begge kodene og kjøre `git commit` for å manuelt fullføre sammenslåingen.
2. Vi kan kjøre `git merge --abort`, om vi ikke lenger vil slå sammen grenene.

Dersom man velger å gå for alternativ 1 siden vi helst vil beholde koden fra begge parter, kan man først åpne `main.c` på nytt for å se endringene som har blitt gjort (koden blir endret automatisk etter at man har kjørt `git merge`):

```
#include <stdio.h>

int main(){
   printf("Hello world\n");
<<<<<<< HEAD
   if(1 > 0){
      return 1;
   }
=======
   printf("...and Mars\n");
>>>>>>> other
   return 0;
}
```

Her kan man se at `git` automatisk har satt inn konfliktmarkører der koden var forskjellig. Alt mellom `<<<<<<< HEAD` og `=======` var på `master`-grenen, mens alt som ligger mellom `=======` og`>>>>>>> other` lå på `other`-grenen.

For å fortelle `git` at konfliktene er tatt hånd om, må man redigere filen slik den skal være, og så legge den til i `git` på vanlig måte. Dette gjøres ved å endre `main.c` til koden under:

```
#include <stdio.h>

int main(){
       printf("Hello world\n");
       printf("...and Mars\n");
```

```
        if(1 > 0){
                return 1;
        }
        return 0;
}
```

for å derette kjøre `git add main.c`, etterfulgt av `git commit -m "conflict solved"`. Hvis man nå kaller `git lg` har vi denne historikkgrafen:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "conflict solved"

[master 3bff360] conflict solved

student@Ubuntu:~$ git lg

*   3bff360 (HEAD -> master) conflict solved
|\
| * eb331fb (other) greet mars as well
* | 835af8f assert truth
|/
* df7e5b2 classic example code
* 7cb84fb added main.c
```

Altså er kodebasene nå slått sammen, men som dere ser, vil ikke `other`- grenen automatisk trekkes etter. Denne grenen kan fjernes ved å kalle `git branch -d other`:

```
student@Ubuntu:~$ git branch -d other

Deleted branch other (was eb331fb).

