UMEÅ UNIVERSITET
Instituionen förDatavetenskap

9 november 2018

5DV088: Systemnära programmering
Hösten 2018, 7.5p

Rapport för

Laboration 3 - mish

Name Buster Hultgren Wärn
E-mail dv17bhn@cs.umu.se

Kursansvarig
Mikael Rännar
Handledare
Klas af Geijerstam, Klas af Geijerstam, Elias Åström

Innehåll

1 Introduktion

Laboration 3 - mish - går ut på att skriva ett eget skalprogram. Ett skalprogram är en gränsyta till ett operativssystems kärna. Den sköter hand om bakomliggande processer när vissa operationer kallas på. Program kan startas härifrån och skalprogrammet tar hand om argument, omdirektion av standard I/O samt pipor emellan program. Skalprogrammet körs oftasst via ett command-line user interface - en terminal. Skalprogrammet ska kunna ta in ett antal kommandon, separerade med pipor (|), och kunna omdiregera deras standard input (stdin) och/eller standard output (stdout) mellan varandra via pipor. Det första respektive sista kommandots stdin/stdout ska även kunnas omdiregera till/från textfiler. [1]

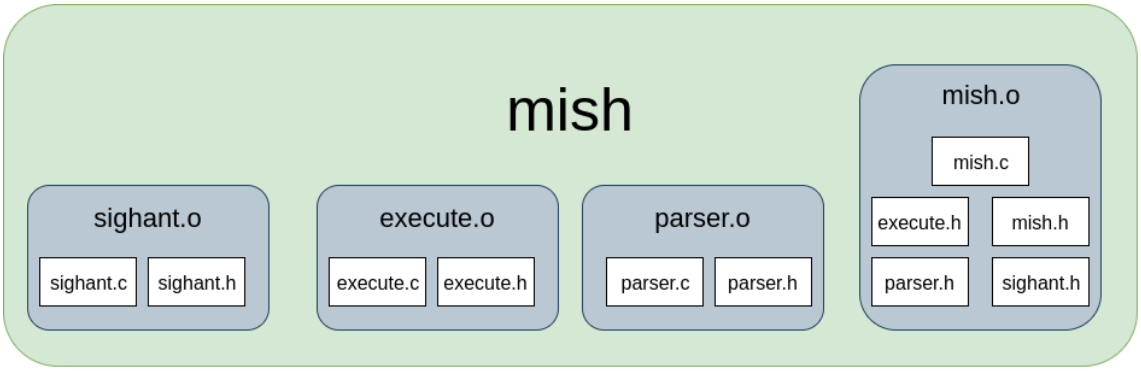
1.1 Filer, kompilering och körning

Mish är uppbyggt och kompileras med 9 olika filer (inkl. Makefile):

1. execute.c
2. execute.h
3. Makefile
4. mish.c
5. mish.h
6. parser.c
7. parser.h
8. sighant.c
9. sighant.h

För att kompilera programmet behöver endast *make* skrivas in i terminalen så kompileras programmet via GCC med instruktionerna givna i make-filen.

Makefilen kompilerar fyra stycken objekt filer. Dessa filer kompileras sedan till en exekverbar fil *mish*. Figur 1 beskriver förhållanden i hur exekverbara filen, objektfilen, .c och .h relaterar till varandra.



Figur 1: Makefile relations

Flaggor som används under kompilering är följande:

```
-std=gnu11 -g -Wall -Wextra -Werror -Wmissing-declarations -Wmissing-prototypes
-Werror-implicit-function-declaration -Wreturn-type -Wparentheses -Wunused
-Wold-style-definition -Wundef -Wshadow -Wstrict-prototypes -Wswitch-default
-Wunreachable-code
```

När exekverbara filen *mish* är kompilerad så körs den via terminalen genom kommandot.

```
./mish
```

1.2 Åtkomst och användarhandledning

Alla 9 filer ligger under /home/dv17/dv17bhn/edu/sysprog/lab3 på institutionens datorer.

Parser.c och parser.h är filer som givna filer av kursen för att underlätta med inläsning av kommandon som ges till skalprogrammet. Parser läser av en sträng ifrån stdin och omvandlar raden till kommandon, som är separerade av pipor (|). Parser kan läser även av om ett kommandos standard output/input ska diregeras om till/från en fil.

Execute (execute.c och execute.h) har en gränsyta som är given av kursen, men implementerad av författaren. Gränsytan har två operationer: att kunna duplicera en standard input/output till en pipa, och sen att kunna omdiregera en standard output/input till/från en fil.

Sighant (sighant.c och sighant.h) är en signalhanterare. I *mish* används endast sighant ifall *SIGINT* skickas till en körning av *mish*.

