**UMEÅ UNIVERSITET**
**Institutionen för Datavetenskap**
**Laborationsrapport**

**November 5, 2014**

Laboration 4

# Systemnära Programmering 7,5 hp

mish

**Namn** Christer Jakobsson
**E-mail** dv12cjn@cs.umu.se
**Sökväg** ~dv12cjn/edu/sysprog/lab4

**Handledare**
Mikael Rännar, Adam Dahlgren Lindström, Niklas Fries, Oskar Ottander

# Contents

# 1 Problemspecifikation

*Denna laborations syfte har varit att få praktisk erfarenhet av att använda fork och andra systemanrop i linux, samt att använda fil deskriptorer för att omdirigera standard in och standard out*

## 1.1 Problemsammanfattning

Denna laboration har gått ut på att skapa ett minimalt *shell* (kommandoskal). Ett skal är ett program som körs då man öppnar en terminal, det skriver ut en prompt och väntar på kommandon att exekvera.

De funktioner som skalet skall klara av är:

- Exekvering av kommandon och program
- Pipor mellan program
- Omdirigering av standard input och standard output
- Hantera signalen SIGINT. Vid signal, avbryta exekveringen av alla uppstartade program.

## 1.2 Originalspecifikation

Specifikationen i sin helhet finns på
`http://www8.cs.umu.se/kurser/5DV088/HT14/lab/lab4/`

# 2 Åtkomst och användarhandledning

## 2.1 Filer som ingår i lösningen

I katalogen ~`dv12cjn/edu/sysprog/lab4` ...

| | |
|---|---|
| mish.c | Fil som är huvuddelen i programmet. |
| mish.h | Header fil för *mish.c* |
| execute.c | Innehåller två funktioner som ingick i uppgiften att implementera. |
| execute.h | Header fil till *execute.c* |
| parser.c | Given fil för uppgiften, behandlar en kommandorad och skapar structar med information om varje kommando. |
| parser.h | Header fil till *parser.c* |
| sighant.c | Signal hanterare till *mish* |

sighant.h Header fil till *sighant.c*

makefile make fil för att kompilera alla delar till programmet och skapa ett körbart *mish*

### 2.2 Kompilering och körning

För att kompilera så använder man medföljande makefil och kör den genom att stå i samma mapp som filerna finns i och skriva *make*. Detta kommer att kompilera de filer som inte har kompilerats förut och skapa ett exekverbart program som heter *mish*.

För att exekvera programmet skriver man i terminalen *./mish*. Detta kommer att starta programmet och det kommer att skriva ut en prompt med aktuell katalog som programmet befinner sig i följt av *:mish*. Nu så väntar mish på att användaren ska skriva in ett kommando och när användaren trycker Enter så försöker mish att köra kommandot, visar ett felmeddelande om något fel uppstår.

Programmet avslutas när det får signalen *EOF* (End-Of-File), vilket kan göras genom att trycka CTRL + D.

## 3 Systembeskrivning

*I detta avsnitt beskrivs systemet i sin helhet mer i detalj. Datastrukturer och annan intern representation som är central för uppgiftens lösning behandlas. Dessutom beskrivs de olika komponenternas relationer till varandra.*

### 3.1 Systemöversikt

#### 3.1.1 mish

*mish* är huvuddelen i programmet, börjar med att skriva ut en prompt som en vanlig terminal och väntar på kommandon, Detta är i en while loop som kommmer att iterera ända tills användaren ger programmet EOF varpå programmet avslutas. Användaren kan här skriva ett kommando och när Enter trycks så skickas den rad som användaren skrivit in till den givna parsern som tolkar raden och bygger upp en array innehållande structs för varje kommando. Programmet försöker sedan att exekvera de kommandon den har fått, om någon av kommandona inte var ett riktigt kommando eller om kommandot får felaktiga parametrar så kommer det inte kunna exekveras och ett felmeddelande skrivs ut och prompten skrivt ut igen. Om det var en giltig kommandorad så kommer parsern att dela upp varje kommando och skapa en struct för varje kommando som ska exekveras och skapa en array av dessa structar. Structen innehåller information om kommandot, om det har en in eller utfil samt vilka parametrar som kommandot ska exekveras med.

Med denna array av structar så itererar jag igenom alla kommandon och kollar om kommandot som skall köras antingen är *echo* eller *cd*. Om så är fallet så sätts den aktuella structens variabel *internal* till ett värde som sedan kontrolleras innan kommando exekvering och kör det interna kommandot om det matchade. Sedan så loopas arrayen av kommandon igenom och för varje kommando så skapas en kopia av processen som körs och dess standard input och/eller standard output omdirigeras för att skapa pipor emellan alla kommandon som ska exekveras. Kontrollerar även om standard in/out ska omdirigeras ifrån eller till en fil för kommandot.

### 3.1.2 sighant

Denna fil är en signal hanterare vars uppgift är att döda alla barnprocesser till *mish* om användaren skickar SIGINT. Detta görs genom att *mish* skickar SIGINT till barnen och för att försöka avbryta dem. Innehåller två globala variabler som håller reda på varje barns process id, en array som lagrar alla pids och en int som är antalet barnprocesser. Dessa globala variabler används sedan i mish filen som stoppar in pids i arrayen och ändrar heltalet som innehåller hur många barnprocesser så det är korrekt antal.

