Föreläsning 2

Vi har i förra föreläsningen gått igenom allmänt kring vad ett operativsystem är, vad det finns för olika typer av programvara (användarprogram, systemprogram), vi har också börjat beskriva ett *UNIX*-system och hur man kan kontrollera ett sådant textmässigt via en så kallad kommandotolk. Då studerade vi kommandona `sleep`, `whoami`, `pwd`, `man`, `info`, `ls`, `cd`, `mkdir` och `rmdir`. På övningen fick ni i uppgift att själva undersöka kommandona `clear`, `echo`, `date` och `mv`.

Idag ska vi titta på mer av de grundläggande kommandona. Dagens lista på kommandon är `touch`, `rm`, `cp`, `cat`, `emacs`, `gcc` och `python`. Med dessa kommandon kommer ni att kunna slutföra grundkursen i programmering som kommer efter denna kurs. Vi kommer också att börja studera programmering av *Guido van Robot*. Vi ska också att skapa ordning och reda genom att faktiskt inte spara några dokument i den virtuella maskinen, vi ska lagra alla dokument på värdmaskinen via någonting som kallas för *delade kataloger*, på engelska *shared folders*. Då kommer vi att ytligt beröra kommandot `mount`. Slutligen ska vi beskriva någonting som heter *Peer Instruction* som är en alternativ undervisningsform som vi kommer att tillämpa nästa gång vi ses. Glöm inte Guido!

Fortsatt fil- och kataloghantering

De kommandon vi studerat i förra föreläsningen och övningen behandlade orientering (`cd`, `pwd`, `whoami`), dokumentation (`man`, `info`), och fil- och kataloghantering (`mkdir`, `rmdir`, `mv`). De kommandon vi ska studera nu fullbordar framställningen av det ni behöver för att någorlunda fullständigt kunna hantera filer. Vi ska först dock ge några kommentarer om vad en fil är i *UNIX*-sammanhang.

Vad är en fil?

I *UNIX*-sammanhang är begreppet fil väldigt brett. Vi ska dock inte ta in filbegreppet i dess vidaste betydelse, utan för oss, i denna kurs, nöjer vi oss med att anse att en fil är någonting som finns i filhierarkin med rotkatalogen `/` överst. Vi anser att kataloger är en speciell form av filer och *reguljära* filer, som finns på någon form av lagringsmedium (som tex en hårddisk eller CD-skiva) har två saker: dels ett namn som vi hittar i filhierarkin och dels ett associerat lagringsutrymme (på det lagringmedium som filer är placerad på.) Sammanfattningsvis: både kataloger och reguljära filer är filer och har namn som återfinns någonstans i filhierarkin med `/` överst.

`touch` och `rm`

En fil har en tidpunkt då den skapades, detta kallas time stamp och vi kan se den time stamp om vi använder `ls -l`. Med kommandot `touch` kan vi faktiskt ändra time stamp på en fil. Vi kommer troligen inte att behöva göra detta någonsin, men touch är ett intressant kommando som är bra att använda som studieobjekt. Vi passar också på att införa kommandot rm. Det är kommandot som aman använder för att ta bort filer. Namnet på kommandot rm är en förkortning för remove som ju betyder ta bort på engelska. Vi ser på en sekvens av kommandon som utförs i en tom katalog som exemplifierar kommandona `touch` och `rm`. Vi väver också in användning av kommandona `pwd` och `ls -l`. Katalogen som vi utför kommandona i heter `/home/me/tom/` och innan denna sekvens av kommandon utförs har vi gjort `mkdir tom` och `cd tom`.

```
$ pwd
/home/me/tom
$ ls -l
totalt 0
$ touch fil1 fil2 fil3
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.26 fil1
-rw-r--r-- 1 me me 0 15 apr 13.26 fil2
-rw-r--r-- 1 me me 0 15 apr 13.26 fil3
```

De tre filerna `fil1`, `fil2` och `fil3` finns inte då vi ger kommandot `touch fil1 fil2 fil3` så de skapas i respons till detta kommando. De har alla storleken 0 (de är alltså tomma) och samma time stamp: `15 apr 13.26`. I samma katalog ger vi senare ytterligare kommandon, vi ser på resultatet:

```
$ touch fil1 fil2
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.34 fil1
-rw-r--r-- 1 me me 0 15 apr 13.34 fil2
-rw-r--r-- 1 me me 0 15 apr 13.26 fil3
```

Vi ändrar alltså time stamp på `fil1` och `fil2` men inte `fil3`. Vi studerar nu hur kommandot rm fungerar via en ny sekvens av kommandon som vi utför precis efter ovanstående:

```
$ touch fil3 fil4
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.34 fil1
-rw-r--r-- 1 me me 0 15 apr 13.34 fil2
-rw-r--r-- 1 me me 0 15 apr 13.38 fil3
-rw-r--r-- 1 me me 0 15 apr 13.38 fil4
$ rm fil1
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.34 fil2
-rw-r--r-- 1 me me 0 15 apr 13.38 fil3
-rw-r--r-- 1 me me 0 15 apr 13.38 fil4
$ rm fil2 fil3
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.38 fil4
```

Vi börjar först med att göra `touch fil3 fil4`. Eftersom `fil3` redan finns så ändras dess time stamp och eftersom `fil4` inte finns så skapas den. Sedan tar vi bort `fil1` med `rm` och sedan tar vi bort både `fil2` och `fil3` med `rm`. Vi studerar löpande tillståndet på katalogen `tom` och dess innehåll genom att ge kommandot `ls -l` mellan varje kommando. Som alltid ska vi också studera

dokumentationen till de kommandon vi använder. Om vi öppnar manualsidorna till `touch` respektive `rm` ser vi följande texter:

```
NAME
      touch - change file timestamps

SYNOPSIS
      touch [OPTION]... FILE...

DESCRIPTION
      Update the access and modification times of each FILE to the
      current time.
```

respektive

```
NAME
      rm - remove files or directories

SYNOPSIS
      rm [OPTION]... FILE...

DESCRIPTION
      This  manual page documents the GNU version of rm.  rm removes
      each specified file.  By default, it does not remove
      directories.
```

Dessa texter beskriver beteendet hos de kommandon vi undersökt. Vi kan observera att kommandot `rm` tydligen, enligt manualsidan, kan användas för att ta bort kataloger. Längre ner i manualsidan kan man läsa sig till att det går att ta bort kataloger om man lägger på `-r`, för *recursive*, alltså man skriver `rm -r katalog1` och då kommer katalog1 att tas bort även om den inte är tom. Det är ibland riskabelt att göra så eftersom hela innehållet i katalogen försvinner. I början så rekommenderar vi er att använda er av `rmdir` när kataloger ska tas bort eftersom detta kommando kräver att katalogen som tas bort är tom.