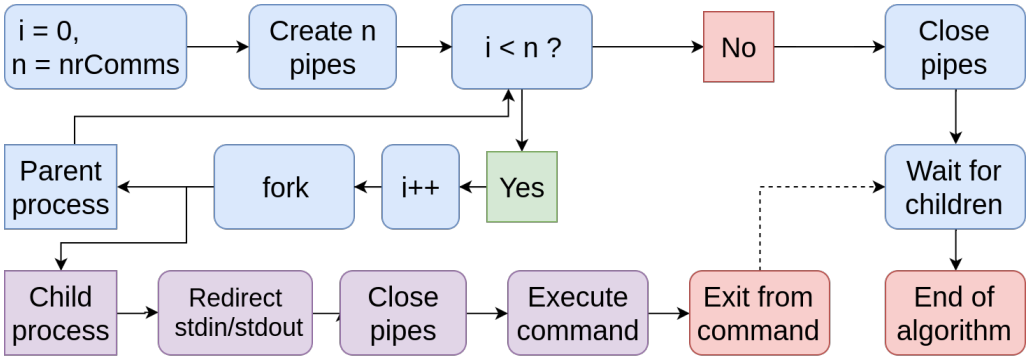
Mish (mish.c och mish.h) är huvuddelen av programmet. Här finns en prompt till skalet, två interna kommandon för *echo* och *cd* och olika kommandon kan exekveras. *Mish* skapar upp barnprocesser för varje kommando som används och skapar upp pipor emellan barnprocesser. *Mish* använder är beroende av alla andra prgram för att fungera.

2 Algoritmbeskrivning

Alrgoritmen i figur 2 visar hur förälderprocessen skapar upp barnprocesser via systemanropet fork() och vad som sker i de båda processerna. Först så sätts två variabler - i som en iterator och n som beskriver hur många kommandon som ska hanteras. För n kommandon ska $n-1$ pipor skapas. Efter detta itererar förälderprocessen n gånger och skapar varje gång upp ett barn. Då ett barn skapas upp delas flödet mellan barnet och föräldern på följande sätt:

Då ett barn skapas så omdiregeras dess stdin/stdout om, via systemanropet dup2(), till antingen en pipa eller en fil beroende på specifika villkor (mer om detta i kapitel 3.2). Efter detta stängs piporna, alltså fildeskriporerna som pekar på pipan. Detta sker då endast i barnprocessen och påverkar inte förälderprocessen eller andra barnprocesser då de **inte** delar något minne. Nu exekverar barnet sitt kommando, via systemaropet execvp(), och det kommandot tar över processen. Processen stängs sedan från det kommandot.

Efter att förälderprocessen har skapat upp ett barn så kollar det ifall det har skapat tillräckligt många barn genom att jämföra i med n. Ifall $i < n$ så fortsätter den att skapa barnprocesser. Annars, så stänger den sina pipor, väntar på att alla barnprocesser har kört färdigt. När de avslutas så är algoritmen färdig.



Figur 2: Algoritm: fork

3 Systembeskrivning

Skalets centrala del är mish.c. Här sköts prompten, signaler omdiregeras och nya processer körs. Även interna kommandon “echo” och “cd” är skrivna här.

3.1 Prompt, parser och kommandon

Prompten är byggd med en while-snurra som printar “textitmish%” till stderr. En buffer läser in från stdin, och såvida buffern inte innehåller strängarna "*exit*", "*quit*"eller end of file (EOF) så kommer snurran att fortsätta. I andra fall så används gränsytan i *parser* för att läsa in bufferten och göra om den till en eller flera *command structs*. Dessa structs är uppbyggda för att hålla ett bash kommando med numrerat antal argument. Fältet *argv* lagrar kommandot orch argumenten,

medans *argc* håller koll på hur många argument som används. Ifall ett kommandos stdin/stdout ska omdiregeras till en fil sparas detta i variablerna *infile* och *outfile*. Parser skiljer på olika kommandon med pipor (|) och läser av in/utfiler med hjälp av $<$ och $>$.

Om följande rad läses in:

```
cat < words.txt | wc -c > file.txt
```

skapas alltså två stycken structs med informationen.

```
                                    command 1
argv = [‘‘cat’’]
argc = 1
infile = words.txt
outfile = NULL

                                    command 2

argv = [‘‘wc’’, ‘‘-c’’]
argc = 2
infile = NULL
outfile = file.txt
```

Alla kommandon skickas sedan in till funktionen executeExternalCommands(), som ser till att var och ett av kommandona exekveras i rätt ordning samt omdiregerar deras stdin/stdout.

