Lektionstillfälle 6

Programexekvering

med Mikael Larsson

NACKADEMIN

Återblick

Förra gången jobbade vi med användare och rättigheter.

- Vi gick igenom olika slags användare, hur användare skapas och ändras.
- Vi pratade om lösenord, hashning, salt och peppar.
- Vi började också titta på processer och processträd.

Dagens lektion

Mål: Att veta hur program startas, pausas och avbryts

- PATH
- Argument
- Inmatning, utmatning och omdirigering
- Förgrunds- och bakgrundsprocesser
- Jobbkontroll
- Exit codes
- Signaler*

TLCL kapitel 6, 7, 10

**överkurs*

Termer och begrepp

- Bakgrundsprocesser
- Signaler
- stdin, stdout och stderr
- Omdirigering, redirection
- Att ”pajpa” (pipea?) till, ”pajpa” från, etc.
- ”&”, ”&&”, ”||”, ”|”, ”<”, ”>”, ”>>”

Hur startas en process?

- Systemets första process init/systemd startas av kärnan och får PID 1. init startar också kthreadd.
- Därefter kan processer skapa nya processer genom systemanropet **fork()**, som skapar en klon av nuvarande process.
- När en process är ”forkad” kan den välja att ladda en ny binär med systemanropet **exec()**.
- exec() **ersätter** nuvarande process’ binär med den nya. Funktionen **system()** gör fork() och exec().

Så – Linux har separerat ”skapa ny process” från ”ladda in och kör binär”.

? **När kan man vilja att ”bara” göra en fork, alltså skapa ny process som är en exakt kopia?**

OK, men hur startar jag en process?

- Nya processer skapas av skalet när exekverbara filer körs.
- När exempelvis **ls** körs startas en process med binären ”/bin/ls”.
- För att kunna skriva bara **ls** istället för den absoluta sökvägen ”/bin/ls” måste platsen där **ls** bor finnas i miljövariabeln PATH.

Använd **printenv PATH** för att se nuvarande PATH.
När en användare loggar in tas PATH från **/etc/environment.** Egna tillägg kan göras i ~/.profile

? OBS att ”.” inte är med i PATH.
Vad får det för konsekvens?

? Finns det något annat sätt att starta en exekverbar fil utan att addera sökvägen till PATH?

Shell och subshell

- Många gånger skapar skalet också nya skal (subshell) av sig självt för att lösa vissa uppgifter.
- Det är inte alltid tydligt att det händer.

Exempel:

```
ps -ef | grep $USER | tail -n 4
```

vs

```
(ps -ef | grep $USER) | tail -n 5
```

Förgrunds- och bakgrundsprocesser

Varje terminal kan ha som mest en förgrundsprocess igång, men många bakgrundsprocesser.

För att starta ett program i bakgrunden lägger man till En ampersand/och-tecken (&) efter kommandot:

```
$ free –s 10 &
[1] 1553
```

För att plocka fram programmet i förgrunden igen använder man ”fg”:

```
$ fg %1
free –s 10
```

Processer och ”jobb”

Vill man skicka ett program till bakgrunden kan man trycka **[ctrl-Z]**, vilket pausar processen, och sen använda ”bg” för att starta processen igen i bakgrunden.

OBS [ctrl-Z] avslutar inte processen!

fg, bg och [ctrl-Z] är jobb-kommandon i skalet.
Ett “jobb” är en process som hanteras interaktivt av skalet.

Jobb är numrerade från 1 och uppåt, och är också knutna till en process med en eget PID.

Använd “jobs” för att lista jobb.

D	uninterruptible sleep (usually IO)
I	Idle kernel thread
R	running or runnable (on run queue)
S	interruptible sleep (waiting for an event to complete)
T	stopped by job control signal
t	stopped by debugger during the tracing
W	paging (not valid since the 2.6.xx kernel)
X	dead (should never be seen)
Z	defunct ("zombie") process, terminated but not reaped by its parent
	----------
<	high-priority
s	is a session leader
l	is multi-threaded
+	is in the foreground

Tips: “fg”, “bg“ och “jobs” är interna skal-kommandon, använd “help” för att få mer info om argument.

Tips för bakgrundsjobb

- Redigera filer och hoppa till manblad eller test
- Vänta på långa jobb
- Flytta en server till bakgrunden

Att avbryta en process

Vi har sett att man kan avbryta en process med [ctrl+c].
Men – det fungerar bara på förgrundsprocesser.

För att avbryta bakgrundsprocesser används “kill”.