student@Ubuntu:~$ git lg

*   3bff360 (HEAD -> master) conflict solved
|\
| * eb331fb greet mars as well
* | 835af8f assert truth
|/
* df7e5b2 classic example code
* 7cb84fb added main.c
```

III .7 `git help`

Dersom man er helt lost, så burde man bruke kommandoen `git help`. Dette er en kommando som er spesielt nyttig dersom man vil ha en oversikt over mulige flagg som enhver `git`-kommando støtte. I tillegg gir den litt informasjon om `git`-kommandoen selv. Eksempelsvis, `git help commit` får opp hjelpesiden til `git commit`.

1 Oppgave (100 %) - Grunnleggende Git

For å få godkjent øvingen, skal dere vise at dere har skjønt det meste av walk-throughen. Dere skal derfor vise hva dere har gjort til en studass og besvare noen spørsmål fra studassen på sal, før dere får øvingen godkjent. Dere oppfordres også til å utforske `git` på egenhånd ved å bruke `git help` eller `git help tutorial`, ettersom dere kommer til å bruke `git` til heisprosjektet (og videre i arbeidslivet).

2 Oppgave (anbefalt) - GitHub

Når man bruker `git` i praksis, er det vanlig i kombinasjon med en såkalt *code hosting platform*. GitHub er det mest brukte eksempelet på dette, men det finnes også andre alternativer som Bitbucket, GitLab, mm. I TTK4235 anbefaler vi bruk av GitHub, og som student ved NTNU får man faktisk tilgang til GitHub Pro.

For å ta i bruk GitHub må man først lage seg en bruker, og dette gjøres på nettsiden https://github.com/. Hvis man registrerer seg med stud-mailen gjør dette ting litt enklere når man skal oppgradere til Pro-bruker. Hvordan dette gjøres lar vi dere finne ut av på egenhånd.

Når man har laget seg en bruker og logget inn er det på tide å lage sitt første *repository*. Dette gjøres ved å trykke på den grønne knappen på hovedsiden,

hvor det enten står `New` eller `Create repository`. Her får man muligheten til å velge navn under *Repository name* og beskrivelse under *Description*. Man får også muligheten til å bestemme om *repositoryet* skal være *Public* eller *Private*, altså om andre internett-brukere skal kunne se det eller ikke. Når man er fornøyd med valgene sine kan man trykke på `Create repository` for å fullføre prosessen. Deretter kommer man til hovedsiden til ditt nylagde *repository*.

Nå er det på tide å lage et såkalt *token*. Dette er et alternativ til å bruke passord til autentisering, når man bruker kommandolinja, altså terminalen, i Linux. Det finnes to typer *tokens*, nemlig *fine-grained personal access tokens* og *Personal access tokens (classic)* . Vi skal benytte oss av sistnevnte, ettersom de er bittelitt enklere å bruke. Et *token* knyttes typisk opp mot et *repository*, og brukes for å aksessere dette *repositoryet* fra terminalen. Følg disse stegene for å lage et *classic personal access token*:

1. Trykk på profilbildet ditt (øvre høyre hjørne), og deretter *Settings*.
2. Deretter trykker du på *Developer settings* i den venstre sidebaren.
3. Deretter trykker du på *Personal access tokens*, og så *Tokens (classic)* i den venstre sidebaren.
4. Trykk på *Generate new token*, og deretter *Generate new token (classic)*.
5. Under *Note*, skriv inn et navn til ditt *token*. Dette kan f.eks. være det samme som navnet på *repositoryet* du generelt kommer til å bruke det til.
6. Under *Expiration* velger du hvor lenge det skal være gyldig.
7. Under *Select scopes* velger du de rettighetene du vil at *tokenet* skal ha. Til våre formål i TTK4235 kan det være greit å ha så mange rettigheter som mulig, men dersom sikkerhet hadde vært av større viktighet, hadde vi begrenset dette også. Kryss derfor av i alle boksene.
8. Trykk på *Generate token* for å fullføre prosessen. Husk å kopiere *tokenet* et trygt sted, ettersom du ikke får muligheten til å se det igjen. Det er også alltid mulig å lage et nytt *token* dersom uhellet skulle være ute.

Nå er vi i stand til å klone *repositoryet* vårt ved å bruke følgende kommando: `git clone https://github.com/user_name/repository_name.git`, der "user_name" og "repository_name" byttes ut med henholdvis GitHub-brukernavnet og *repository*-navnet ditt. Her vil du bli spurt om å skrive inn brukernavnet ditt og ditt passord (her skriver du inn ditt *token*), og så har du klonet ditt *repository*.

Nå kan du for eksempel lage en enkel tekstfil i *repositoryet* via terminalen, og deretter bruke `git add file_name` for å legge til denne filen i *staging area*. Så kan du *comitte* gjennom kommandoen `git commit -m "first commit"`, og deretter *pushe* ved kommandoen `git push`. Du har nå laget et GitHub-*repository*, og gjort en *commit*. Hvis du tar en titt på hovedsiden til ditt *repository* vil du nå se endringene som har blitt gjort. Arbeidsflyten ellers er lik som forklart tidligere

i denne øvingen, bortsett fra at *merge* med fordel kan gjøres i nettsiden til selve *repositoryet*.

I dette eksempelet gjorde vi en *commit* rett inn i hovedgrenen `main`. Dette bør man egentlig aldri gjøre, og vi anbefaler å gjøre *commits* i andre grener, og deretter å *merge* i GitHub sin nettleser. Dette er en svært vanlig konvensjon som hovedsaklig går ut på å holde `main` kjørbar. I tillegg vil vi nevne at dere **ikke** bør bruke *global credentials* (f.eks. `--global`-flagget i `git config`) når dere jobber på offentlige datamaskiner, noe dere vil gjøre i TTK4235. For mer informasjon om `Git` og `Git`-konvensjoner anbefaler vi å ta en titt på boken *Pro Git* skrevet av Scott Chacon, som er tilgjengelig gratis på nett.

TTK4235 - Tilpassede datasystemer
Vår 2024

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Øving 3
Kompilering med GNU make

Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang
2024	Terje Haugland Jacobsson Tord Natlandsmyr

I Introduksjon - Praktisk rundt filene

I denne øvingen får dere utlevert noen `.c` og `.h`-filer under `source`-mappen. Tabellen under lister opp alle filene som kommer med i `source`-mappen samt litt informasjon om dere skal endre på filene eller om dere skal la dem forbli uendret.

Filer	**Skal filen(e) endres?**
`source/main.c`	Nei
`source/drikke.c`	Nei
`source/grønnsaker.c`	Nei
`source/protein.c`	Nei
`source/taco_krydder.c`	Nei
`source/taco_lefse.c`	Nei
`source/taco_saus.c`	Nei
`Main/*`	Nei
`.github/*`	Nei

II Introduksjon - Praktisk rundt øvingen

GNU make er et automatisk byggeverktøy som kan gjøre store prosjekter lettere å håndtere. 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 hvilke filer som avhenger av

hvilke andre filer. Med et slikt 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`, og `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 III 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.

III 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
```