`cp` och `cat`

De sista kommandona som har med filhantering att göra (som vi studerar här) är `cp` och `cat`. Namnen på kommandona är, som så ofta, förkortningar och de här är förkortningar för *copy* respektive *concatenate*. Det ena kommandot kopierar filer och det andra konkatenerar filer. Konkatenering (engelska: *concatenation*) betyder sammanslagning och det innebär att man alltså kan använda kommandot `cat` för att slå samman flera filer till en. Vi ska se hur de används men vi ska vända på steken här och först läsa manualsidor till kommandona:

```
NAME
      cp - copy files and directories

SYNOPSIS
      cp [OPTION]... [-T] SOURCE DEST
      cp [OPTION]... SOURCE... DIRECTORY
      cp [OPTION]... -t DIRECTORY SOURCE...

DESCRIPTION
      Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.
```

Vi har en en lite mer komplicerad manualtext här som vi ska studera lite närmare. Texten "`cp - copy files and directories`" är förstås inte svår att förstå, det anger i princip bara att detta är ett kommando som man använder för att kopiera filer och kataloger. Det är i nästa sektion som det blir lite mer att läsa. Vi ser tre rader, varje rad anger ett funktionssätt hos kommandot och manualsidan kan således dokumentera flera sätt som ett kommando kan fungera på. Vi ska bara studera de första två, ni som är intresserade kan läsa i manualsidan till `cp` för att förstå det tredje sättet att använda `cp`.

```
cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
```

För enkelhetens skull bortser vi också från allting inom hakparenteser, det kan man göra när man läser manualsidor, för vid en enkel användning av kommandot så behöver man ofta inte använda det som står inom hakparenteser. Vi har då två olika sätt att anropa `cp` som vi ska studera:

```
cp SOURCE DEST
cp SOURCE... DIRECTORY
```

och nu blir manualtexten mycket lättare att läsa. Det stod "`Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.`" vilket här alltså betyder att kommandot kopierar det som anges i `SOURCE` till det som anges i `DEST`. Texten är en förkortning för destination, dvs mål. På svenska skulle vi kunna översätta det så här: `cp KÄLLA MÅL`. Det som finns i källa kopieras alltså över till det som anges under MÅL. Ett första enkelt exempel på användningen illustreras då av följande sekvens av kommandon:

```
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.38 fil4
$ cp fil4 kopia_av_fil4
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 13.38 fil4
-rw-r--r-- 1 me me 0 15 apr 16.16 kopia_av_fil4
```

Vi ser här en kopiering av `fil4`, som har längden noll, över till en annan fil som, fantasilöst nog kallas `kopia_av_fil4`. Den får också längden noll eftersom `fil4` var tom. Med növändighet kommer också kopia att få en senare time stamp än originalet som vi också ser här.

Detta illustrerar det första sättet som `cp` kan fungera på, en vanligt kopiering av en fil till en annan. Det andra sättet, som beskrevs av raden "`cp SOURCE... DIRECTORY`" kan illustreras av följande följd av kommandon:

```
$ mkdir nykatalog
$ cp fil4 kopia_av_fil4 nykatalog
$ cd nykatalog/
$ ls -l
totalt 0
-rw-r--r-- 1 me me 0 15 apr 16.20 fil4
-rw-r--r-- 1 me me 0 15 apr 16.20 kopia_av_fil4
```

Vi fortsätter då i samma katalog som ovan, men vi skapar en ny katalog, som heter `nykatalog/` och som genom det andra sättet att fungera hos `cp` får båda filerna nedkopierade i sig.

Manualsidan till `cat` lyder så här:

```
NAME
      cat - concatenate files and print on the standard output

SYNOPSIS
      cat [OPTION]... [FILE]...

DESCRIPTION
      Concatenate FILE(s), or standard input, to standard output.
```

För att förstå det här behöver vi först införa ett speciellt *UNIX*-begrepp som heter *ström*. Varje kommando i *UNIX* har tre strömmar, en inmatningsström, som kallas *stdin*, förkortning för *standard in*, en utmatningsström, som kallas *stdout*, förkortning för *standard out* och en felström som vi inte berör här. När vi använder en kommandotolk så är normalt *stdout* kopplad till skärmen, om vi läser manualsidan till `echo` så ser vi följande:

```
NAME
      echo - display a line of text

SYNOPSIS
      echo [SHORT-OPTION]... [STRING]...
      echo LONG-OPTION

DESCRIPTION
      Echo the STRING(s) to standard output.
```

Som alltså beskriver att echo skriver ut, på stdout, den text som man anger i `STRING`. Om vi alltså skriver `echo “Hej”` så skrivs `Hej` ut. Men man kan faktiskt styra om stdout så att resultatet hamnar i en fil istället. Om vi skriver så här:

```
echo “Denna text hamnar i en fil” > fil1
```

så skrivs inte texten ut på skärmen. Det som händer istället är att det skapas en ny `fil1`, om den inte redan fanns, som innehåller den text som angetts (`Denna text hamnar i en fil`). Om vi gör detta kommando följt av `ls -l` så ser vi följande resultat:

```
$ echo "Denna text hamnar i en fil" > fil1
$ ls -l
totalt 4
-rw-r--r-- 1 me me 27 15 apr 16.38 fil1
```

Vi ser alltså att det skapats en ny fil. Rent tekniskt säger vi att strömmen *stdout* från `echo`-kommandot dirigerades om till filen `fil1`. Vi ser att storleken på fil1 är 27 och det stämmer om vi räknar antalet tecken i strängen `“Denna text hamnar i en fil”` det är 26 tecken. Varifrån kommer det 27:e då? (Ledning: Läs manualsidan till `echo`, fundera över vad `echo -n` betyder.)