3.2 Olika processer och omdiregering

Varje kommando som ska exekvera kommer att få en egen process som är skild från huvudprocessen. Ifrån denna process så kommer stdin/stdout att omdiregeras för att sedan systemanropet execvp() kallas på med kommandot och dess argument som sina argument. execvp() tar över en process med ett annat program. Om t.ex. foo skickas in som argument till execvp() så kommer processen att bytas från mish till foo. Notera att processerns fildeskriptorer är kvar, därmed stannar tidigare omdiregeringar som i gjord i processen.

Barnprocesser skapas via systemanropet fork(), som skapar upp en ny process. Barnprocesser skapas ifrån en snurra och alogritmen för detta är beskrivet i kapitel 2.

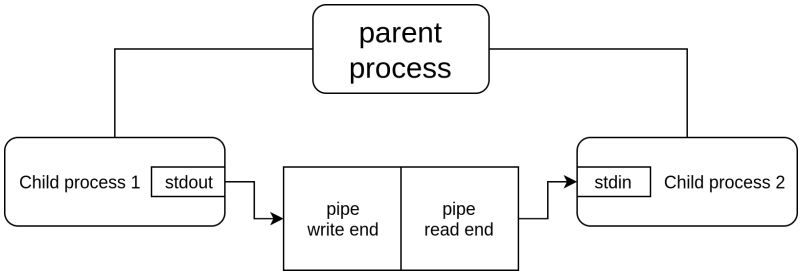
Barnprocessernas stdin/stdout omdiregeras på fyra sätt:

Ifall barnprocessen...

1. är det första kommandot **och** kommandot har en infil - stdin omdiregeras till infilen
2. är det sista kommandot **och** kommandot har en utfil - stdout omdiregeras till utfilen

3. **inte** är det första kommandot och $n > 1$ - stdin omdiregeras till en pipas läsände
4. **inte** är det sista kommandot och $n > 1$ - stdout omdiregeras till en pipas skrivände

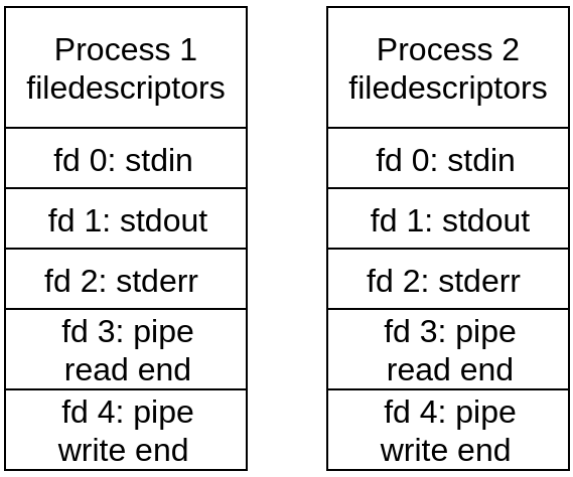
Mellan varje barnprocess (om det finns flera) finns det en pipa och det är pipans läs/skrivändar som stdin/stdout omdiregeras till - alltså för n st barn så kommer det finnas $n - 1$ pipor. Figur 3 illustrerar relationen mellan två barnprocesser och pipan som komminucerar emellan dem.



Figur 3: Pipa mellan två barnprocesser

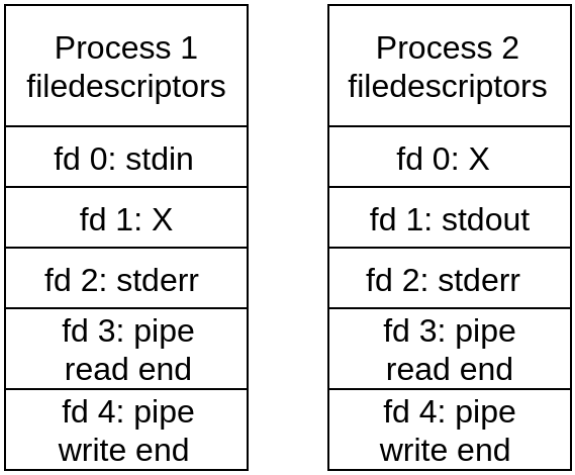
All omdiregering sker i execute.c och dess två funktioner duppipe() och redirect(). Dessa två funktioner kopierar fildeskriptorer från antingen en fil eller läs/skriv änden av en pipa för att lägga först stänga och sen lägga över den på stdin/stdout. duppipe() och redirect() fungerar essentiellt likadant med systemanropet dup2(). Tre steg är nödvändiga, och dessa visas genom figur 4 - 7 som illustrerar omdiregering av två barnprocessers stdin/stdout till en pipa.