För att avbryta en process med PID 1234:

```
kill 1234
```

För att avbryta jobbet med jobb-ID 2:

```
kill %2
```

”kill” och rättigheter

Vem som helst får inte döda en process. Om du inte är “root” kan du bara döda dina egna processer.

```
frasse@frasse:~$ kill 1
-bash: kill: (1) - Operation not permitted
frasse@frasse:~$ _
```

??? Om jag startar en process i bakgrunden och loggar ut, så avslutas ju mitt skal.
Det var skalet som startade processen och var förälder till processen.

Vad händer med barnprocessen, och vem äger en föräldralös process?

Egentligen dödar inte ”kill”…

TLCL s119-

Nu har vi använt ”kill” för att döda processer, och kommandot heter ju som det gör.

Men – det ”kill” gör är att det skickar en **signal** till processen.

Signaler används som ett slags kommunikation mellan processer, och mellan kärnan och processer.

Det finns många signaler, använd **”kill -l”** för att lista.

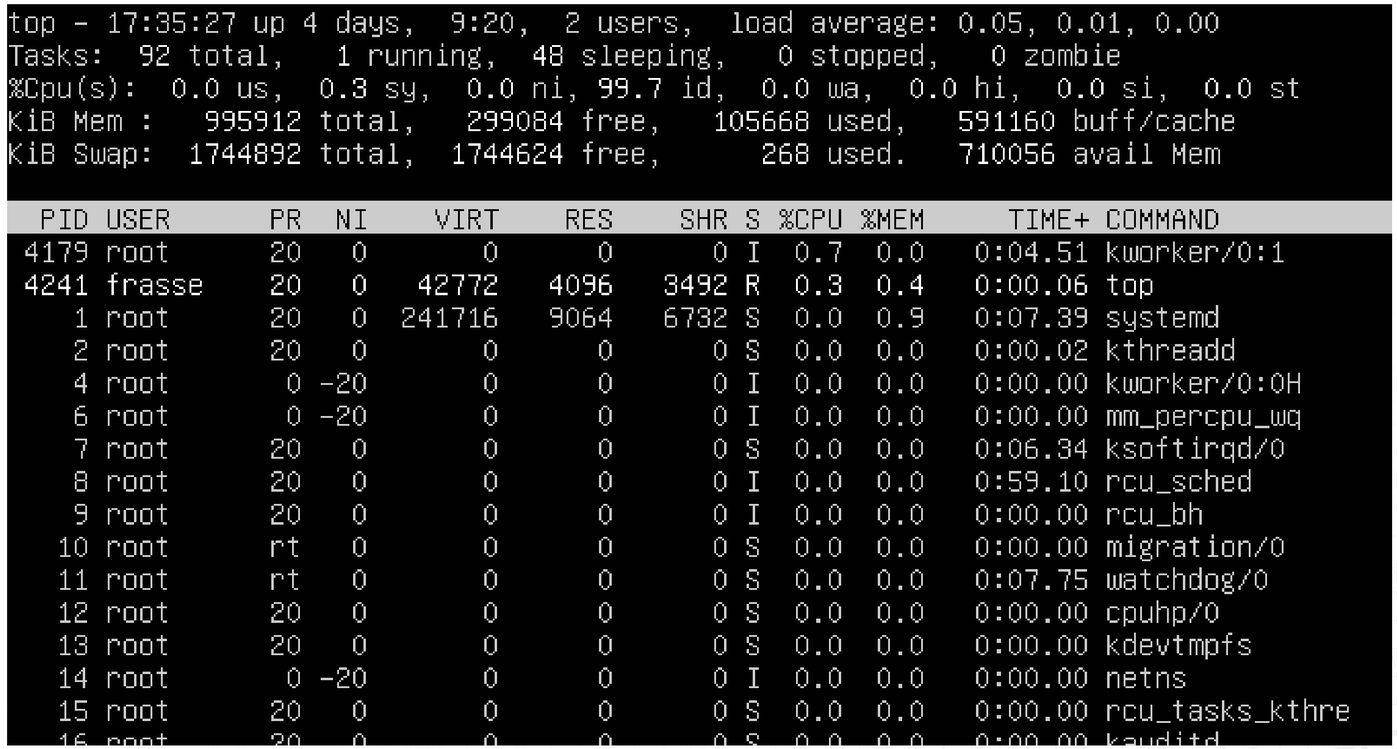
Just nu är det bara SIGTERM (15), som också ”kill” skickar som standard, som spelar roll.

Att trycka på [ctrl+z] i skalet skickar signalen SIGTSTP till processen.

Olika sätt att titta på processer

Vi har använt “ps” och “pstree”.

Ytterligare ett vanligt och mycket användbart kommando är “top”, där man kan få en snabb överblicksbild av läget:



Laboration 1

Se material på studentportalen.