Med denna diskussion i bagaget kan vi lättare förklara vad kommandot `cat` gör. Manualtexten lyder: “`cat - concatenate files and print on the standard output`”. Vi kan alltså förvänta oss att cat tar ett antal filer, slår samman dem och skriver ut resultatet på *stdout*, dvs skärmen om vi kör i en kommandotolk. Vi ser på ett par programexempel som illustrerar denna

funktionalitet, vi väver här också in anrop till `cp` och studerar löpande resultatet av varje kommando med `ls -l`. Vi börjar med en tom katalog igen:

```
$ echo "potatis morot sallad" > lista1
$ echo "korv juice frukt" > lista2
$ ls -l
totalt 8
-rw-r--r-- 1 me me 21 15 apr 16.48 lista1
-rw-r--r-- 1 me me 17 15 apr 16.49 lista2
$ cat lista1
potatis morot sallad
$ cat lista2
korv juice frukt
$ cat lista1 lista2
potatis morot sallad
korv juice frukt
$ cat lista1 lista2 > lista3
$ ls -l
totalt 12
-rw-r--r-- 1 me me 21 15 apr 16.48 lista1
-rw-r--r-- 1 me me 17 15 apr 16.49 lista2
-rw-r--r-- 1 me me 38 15 apr 17.53 lista3
$ cat lista3
potatis morot sallad
korv juice frukt
```

Först studerar vi hur kommandot `cat` kan användas för att mata ut innehållet i enskilda filer, kommandona `cat lista1` och `cat lista2` illustrerar detta. Därefter ser vi hur `cat`-kommandot kan användas för att slå ihop innehållet i två filer. Utmatningen från kommandot `cat lista1 lista2` ger oss innehållen i båda filerna `lista1` och `lista2` i en lång sammanhållen listning. Nästa steg är att lägga denna långa listning i en tredje fil som vi kallar `lista3`. Kommandot `cat lista1 lista2 > lista3` gör detta och det ser nästan likadant ut som kommandot `cat lista1 lista2`, skillnaden är att vi adderar “`> lista3`”. Vi utläser kommandot “`cat lista1 lista2 > lista3`” så här: “Gör först en sammanslagning av innehållet i filerna `lista1` och `lista2`. Skriv inte ut detta på *stdout*, utan dirigera om den utmatningen så att den går till filen `lista3` istället”. När vi sedan gör `ls -l` så ser vi att storleken på innehållet i filen `lista3` är 38 som är summan av 21 och 17 som är storlekarna på filerna `lista1` och `lista2`. Det är ju naturligt eftersom innehållet i `lista3` är precis innehållen i `lista1` respektive `lista2`. Vi får ytterligare en bekräftelse på detta genom att studera innehållet i `lista3` med `cat` som är det sista kommandot ovan.

Mer om omdirigeringar

Kommandon i *UNIX*-miljön har alltså möjligheter att ställa om sina in- och utmatningar. Det kan bli ännu tydligare om vi studerar `cat`-kommandot som alltså slår ihop det den får och lägger i en ny fil. Ovan har vi sett hur vi kan ställa om utmatningen från det kommandot, men nu ska vi se hur vi också kan ställa om inmatningen. Om vi ger kommandot `cat > resultat`, det vill säga vi anger en omdirigering med pilen direkt, utan att ange någon fil som vi ska läsa från, ja, då läser

`cat` från *stdin* och när vi kör `cat` från en kommandotolk så blir *stdin* själv tangentbordet som vi skriver med, det innebär att `cat` läser från tangentbordet och skickar resultatet till den fil vi angett som ovan heter resultat. Vi ser på en provkörning:

```
$ cat > resultat
rad 1
rad 2
kalle
olle
lisa
$ ls -l
totalt 24
drwxr-xr-x 2 me me 4096 11 apr 17.22 bin
drwxr-xr-x 2 me me 4096 15 apr 13.25 Desktop
drwxr-xr-x 2 me me 4096 21 jan  2010 GvR
drwxr-xr-x 9 me me 4096 11 apr 17.18 GvRng_4.4
-rw-r--r-- 1 me me   28 18 apr 11.05 resultat
drwxr-xr-x 2 me me 4096 18 apr 11.04 tom
$
```

Kommandot `cat` läser till den hittar ett filslut, vi måste alltså ange filslut med tangentbordet, det är ctrl-D, så efter texten "`lisa`" ovan trycktes ctrl-D. Då kom prompten tillbaka och vi gav då kommandot `ls -l` och ser att det skapats en ny fil som heter resultat och dess storlek är 28 tecken som är precis inmatningen av

```
rad 1
rad 2
kalle
olle
lisa
```

Vi kan faktiskt ovan också välja att inte ange någon utmatningsfil, vi kan bara skriva kommandot "`cat`" utan några andra argument. Det ska då utläsas som att tag allting som kommer på *stdin* (tangentbordet) och skicka ut på *stdout* (skärmen). Det blir alltså bara ett kommando som upprepar allt som skrivs på tangentbordet. Vi ser på en provkörning:

```
$ cat
hej
hej
sluta härmas!
sluta härmas!
$
```

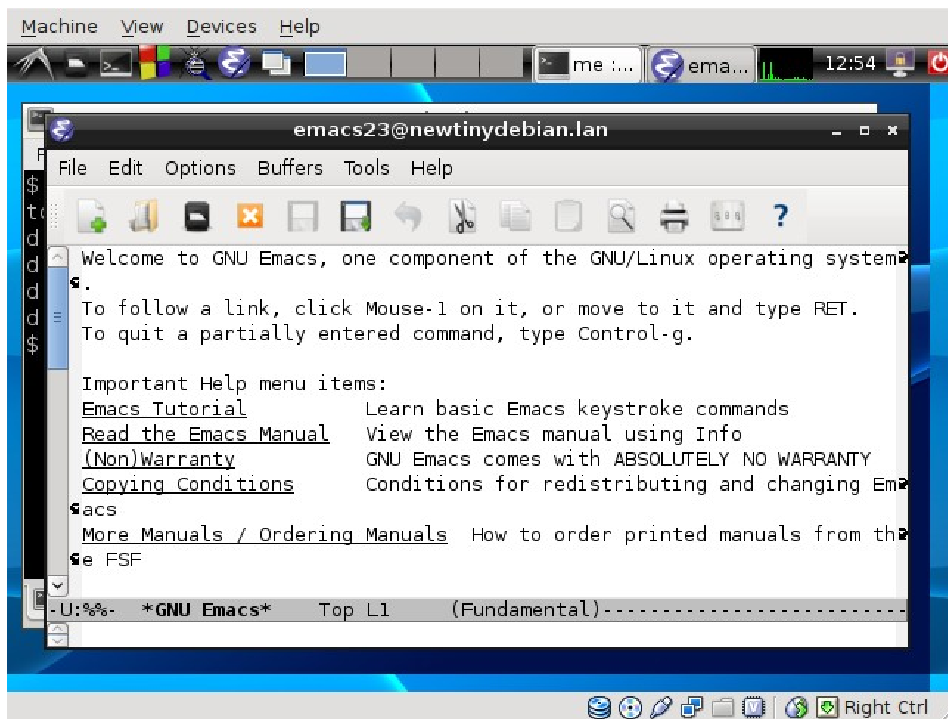
Kommandot `cat` avslutar då den läser ett filslut så vi får stopp på härmapan genom att trycka ctrl-D som förut.