Första steget är att skapa upp två stycken barnprocesser, där båda har fildeskriptorer till samma pipa (detta visat i kapitel 2). I figur 4 finns två processer då båda processerna ska diregeras om så att Process 1 stdout går till Process 2 stdin. Vid omdiregering av fil så behöver endast en process stdin/stdout omdiregeras, och istället för till en pipände så till en fildeskripor för den filen.



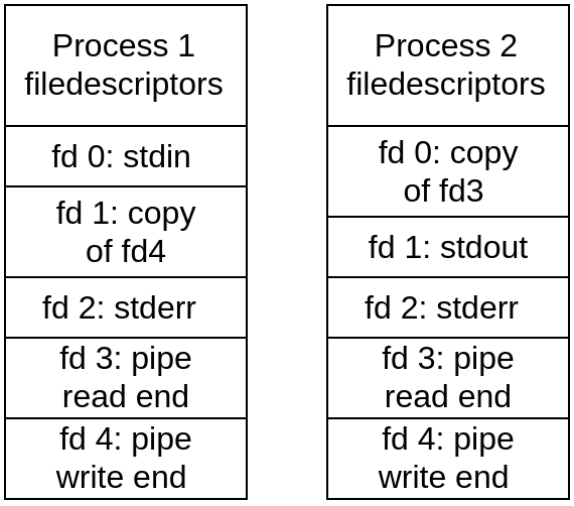
Figur 4: Omdiregering av två processers stdin/stdout steg 1

I det andra steget, visat i figur 5, stängs respektive fildeskriptor i varje process.



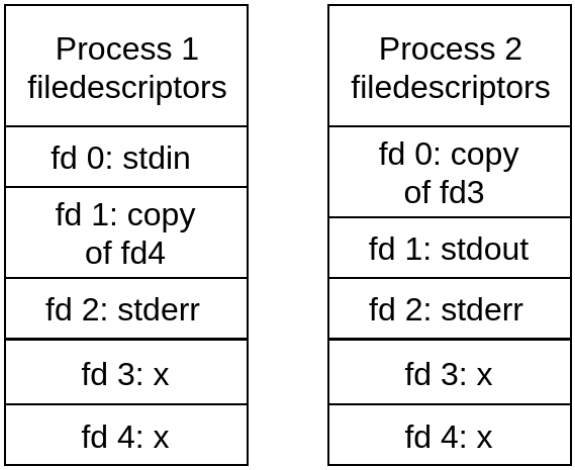
Figur 5: Omdiregering av två processers stdin/stdout steg 2

I steg tre, visat i 6, så kopieras fildeskriptorn från antingen läs/skriv änden av en pipa till respektive del för processens stdin/stdout.



Figur 6: Omdiregering av två processers stdin/stdout steg 3

Det fjärde och sista steget är att stänga fildeskriporerna till pipan, visat i figur 7.



Figur 7: Omdiregering av två processers stdin/stdout steg 4

Stegen behöver följas imperativt, men de olika processerna kommer inte att göra sina steg samtidigt. Process ordningnen är slumpmässig, men även irrelevant. Ifall ett senare kommando kör innan ett tidagare så diregeras stdin på process två som ovan beskrivt. När execvp() sedan kör så väntar processen på input från sin stdin - alltså från pipans läsende. När den första processen sedan kör och skriver ut till pipans skrivände så plockar den andra processen upp detta.

3.3 Interna kommandon

Två kommandon - “echo” och “cd” - är implementerade i mish. Echo skriver ut en sträng till stdout och skippar citat-tecken ("), så länge de inte har ett omvänt snedstreck framför sig. Cd ändrar den nuvarande katalogen. Den tar in ett argument, som är sökväg från den nuvarande katalogen till den nya. Om inget argument ges så går den tillbaka till hemkatalogen.

Inget av dessa kommandon kan omdiregeras på något sätt. In/utfilar kommer att ignoreras, och ifall dessa kommandon pipas med något annat så kommer ett felmeddelande skrivas ut.

3.4 Signalhantering

Den edna signal som behöver hanteras, specifierat under kapitel 4.1, är SIGINT [1]. Signalhanteringen sker i sighant:s gränsyta. Dess funktion signalHander() tar emot en signal som heltal och en funktionspekare till en annan signalhanteringsfunktion. signalHandler skapar då en sigaction struct med flaggan SA_RESTART. Den nya signalhanteringsfunktion skickas tillsammans med signalen som argument till sigaction() som hanterar signalen.

signalHandler() är skriven så att den är generell, men den enda signalhanterings funktion den tar emot är terminateChildren() som är skriven i mish:s gränsyta. terminateChildren() skickar via systemanropet kill() signalen SIGINT till varje barnprocess (om det så finns några) så att barnen terminerar. terminateChildren() håller reda på vilka barnprocesser som är igång genom de globala varierna NRCHILDREN, ett heltal som håller koll på hur många processer som lever, och

CHILDPIDS, ett fält som håller koll på alla barnprocessers PID.