### 3.1.3 execute

Denna fil implementerar de två funktioner vars huvuden angivits i *execute.h*. En funktion som duplicerar en pipa till standard in/out och en som omdirigerar standard in/out till eller ifrån en fil.

### 3.1.4 Givna filer

*För att förenkla laborationen så är en del filer givna som ska hjälpa mig att bygga skalet.*

parser Denna fil tar kommandoraden som användaren skrivit in, behandlar den och bygger upp en array av Command structar. Varje struct i arrayen innehåller information om vilket kommando som ska köras, med vilka argument och om det datat ska hämtas eller skrivas från en fil.

execute.h Denna fil är till för att jag ska implementera två funktioner. En funktion som ska duplicera en pipa till en standard I/O fil deskriptor samt en funktion som ska omdirigera läsning/skrivning från eller till en fil.

## 3.2 Anropningsdiagram

*Figure 1 visar ett diagram över hur funktioner i programmet anropar varandra och vad de returnerar.*

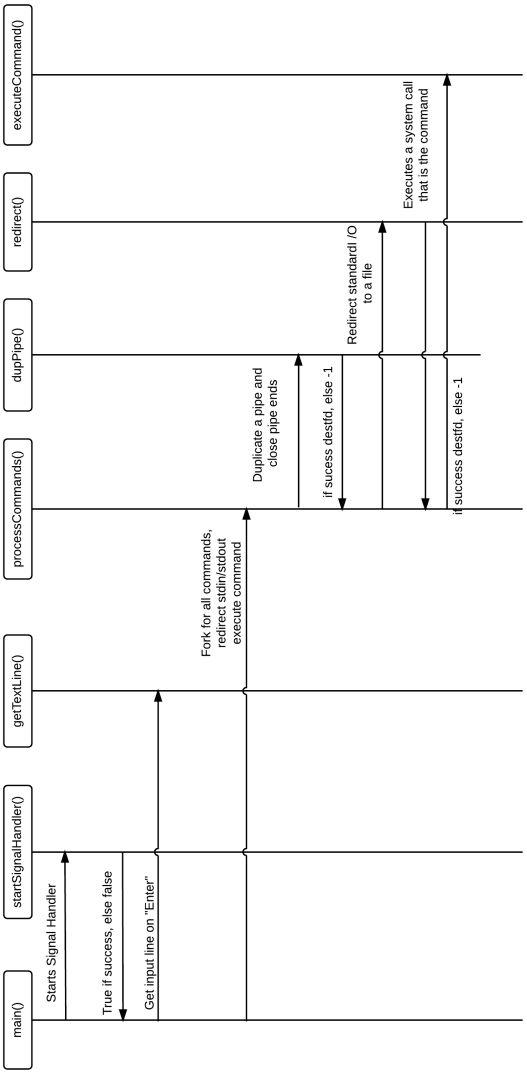


Figure 1: Anropsdiagram

### 3.3 Processkommunikation

För varje process som ska köras så görs en fork, detta skapar en barnprocess till *mish* och barnet kommer då att omdirigera sin standard in/out beroende på hur kommandot som användaren skrev in såg ut och därefter kommer barnet att exekvera det systemanrop som det fått av struct arrayen.

*En omdirigering av standard in, så programmet läser från infil*
Kommando: **wc** < **infil**

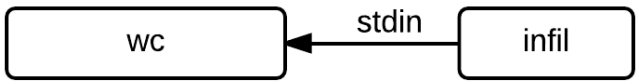


Figure 2: wc < infil

*En omdirigering av standard out, så ls skriver till utfil*
Kommando: **ls** > **utfil**

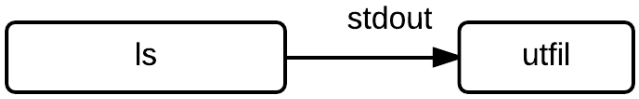


Figure 3: ls > utfil

*En pipa mellan ls och wc, gör så att ls utdata kommer att läsas av wc.*
Kommando: **ls** | **wc**

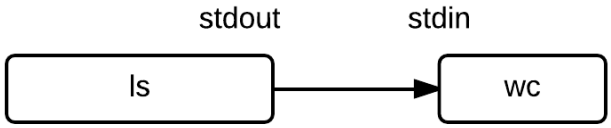


Figure 4: ls | wc

*Ett kommando som använder sig av en pipa, en infil och en utfil*
Kommando: **ls** | **wc**

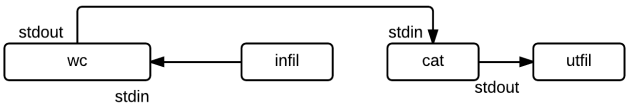


Figure 5: wc < infil | cat > utfil

# 4 Lösningens begränsningar

*I detta avsnitt beskrivs alla begränsningar som lösningen av uppgiften innehåller. Detta innefattar även funna begränsningar som strider mot specifikationen.*

Om användaren skriver *fil* > *fil2* dvs att *fil* ska skriva till *fil2* vilket ej kan fungera för att *fil* är en fil inte ett kommando som gör något så kommer fil2 ändå att skapas.

En begränsning är att *mish* inte stödjer piltangenter såsom en vanlig terminal gör att man kan flytta positionen man skriver på och ta bort tecken mitt i det man skrivit.

Ännu en begränsning är att *mish* inte stödjer historik funktionen som de vanliga terminalerna använder sig av att man med hjälp av upp och ner piltangenterna kan blädda igenom vad man skrivit förr.