III .1 Generell virkemåte

På starten av regelen er det definert et sluttprodukt (et mål). For å genere målet, forteller man `make` at en rekke ingredienser, spesifisert etter kolonnen, 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 ingenting skje.

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 kolonen. Akkurat denne kommandoen er spesielt nyttig, ettersom den fjerner alle objektfilene som har blitt kompilert.

III .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 selve den kjørbare filen i 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, mens `-g3` brukes for å generere debugging-informasjon. I noen situasjoner kan det være lurt å ikke kompilere `gcc` med `-O0` fordi det fjerner muligheten til å optimalisere ytelsen til koden.

III .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.

III .3.1 De to variabel-variantene

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

```
LEFSE = $(MEL)
MEL = $(KORN)
KORN = "karbohydrater"

default:
	echo $(LEFSE)
```

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"
```

III .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 avdefinerisering av en variabel med deklarasjonen `undefine`. `make` kan også definere multilinjevariabler slik:

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

III .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`. Man kan da 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
```

III .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

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

`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.

III .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:

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

```
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` med `ifeq` foran:

```
ifeq ($(CC), gcc)
       LIBS = $(GCC_LIBS)
else ifeq ($(CC), clang)
       LIBS = $(CLANG_LIBS)
else
       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 et *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) `grønnsaker.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 <elevens-navn>` hvor `elevens-navn` er input til programmet. Når dere får til dette, og kan kjøre programmet, er dere klare for godkjenning.

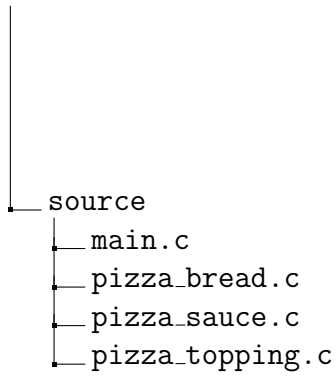
A Appendiks - Mer avanserte funksjoner

Oppgaven dere nå løste var gjort med 'brute force'. Dette funker, men det finnes andre elegante løsninger. Disse løsningene er basert 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 makefilen og det ferdige programmet:

```
pizza
├── Makefile
└── build
    ├── main.o
    ├── pizza_bread.o
    ├── pizza_sauce.o
    └── pizza_topping.o
```



Det eneste man trenger for å bygge `pizza` er kildefilene (`source`) og makefilen. 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 `%`. 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`, og `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`, og `pizza_topping.c`) for hver enkel fil, men man kan bare bruke variabelen `SOURCES`. Den komplette makefilen 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)
```