3.5 Flödesschema

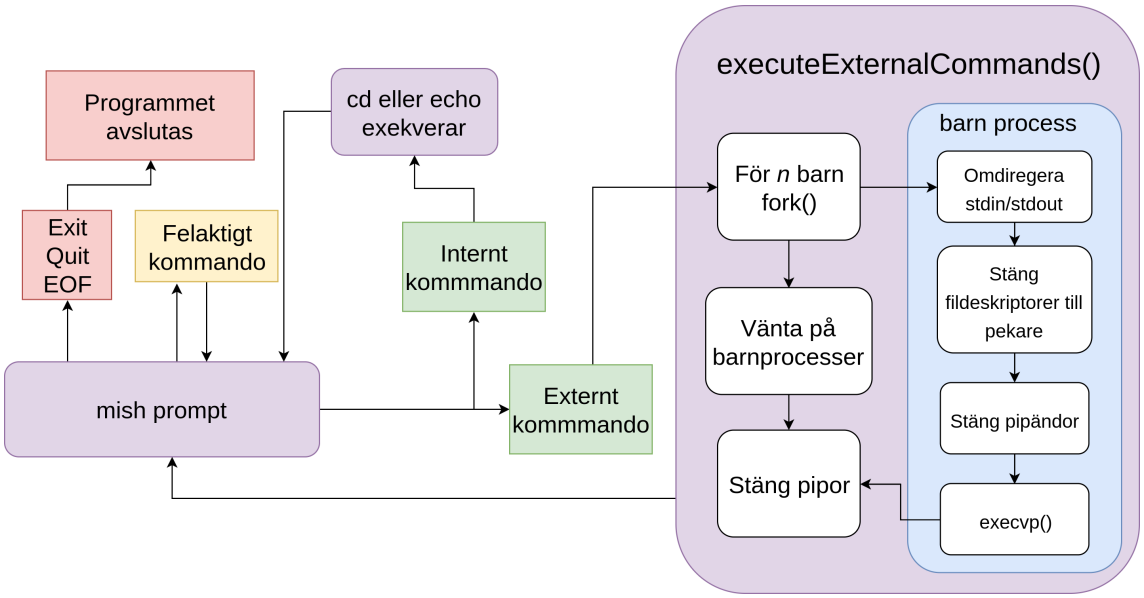
Det första som sker i programmet är prompten. Den läser in en rad från stdin, och gör via parser om detta till ett antal kommandon (såvida den inte läser "quit", "exit"eller EOF). Alla kommandon skickas till executeExternalCommands().

Det första executeExternalCommands() gör är att för n st kommandon skapa $n-1$ pipor. En for-snurra öppnas sedan som anropar fork() och skapar en barnprocess till varje kommando. När alla barn har skapats upp så stänger föräldern sin fildeskriptor till alla pipor.

Varje barnprocess kommer diregera sina sta $i < n$ diregeras stdout till skrivänden av pipa i, och för varje barnprocess $i > 1$ så diregeras stdin till läsenden pipa $i-1$. Om $i = 1$ eller $i = n$ och kommandot ska omdiregeras till fil så kommer detta att ske. När barnprocessen är omdiregerad så stänger den sina fildeskriptorer till piporna för att sedan sedan köra execvp() med sina kommandon.

Förälder processen väntar (med systemanropet waitpid()) på att barnprocesserna kört färdigt. Efter detta så avslutas executeExternalCommands och prompten väntar på ett nytt input.

Figur 8 visar ett flöversiktligt flödesschema för hur programmet exekverar.