Textredigerare

Vi kan förstås skapa textfiler med `cat` som vi sett ovan, men när vi programmerar behöver vi möjligheter att löpande arbeta med textfiler. Därför använder vi helst så kallade *textredigerare*, eller som det ogentligt ofta kallas “texteditorer”. Ordet “edit” är ju engelska men det har smugit sig in i datorteknikssvenskan och vi kan rent pragmatiskt säga texteditor eller bara editor. Vissa säger till och med att man “editerar text”, men det känns ganska konstigt tycker jag. Nåja, tiderna förändras.

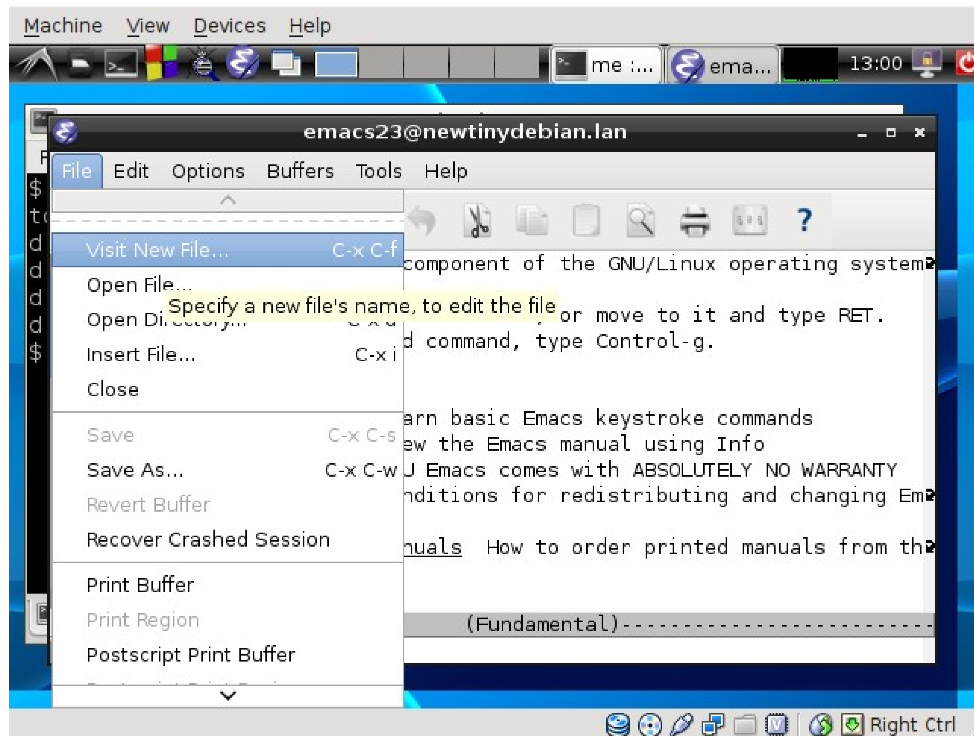
En texteditor är ett speciellt program som är anpassat till att redigera textfiler. En utmärkt texteditor som är speciellt anpassad till att redigera program heter *emacs* och vi rekommenderar att ni använder den genom hela kursen. Den finns förinstallerad i NewTinyDebian och man kan starta den genom en snabbknapp uppe på listen som vi visade förut:

Emacs-knappen är nr 6 och om vi klickar på den får vi följande vy:



Emacs är ett stort program men det enda vi ska använda det till är egentligen bara att skriva och behandla program. Den fjärde knappen, är startknappen till *Code::Blocks* som tycker om att köra. Jag tycker dock bättre om *Emacs* i inlärningsmiljöer eftersom vi lättare, i *Emacs*, kan skilja på så kallade källkodsfiler och exekverbara filer. Exakt vad detta betyder ska vi se på snart. Det är lätt att lära sig hur *Emacs* fungerar, till en början kan man bara använda menyerna och klicka för att aktivera olika kommandon.

När man öppnar filmenyn, genom att klicka på den med musen, ser vi följande:



Nu syns inte muspekaren i denna bild, muspekaren kommer inte med i screenshots, men vi har trots det en pekare som just nu pekar på menyalternativet “Visit New File...”. Det är med hjälp av detta som man skapar nya filer med *Emacs*. Vi kan också se att det står C-x C-f efteråt, det är en angivelse av hur man aktiverar detta kommando med hjälp av tangentbordet. Bokstaven “C” står för ctrl, alltså *control*, och om man alltså trycker ctrl-x ctrl-f aktiverar vi nyfil-funktionen i *Emacs*. Vi ser också att spara-funktioner syns längre ner i menyn, “spara” (*Save*) är ctrl-x ctrl-s, “spara som” (*Save As*) är ctrl-x ctrl-w.

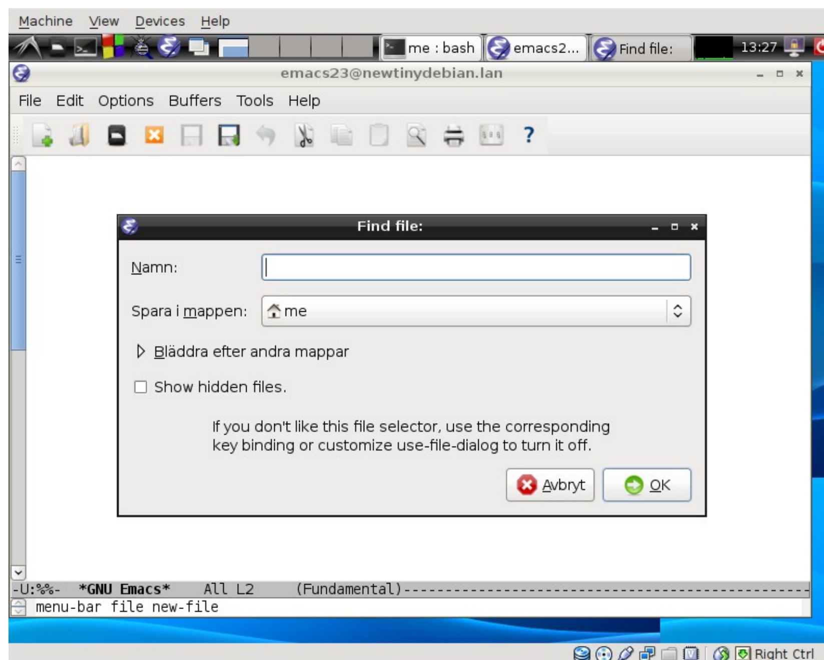
Vi kan gå igenom menyerna och bekanta oss med *Emacs*. Under menyn *Help* finns faktiskt en hel Emacs-tutorial som hjälper oss på ett strukturerat sätt. Vi ska inte gå igenom dessa detaljer här, det vore slöseri med tid, det är sådana här saker som lämpar sig ypperligt för självstudier. Vi ska bara studera en serie manövrar som leder till att man sparar en fil i en förutbestämd katalog.