NACKADEMIN

Avslutskoder, exit codes

Alla program avslutas med en kod, som är ett tal mellan 0 och 255.

Om talet är noll, har programmet avslutats utan fel, eller med resultat **true**.

I alla andra fall har något slags fel inträffat, eller så visar programmet en kod som betyder något speciellt:
1 = Operation not permitted
2 = No such file or directory

Avslutskoden från det senast körda kommandot hamnar i variabeln ”$?”, och kan printas med:

```
echo $?
```

Exempel i java:

```
System.exit(42);
```

Exempel i python:

```
sys.exit(42);
```

Exempel i bash:

```
(exit 42); echo $?
(exit -214); echo $?
true ; echo $?
false ; echo $?
```

Att kombinera kommandon

Kommandon kan kombineras, alltså köras efter varandra, beroende på om föregående program lyckas eller inte.

Detta görs med operatorerna && (and) och || (or).

Exempel:

```
mkdir temp && cd temp

grep –q korv 24384.txt && echo “Found!” || echo “Try again.”

true  && echo "That is true!" || echo "That is false!"
false && echo "That is true!" || echo "That is false!"
```

Pajpa och greppa!

TLCL kapitel 6

Många program kan läsa från ***stdin***, (standard input) och skriva till ***stdout*** (standard output) eller ***stderr*** (standard error).

Man kan se alla **std*** som anonyma filer som alltid är kopplade till en process.

```
frasse@frasse:~$ df | grep mapper
/dev/mapper/ubuntu--vg-ubuntu--lv   8703348 3591344   4650180  44% /
```

Kapitel 6 innehåller allt ni behöver veta om omdirigering!

NACKADEMIN

Filnamn som argument eller pipe?

Det kan vara lite ovant först att ett program som ”less” antingen kan läsa från en fil angiven som ett argument, eller från ***stdin***.

Man kan räkna med att alla program som är ”filter” fungerar så.

I exemplet med ”grep”, som använder både argument och filnamn ”vet” programmet att det ska läsa från ***stdin*** när det inte fått ett filargument.

Från man-bladet:

`grep [OPTIONS] PATTERN [FILE…]`

Omdirigera till och från filer

Vi har sett att man kan skicka data mellan kommandon med ”pipe” – tecknet, ”|”.

Man kan också läsa från och skriva till namngivna filer med ”<” respektive ”>”. Man kan se tecknen som ”trattar”.

För att spara utskriften från ett kommando:

```
ps -ef > processes.txt
```

Fyra ekvivalenta kommandon:

```
grep onions 24384.txt | tail > onions.txt
grep onions < 24384.txt | tail > onions.txt
(grep onions | tail) < 24384.txt > onions.txt
cat 24384.txt | grep onions | tail > onions.txt
```

>>

Om man omdirigerar ett kommando till en fil och kommandot inte producerar någon output, så blir resultatet en tom fil.

Med ”>>” istället för ”>” kan man lägga till, ”appenda” till en fil istället.

```
grep onion 24*txt > veggies.txt && \
grep carrot 24*txt >> veggies.txt
```

Omdirigera *stderr*

Hittills har vi omdirigerat ***stdin*** (0), och ***stdout*** (1).
*Siffrorna är deras respektive **fd** - file descriptor.*
Ibland vill man omdirigera ***stderr*** (2), till exempel om man vill samla all output från ett program, oavsett om det är fel eller ”vanlig” output.

```
ls -l /bin/usr /usr/bin > ls-output.txt 2>&1
ls -l /bin/usr /usr/bin &> ls-output.txt    # bara i bash
```

Det är också vanligt att dölja felutskrifter som är förväntade, eller som man inte är intresserad av.

```
ls -l /bin/usr /usr/bin 1> ls-output.txt 2> /dev/null
ls -l /bin/usr /usr/bin > ls-output.txt 2> /dev/null
```

```
grep onions < 24384.txt | tail > onions.txt
grep onions 0< 24384.txt | tail 1> onions.txt
```

Filter

I labben kommer ni använda:

- grep – skriv ut rader som matchar
- sort – sortera rader
- uniq – eliminera / rapportera dubblettrader

Bra varianter:

- grep -i, ignore case
- grep -v, reVerse match
- sort -u, unique
- sort -n, numeric
- sort -r, reverse
- uniq -d, print duplicates
- uniq -c, count – print number of duplicates

Skriv program som gör en sak
och gör det bra.