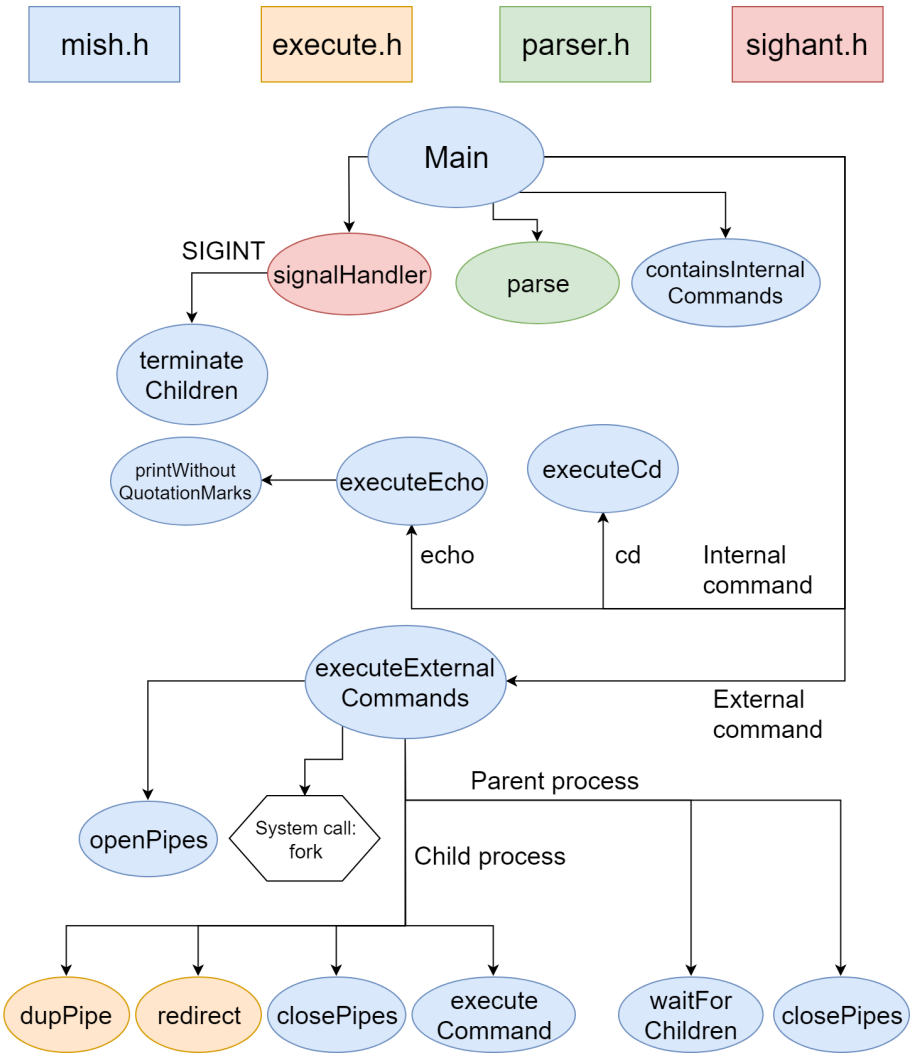


Figur 8: Flödesschema över mish

3.6 anropsdiagram

Anropsdiagrammet i Figur 9 visar endast anropen till olika funktioner och tar endast hänsyn till enkel logik i flödet (t.ex. är det en barn eller förälderprocess). Alltså kommer inte olika snurror eller

mer advancerade *if* satser med i diagrammet. Förutom ett undantag är det bara gränsytorna till mish, execute, parser och sighant som visas. Systemanrop som pipe() och dup2() visas alltså inte här. Anropsdiagrammet är också färkodat mellan funktion och gränsyta. De fyra översta rutorna beskriver denna färgkodning.



Figur 9: Anropsdiagram för mish

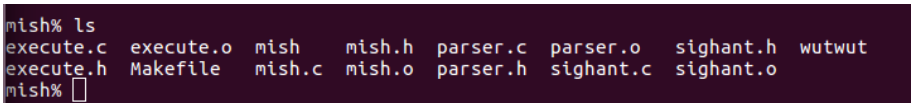
4 Resultat och testkörningar

Det är lite svårt att veta exakt vad som krävs för att veta att mish ska fungera. Chansen finns att programmet har någon bug vid en viss indata. Mish har däremot klarat testerna på labres, och den klarar testerna ifrån kapitel 11 i specifikationen [1]. Ytterligare 10 st tester har gjorts för att i denna rapport visa hur mish hanterar viss indata. Figur 10 - 14 är olika typer av tester för att se att mish fungerar med korrekt indata. Testerna i figur 15 - 19 är tester med fel indata för att se

att mish hanterar detta.

Figur 10 tar in kommandot ls, tillsammans med ett argument, för att mish klarar av ett kommando.

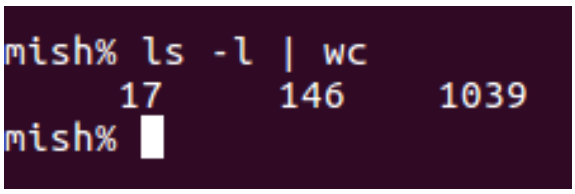
Förväntat resultat: kommandot ls kör.



Figur 10: Korrekt indata. Ett kommando körs.