Exempel: Spara en fil i en förutbestämd katalog med Emacs.

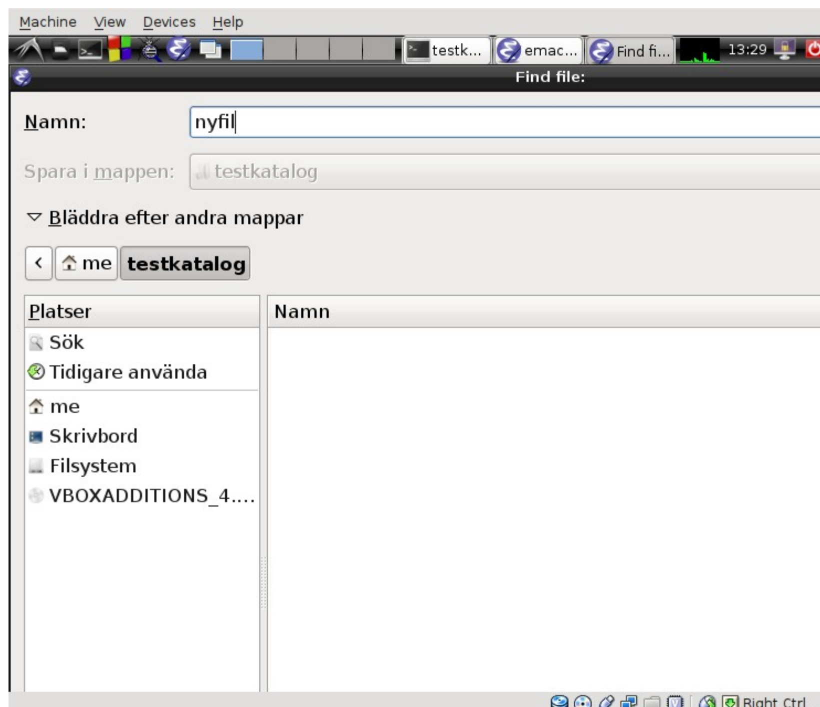
Vi börjar med att skapa den katalog där vi vill spara en fil. Vi ger kommandot `mkdir testkatalog` vid en prompt och har följande vy då det är klart:

```
$ mkdir testkatalog
$ ls -l
totalt 20
drwxr-xr-x 2 me me 4096 11 apr 17.22 bin
drwxr-xr-x 2 me me 4096 15 apr 13.25 Desktop
drwxr-xr-x 2 me me 4096 21 jan  2010 GvR
drwxr-xr-x 9 me me 4096 11 apr 17.18 GvRng_4.4
drwxr-xr-x 2 me me 4096 18 apr 13.12 testkatalog
$
```

Vi lägger alltså katalogen under `/home/me` som syns ovan. Nu startar vi *Emacs* och väljer “Visit New File...” genom att klicka på File-meny. Vi får då följande vy:

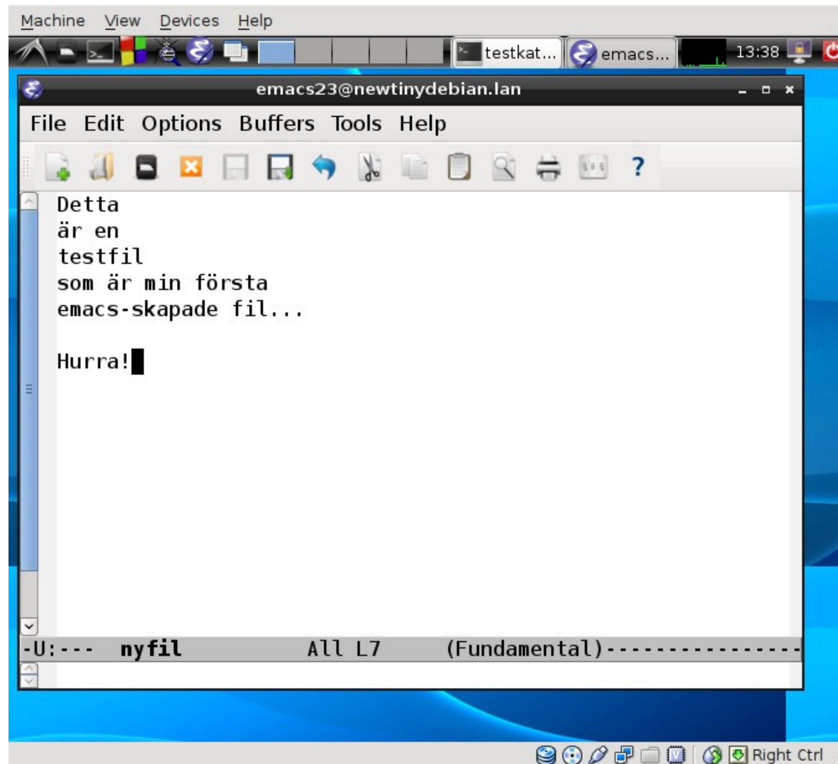


Vi klickar då på Bläddra efter andra mappar och får då upp en filväljare där vi kan vandra genom filhierarkin likt *Windows Explorer* eller *File Manager* som vi sett tidigare. Där väljer vi då katalogen `testkatalog`, som vi skapade nyss med `mkdir` och där kan vi ange filnamnet, som vi kallar `nyfil`. Vi tar en snabb titt på hur det ser ut:

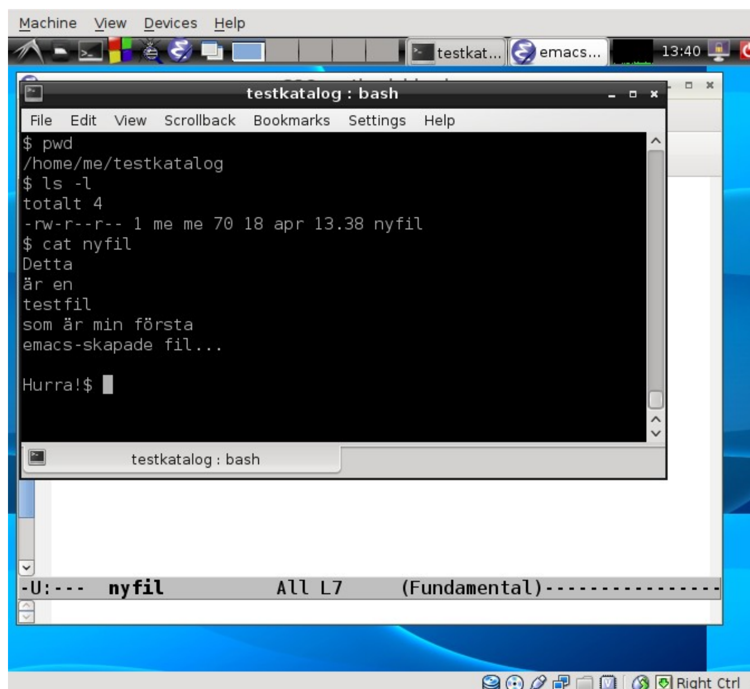


När dessa val är gjorda, när vi skrivit in namnet (`nyfil`) och tryckt på return får vi upp en vanlig

fileditorvy som alltså betyder att vi nu är redo att börja arbeta med filen. Vi kan skriva in ett innehåll och spara med ctrl-x ctrl-s. Vi gör detta. Det ser ut så här:



Om vi nu går till vår kommandotolk (där vi alltså skriver kommandon, typ `ls`, `mkdir`, `cat` etc.) och granskar innehållet i filen `/home/me/testkatalog/nyfil` kan det se ut så här:



Vi granskar situationen med kommandona `pwd`, `ls -l` och `cat` och ser att den är som vi kan förvänta oss. Filen `nyfil` har skapats där vi angav att den skulle skapas och den har det innehåll som vi just matat in. (`Detta är en emacsfil...`)

Nedan ges en sammanställning av emacs-genvägar som kan vara bra att komma ihåg:

Översikt av Emacs-kommandon

(C betyder Ctrl-tangenten)
(M betyder Alt-tangenten)

Övergripande: C-g: Avbryta ett pågående kommando C-z: Hoppa ut ur emacs utan att stoppa C-x C-c: Avsluta	Fönster/buffrar: C-x 1: stäng alla andra fönster "(1) window" C-x o: Hoppa till annat fönster "(o)ther" C-x 2: Öppna nytt fönster "(2)nd window" C-x b: Hoppa till en annan (b)uffer
Flytta markör: C-b: föregående tecken "(b)ack" C-f: nästa tecken "(forward)" C-p: föregående rad "(previous)" C-n: nästa rad "(next)" M-f: Framåt ett ord "(forward)" M-b: Bakåt ett ord "(back)"	Söka: C-r: söker bakåt (inkrementellt) C-s: Söker framåt (också inkrementellt) Kompilera med make-fil: M-x compile
Filer: C-x C-f: Öppna en fil "(f)ind" C-x C-s: Spara filen	Kommandoskal (testa program bla.): M-x shell (Kan också göra make clean.)

Detta dokument finns också på kursens webbsida.

Det kan finnas fler kommandon. Det är bra att gå igenom någon enkel tutorial på nätet om Emacs, en av de bästa finns i första kapitlet av *Advanced Linux Programming*, på www.advancedlinuxprogramming.com. Kapitlet finns under *Downloads*. De som går data-utbildningen kommer också att använda den här boken senare i kursen.

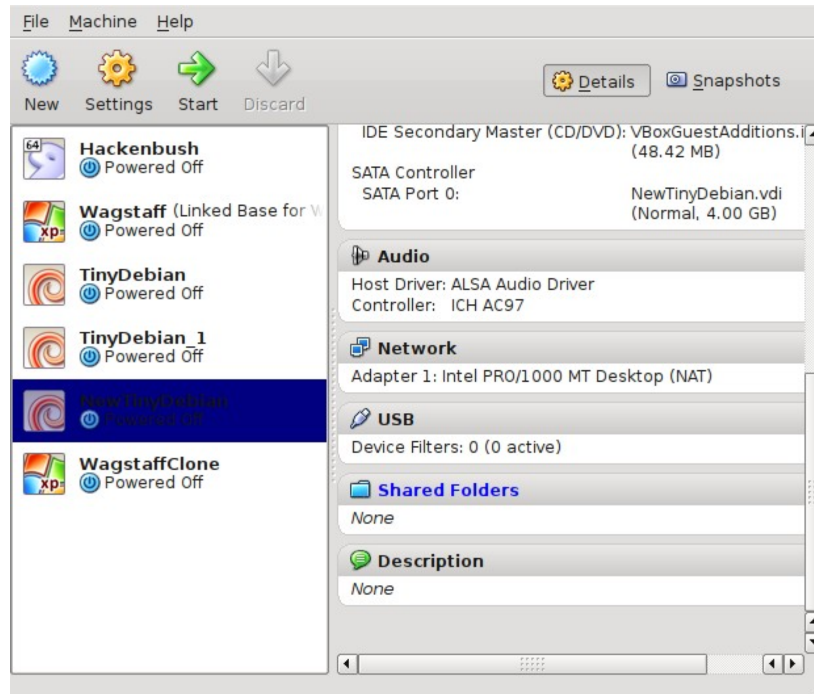
Delade kataloger

Vi kör ju *virtuellt*, som det heter, vår maskin, som heter *NewTinyDebian*, är egentligen inte en fysisk maskin, den består av en samling filer på värddatorn. Det fantastiska program som möjliggöra detta kallas för en virtualiseringsprogramvara och ett exempel är *VirtualBox*.

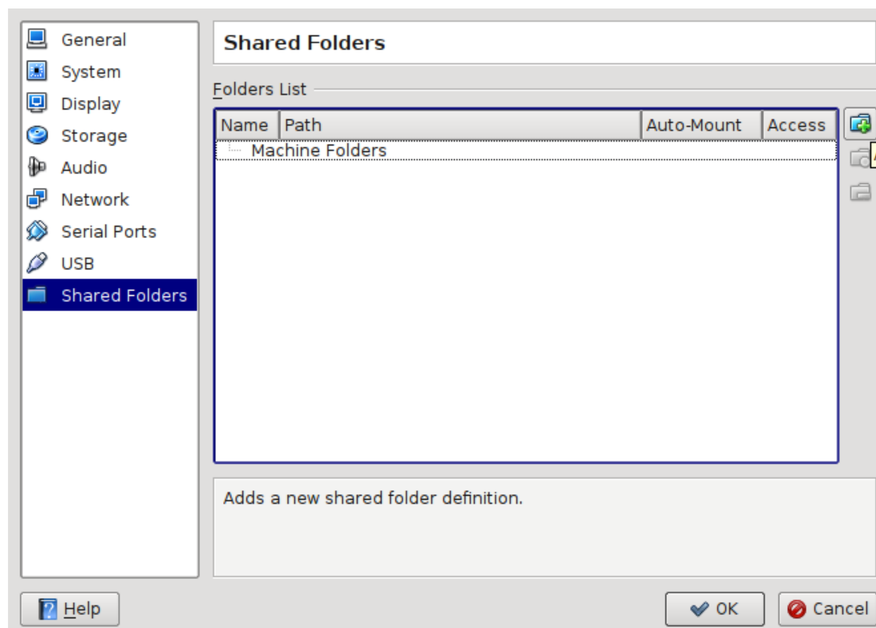
Vi har alltså upplevelsen av att ha två datorer, *NewTinyDebian* och vår värdmaskin. För att kunna arbeta med filer på ett konkret sätt som vi är vana vid, till exempel skriva ut på en skrivare, eller liknande, vill vi spara de dokument vi arbetar med inte på *NewTinyDebian* utan faktiskt på värdmaskinen. Det är också mycket meningsfullt att spara filer vi arbetar med utanför *NewTinyDebian* av en mycket speciell anledning: *Vi vill separera verktyg och dokument*. Om vi alltså sparar de filer vi arbetar med på värdmaskinen istället för på *NewTinyDebian* så åstadkommer vi just detta. Dokument på ett ställe och verktyg på ett annat ställe. Vi ska uppnå detta genom att arbeta med så kallade *delade kataloger*.