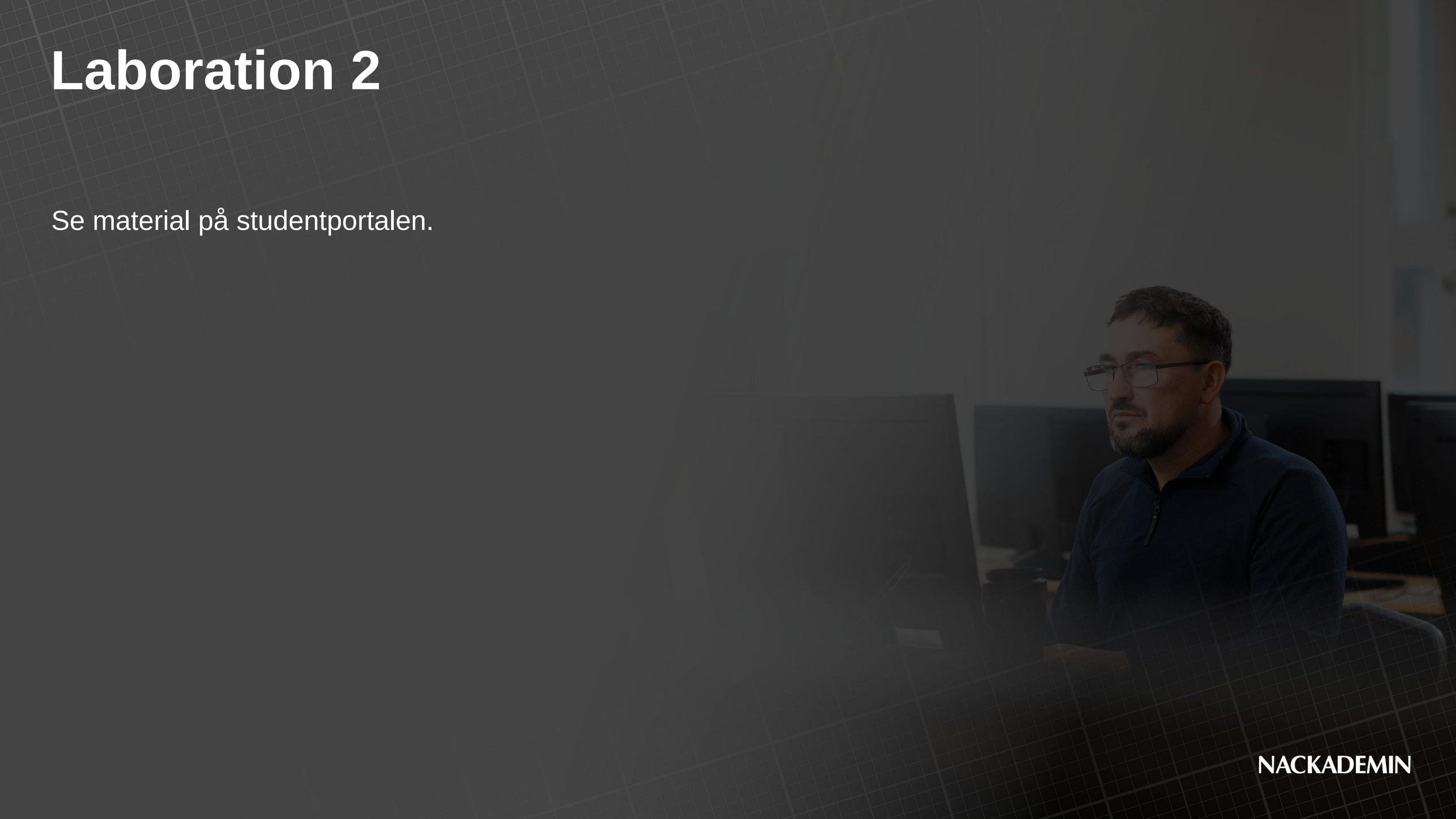
Skriv program som arbetar tillsammans.

Skriv program för att hantera textflöden
eftersom det är ett universellt gränssnitt.

The Unix Principle

Laboration 2

Se material på studentportalen.



NACKADEMIN

Summering

Dagens ämnen har varit:

- PATH
- Förgrunds- och bakgrundsprocesser
- Jobbkontroll
- Exit codes
- Signaler
- Inmatning, utmatning och omdirigering

Läs boken: **TLCL kapitel 6, 7, 10**

Nästa gång

Säker kommunikation

Mål: Att förstå och använda ssh, generera och installera nycklar.

- ssh, scp, sftp
- Serverinställningar
- Klientinställningar
- Nycklar och lösenordslös inloggning
- Tunnlar*

Inlämningsuppgift 1

**överkurs*

Stort tack!

NACKADEMIN