Mish ska även kunna hantera pipor emellan kommandon. I figur 11 visas resultatet då två stycken kommandon används tillsammans med en pipa.

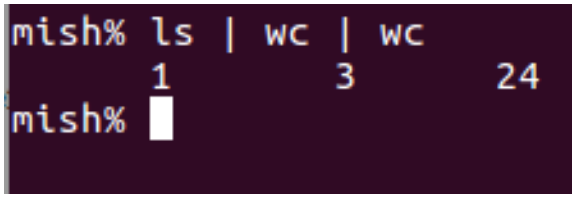
Förväntat resultat: ls kör med flaggan l, vars stdout omdiregeras till stdin för kommandot wc.



Figur 11: Korrekt indata. Två kommandon körs med en pipa.

Figur 12 visar resultatet där ännu en pipa läggs till. Det kan antas härefter att mish kan ta emot godtyckligt många kommandon.

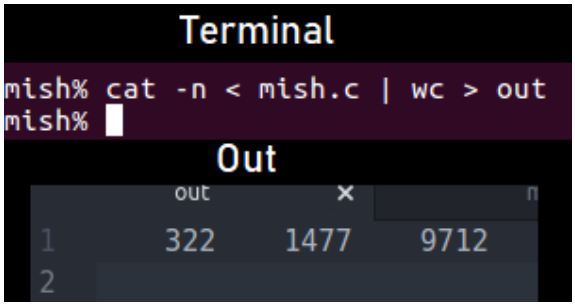
Förväntat resultat: Tre kommandon kör - ls, wc, wc - och stdin/stdout pipas emellan dem.



Figur 12: Korrekt indata. Flera kommandon körs med pipor.

Omdiregering av både stdin samt stdout testas samtidigt med en pipa, där ls tar in mish.c och pipar resultatet till wc, som omdiregerar sin stdout till filen “out”. Resultatet finns i 13.

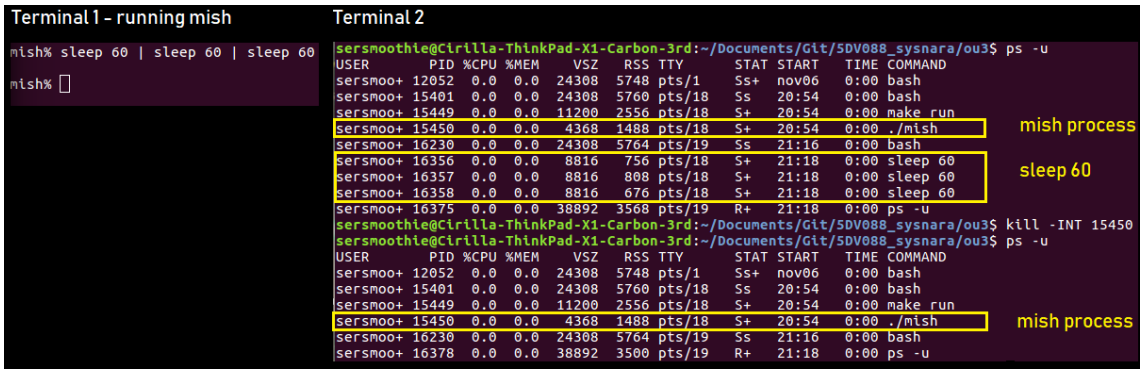
Förväntat resultat: cat med flaggan n kör vars stdin omdiregerats till mish.c. cat pipas sedan till kommandot wc, som skriver resultatet till filen “out”.



Figur 13: Korrekt indata. Omdriregering av stdin/stdout till filer samt pipor.

I figur 14 visas det sista testresultatet från kapitel 11 av specifikationen [1]. I mish skapas 3 barnprocesser som sover i 60 sekunder. När en SIGINT skickas ifrån en annan process till mish, samtidigt som de tre barnen kör, ska mish inte få några zombie processer.

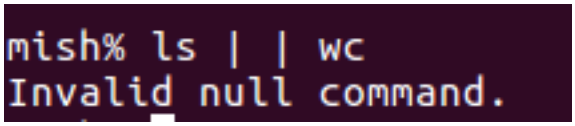
Förväntat resultat: Efter att SIGINT skickats till mish ska det inte finnas några barnprocesser kvar.



Figur 14: Korrekt indata. Zombieprocesser.

Mish ska inte ta emot ett tomt kommando ifrån prompten. Detta test visas i figur 15.