Att skapa en delad katalog

För att se till att en katalog blir delad måste vi först stoppa *NewTinyDebian*. Vi går sedan till kontrollpanelen för alla virtuella maskiner och letar upp *Shared Folders*. Det ser ut så här:

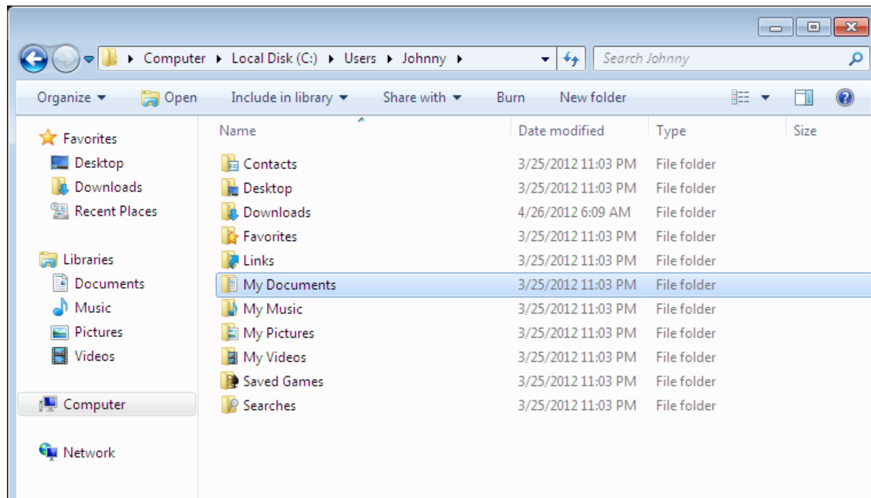


Vi klickar på *Shared Folders* och får upp följande vy:



Vi klickar då här på knappen *Add Shared Folder* (som syns till höger). Då får vi upp en ruta där vi får välja “Folder Path” och “Folder Name”. I rutan för “Folder Path” väljer vi en katalog på värdmaskinen, om vi till exempel har en *Windows*-maskin kan det vara bra att välja den katalog som innehåller alla filer som hör till den användare som kör *VirtualBox*. Katalogstrukturen i *Windows*

utgår ofta från någonting som kallas C:\ och det motsvarar *UNIX*-miljöns rotkatalog. Då det gäller att välja vilken katalog som ska vara delad mellan värddatorn och *NewTinyDebian* rekommenderar jag att ni tar C:\Users\<ert användarnamn>. Här ska "<ert användarnamn>" bytas mot det användarnamn som ni är inloggade som. Om vi tittar på en körning från en *Windows*-dator som jag (tyvärr) har på mitt rum ser det ut så här:



Om jag hade kört *Windows* som operativsystem på min värddator (vilket jag tack och lov inte gör) skulle jag alltså valt att dela katalogen C:\Users\Johnny.

Då *Folder Path* är valt ska man välja ett namn på katalogen som kommer att presenteras för *NewTinyDebian*. Ett namn väljs automatiskt här och ni kan bara låta det valet stå kvar om ni vill. Det är inte så viktigt vad den heter, det är dock MYCKET VIKTIGT att ni väljer **automount**. Det är en kryssruta som man får kryssa i. Välj dock inte read-only.

Nu när allt är klart kan ni starta om *NewTinyDebian* och starta en kommandotolk. Gör sedan cd /media följt av ls -l. Ni bör se någonting i följande stil då:

```
me@newtinydebian:/media$ ls -l
totalt 14
lrwxrwxrwx 1 root root       6 28 mar 23.13 cdrom -> cdrom0
dr-xr-xr-x 4 root root    2048  3 apr 14.49 cdrom0
drwxrwx--- 1 root vboxsf 12288 18 apr 18.46 sf_johnny
me@newtinydebian:/media$
```

Ni kommer troligtvis att ha en katalog som inte heter sf_johnny men något annat, om er användare i i er värddatorn heter kalle kommer katalogen att heta sf_kalle. Detta är då också en delad katalog mellan virtuell maskin och värdmaskin. Katalogen finns på värdmaskinen men är åtkomlig från den virtuella maskinen.

Shared Folders är del av någonting som heter *VirtualBox Guest Additions*. De ger också möjlighet till någonting som heter seamless mode som gör att man kan använda värdmaskinens skrivbord också till den virtuella maskinens. Det ger också maskinerna ett gemensamt clipboard så allt som man kopierar från dokument i en maskin kan klistras in i dokument i den andra maskinen.

Kompilerade och interpreterade språk

Som programmerare måste vi vara medvetna om att det finns två olika sorters programmeringsspråk – de kompilerade språken och de interpreterade språken. Ni kommer att använda väldigt mycket av båda sorter i er utbildning. I grundkursen i programmering kommer vi att studera programmmeringsspråket *C* som är ett kompilerat språk.

För att förstå skillnaden mellan kompilerat och interpreterat språk måste vi gå till grunden beträffande vad det innebär att programmera. Att programmera är att rada upp instruktioner som en maskin (ofta en dator) ska utföra. I datorn finns en central beräkningsenhet (Central Processing Unit, CPU) och den läser och utför hela tiden instruktioner från datorns primärminne. I primärminnet finns det alltså lagrat instruktioner om vad som ska ske. CPU läser flera miljarder gånger per sekund ur primärminnet och instruktionerna är mycket enkla och anpassade till vad CPUn kan göra. CPUn kan inte göra komplicerade saker alls. Men en dator kan fås att göra komplicerade saker genom att ohyggligt många enkla instruktioner kombineras ihop till större program. Det är själv aktiviteten att skapa dessa program som kallas att programmera. Men vi kan inte programmera i CPUns maskinspråk direkt. Vi måste skriva våra program i ett annat språk som människan kan förstå. Dessa språk kallas programmeringsspråk. Ett program kan också kallas för *kod* eller *programkod* och ett stycke text som består av kod som utgör ett program kallas ibland för källkod. En textfil som innehåller en källkod skrivet i ett programmeringsspråk kallas en källkodsfil. CPUn kan som sagt inte förstå detta och därmed kan inte programmet direkt köras utifrån källkoden. Det som behövs för att man ska kunna köra ett program är bland annat någonting som kallas för en kompilator, en översättare som översätter källkoden till körbar maskinkod. (Man behöver något som heter länkar och laddare också, men det bortser vi från i den här diskussionen.) För när ett program är kompilerat, alltså översatt till maskinkod, går det att köra programmet. Observera att då har en *ny* fil uppstått, källkoden innehöll programmet skrivet i ett programmeringsspråk som en människa kan förstå, men maskinkoden måste befinna sig i en speciell körbar maskinkodsfil. Denna fil kan inte människor förstå. Det är mycket stor skillnad mellan källkod och maskinkod.

Själv handlingen att översätta ett program kallas att *kompilera* det. Att köra programmet kallas ibland att *exekvera* det.

De programmeringsspråk där källkoden översätts till en separat körbar fil kallas kompilerande språk. Här har vi *C*, *C++*, *Pascal*, *Ada*, etc. Men det finns en möjlighet till och det är att man kan låta en översättare översätta programmet samtidigt som man kör det. Det kallas då att man *interpreterar* koden, tolkar koden. De språk som fungerar så här kallas då för interpreterade språk. Språken kan också kallas scriptspråk. Här har vi *Python*, *Perl*, *Bash* etc. *Java* är ett mellanting som ni gärna får läsa själva om. *Java* kommer definitivt att ingå senare i er utbildning.

`gcc` och `python`

Det program som gör översättningen kallas som sagt kompilator och vår *C*-kompilator heter `gcc`, det är en förkortning för *GNU's Compiler Collection* och den kan kompilera flera olika språk. *C* är dess förstahandsval, man kan har flera programmeringsspråk installerade men vi ska bara studera *C*.

Ett exempel på ett interpreterande språk som är mycket viktigt idag heter *Python*. Här finns alltså ingen kompilator utan man anger direkt en källkodsfil som `python` kör. Vi kommer att stöta på det som hastigast i och med att vi börjar med *Guido van Robot*.

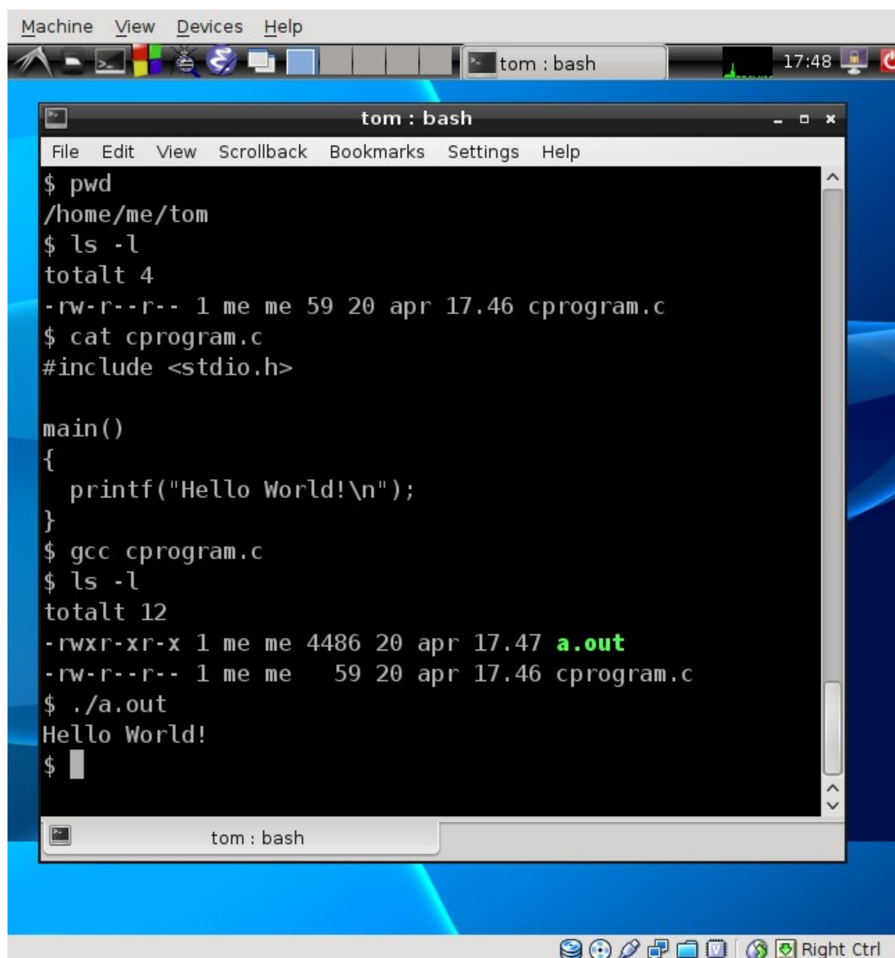
Ett enkelt *C*-program

Nedanstående är ett enkelt C-program som alla måste kunna producera och kompilera och exekvera.

```
#include <stdio.h>

main()
{
  printf("Hello World!\n");
}
```

Vad ni behöver göra med detta är att ta det och skriva in det i en fil som heter `cprogram.c`. Det är filen `cprogram.c` som kallas källkodsfilen eller *C*-programmet. Vi kan göra följande tester för att se att ni förstått proceduren. Vi arbetar återigen i en tom katalog som heter `/home/me/tom`. Efter att vi skapat filen `cprogram.c` med *Emacs* så har vi följande vy:



Vi ser här att vi alltså har två filer, `cprogram.c` är källkodsfilen och `a.out` är det körbara programmet. Vi måste skriva `./a.out` för att kunna köra igång programmet som alltså bara skriver ut texten `Hello World!` på skärmen. Med hjälp av kommandot `ls -l`, observerar vi också att den körbara filen (`a.out`) inte finns innan vi kompilerat käöökodsfilen (`cprogram.c`) med `gcc`. Det är mycket viktigt att ni genomför ovanstående steg så att ni säkert är igång och har en fungerande programmeringsmiljö. Det kommer att vara en av de saker vi examinerar i denna kurs. I Ovningar.pdf gås det igenom vad som krävs för godkänt på denna kurs.