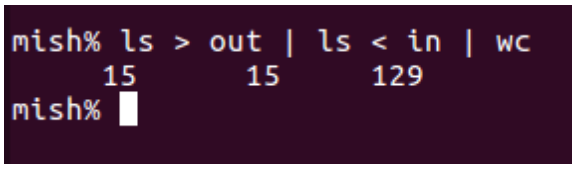
Förväntat resultat: Mish ska inte starta några barnprocesser utan återgå till prompten.



Figur 15: Felaktig indata. Tomt kommando.

Då ett kommando tas in med en < eller > pil för att omdiregera sitt stdin/stdout samtidigt som det ska pipas emellan två kommandon så ska det strunta i pilarna. Resultatet för detta visas i figur 16.

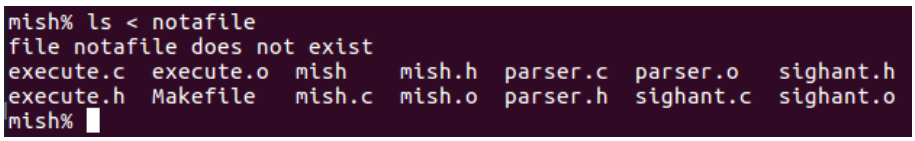
Förväntat resultat: De felaktiga omdiregeringarna ska ignoreras och raden “ls | ls | wc” ska köra.



Figur 16: Felaktig indata. Omdiregering av kommandos stdin/stdiout till fil som redan pipas.

Figur 17 visar resultatet då ett kommandos stdin ska omdiregeras till fil, men filen finns inte.

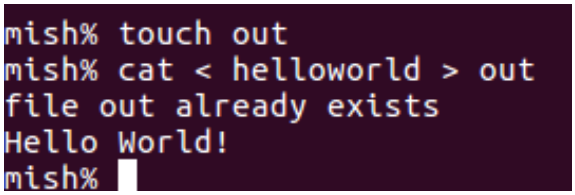
Förväntat resultat: Felmeddelande ska skrivas ut och omdiregeringenpå stdin ska ignoreras.



Figur 17: Felaktig indata. Omdiregering av stdin till fil som inte finns.

Likande testet i 17, så visar 18 samma sak fast då stdout omdiregeras till en fil som redan finns.

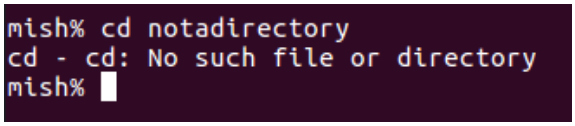
Förväntat resultat: Felmeddelande ska skrivas ut och omdiregeringen på stdout ska ignoreras.



Figur 18: Felaktig indata. Omdiregering av stdout till fil som redan finns.

Figur 19 visar resultatet då ett cd skickas med ett argument som inte är ett tillgängligt directory.

Förväntat resultat: Felmeddelande ska skrivas ut och kommandot ska inte köra.



Figur 19: Felaktig indata. cd till directory som inte finns.

5 Diskussion

Detta har varit en utmanande och rolig laboration. Det svåraste var att börja med laborationen - att omvandla den teoretsika kunskapen som vi har fått ifrån föreläsningar till praktiken. Det är en sak att sitta och läsa om fork() och execvp(), men att faktiskt skriva syntaxen är en helt annan.

Då detta svåra steg var över känns det dock faktiskt som att jag har lärt mig någonting. T.ex. jag förstår mig på varför en fork() behöver göras nu istället för att veta att jag kommer behöva använda fork().

Det krävdes inte heller så värst mycket kod vilket var skönt, och vi behövde inte riktigt hantera strängar själv.

Jag vill gärna tacka de studenter som går kursen med mig och har hjälpt mig förstå några av de essentiella delarana. Sen har handledare Klas hjälpt mig förstå saker där mina studenter inte har lyckats.

I slutändan är det en bra uppgift som har omvandlat teoretisk förståelse till praktisk kunskap. Om någonting behöver förbättras så kan det vara bra att ge ledtrådar till var studenter bör börja om de fastnar. Det svåraste steget var att komma igång.

6 Begränsningar

De främsta begränsningarna är de interna kommandon “echo” och “cd”. Eftersom kommandona är skrivna internt så går det inte att köra execvp() med dem. Detta gör att fork() blir problematiskt med dessa kommandon och det blir svårt att omdiregera stdin/stdout. Det är dock inte ett krav enligt kapitel 5 i specifikationen [1].

7 Referenser

[1] Institutionen för datavetenskap, Umeå universitet, Laboration 3 — mish,"Cambro, 2018-09-13. [Online] Tillgänglig: https://github.com/SerSmoothie/5DV088$_{s}ysnara/blob/master/ou3/mish - specification.pdf$.[$Hmtad$: 2018 − 11 − 08]