

Laboration 3 — mish

2017-09-08

Innehåll

1	Introduktion	
2	Formalia	
2.1	Handledning	
3	Viktigt!	
4	Hjälpmaterial till uppgiften	
4.1	Exempel på hur <code>parse</code> används	
5	Exekvering av kommandon och program	
6	Tips, hjälp och saker att tänka på	
7	Testkörning	
8	Exempelsession	
9	Filer	
9.1	Relevanta avsnitt i kurslitteraturen	
10	Rapport och källkod	

1 Introduktion

Ett `shell` är det text-baserat program som körs då du öppnar en terminal. Programmet skriver ut en så kallad “prompt” och väntar på kommandon från användaren. På labbdatorerna kör `tcsh` sålänge du inte bytt, du kan skriva `echo $SHELL` i en terminal för att få reda på vilket shell du kör.

I den här laborationen ska du implementera ett minimalt shell, `mish`. Det ska klara av exekvering av kommandon och program, pipor mellan program och omdirigering av standard input (`stdin`) och standard output (`stdout`).

2 Formalia

Notera att laborationen *måste* lösas *enskilt*. Läs [riktlinjer för labgenomförande](#) för att ha en tydlig bild om vad det innebär att jobba enskilt och notera konsekvenserna som följer av att strida mot dessa regler.

Se avsnittet “Krav på hur inlämningen ska ske” för mer information om inlämningen.

2.1 Handledning

Skicka era handledningsfrågor till 5dv088-handl@cs.umu.se. Läs [denna text](#) och [även denna](#) som referens när ni skriver era frågor. Oftast kommer ni finna att ni löser problemen själva när ni gör detta.

3 Viktigt!

På uppmaning från institutionens support-personal så vill vi att ni gör era testkörningar på någon av datorerna *itchy* eller *scratchy*. Avsnittet [Testkörning](#) beskriver hur ni gör detta.

4 Hjälpmaterial till uppgiften

Till eran hjälp tillhandahålls filerna [parser.c](#) och [parser.h](#), som tillhandahåller en funktion som läser av en text-sträng och delar upp den i kommandon. Kommandon separeras med tecknet `|`, en så kallad “pipa”.

Funktionens argument, `const char *line` och `command comLine[]`, beskriver en textsträng med kommandon (formatet beskrivs i `.c` filen) och en *pekare* till en array. Den returnerar en `int` som representerar det antal kommandon funktionen “läste”.

Den fyller `comLine` med element på formen

```
typedef struct command_t {
    char **argv;
    int argc;
    char *infile;
    char *outfile;
    int internal;
} command;
```

där

- `argv` är en array med alla argument till kommandot (`argv[0]` är namnet på kommandot som vanligt)
- `argc` är antal element i `argv`
- `infile` är namn på den fil som standard input ska omdirigeras från eller `NULL` om ingen omdirigering är angiven

- `outfile` är namn på filen som standard output ska omdirigeras till, eller `NULL`
- `internal` är en flagga som talar om ifall kommandot är ett internt kommando. Detta fält sätts inte av parsern, men du kan använda den för att skilja mellan interna och externa kommandon.

4.1 Exempel på hur parse används

Som referensmaterial till hur `parse` kan anropas och hur ni kan använda er av dess returvärde och vad för information som `comLine` innehåller efter ett anrop så kan ni referera till [parser_examples.c](#). När den filen kompileras och körs skriver den ut följande,

```
The command-line: "foo < bar.txt" yields a struct that looks like this:
{
  Argv: ["foo"]
  Argc: 1
  Infile: bar.txt
  Outfile: (null)
}

The command-line: "echo hi | grep hi" yields two structs that looks like this:
{
  Argv: ["echo", "hi"]
  Argc: 2
  Infile: (null)
  Outfile: (null)
}

{
  Argv: ["grep", "hi"]
  Argc: 2
  Infile: (null)
  Outfile: (null)
}
```

`mish` ska dessutom hantera signalen `SIGINT` och på den avbryta exekveringen av alla uppstartade program, skriva ut en ny prompt, och efter det vänta på nya kommandon. Detta kan lösas på två sätt, och det enklaste är att ha en array innehållande alla uppstartade programs `PID`, och iterera igenom den arrayen och sända signalen `SIGINT` till vart och ett av programmen. Signalhanteringsfunktionerna ska finnas i filerna `sighant.c` och `sighant.h`.

`mish` ska alltså skriva ut en prompt, läsa in en kommandorad, parsea den med hjälp av parsern och därefter exekvera alla kommandon som parsern returnerar. Vid exekveringen ska det hantera argument, koppla ihop alla kommandon med pipor och hantera all omdirigering av input / output. Om läsning av nytt kommando ger `EOF` (dvs. användaren trycker `Ctrl-D`) så ska shellet avsluta.

Till eran hjälp finns [följande funktionsdeklarationer \(execute.h\)](#) som ni, med fördel, kan inledningsvis implementera. Detta kommer underlätta strukturen av erat program.

5 Exekvering av kommandon och program

`mish` ska kunna exekvera externa program samt interna kommandon. För varje externt program ska en process skapas som exekverar det programmet. Interna kommandon ska exekveras av huvudprocessen. `mish` ska vänta tills alla externa kommandon exekverat klart innan en ny prompt skrivs ut. För att förenkla lite grann antar vi att en kommandorad antingen består av en sekvens av externa kommandon eller ett internt kommando. Du behöver alltså inte bekymra dig om pipor mellan interna och externa kommandon, ej heller omdirigering av standard input och output för interna kommandon. De interna kommandon som ska implementeras är `cd` som ska byta working directory, samt `echo` som skriver ut alla sina argument på standard output.

Environment-variabeln `PATH` ska användas för att lokalisera de externa programmen som ska exekveras. Detta betyder inte att du själv ska leta reda på det externa programmet och köra det. Titta på man-sidan för `execvp` så får du veta vad som menas.

6 Tips, hjälp och saker att tänka på

- Använd rätt version av filerna `execute.h` och `parser.[c, h]`. Har du gjort laborationerna tidigare är det inte säkert att du har de rätta versionerna.
- De filer ni får av oss får inte ändras! Ni ska använda `execute.h`, `parser.c` samt `parser.h` i det skick de är när ni får dem av oss. Inga ändringar är tillåtna.
- Notera att er `echo` inte behöver vara lika avancerad som “vanliga” `echo`. Den behöver tex inte ta hänsyn till escapesekvenser utan behöver endast skriva ut sina argument rakt av. Tänk dock på hur och var mellanslag skrivs ut mellan argumenten till `echo`.
- Funktionen `redirect` som deklarerats i `execute.h` ska inte skriva över existerande filer. Om `stdout` ska omdirigeras till en fil, och filen redan finns, ska `redirect` returnera med en felkod, och `errno` satt till lämpligt värde.
- Då du testkör programmet, prova inte enbart korrekta kommandorader. Testa några felaktiga rader också och se hur ditt program hanterar detta.
- En sak som är bra att veta är att om en barnprocess core-dumpas skrivs inte alltid meddelandet “Segmentation fault (core dumped)” ut. Man kan dock se att en core dump inträffat genom att köra `ls` och se om filen `core` har skapats.
- I denna laboration blir man på vissa ställen tvungen att använda globala variabler. Försök dock undvika globala variabler där det är möjligt (detta gäller allmänt när man programmerar).

Er prompt bör skrivas ut med hjälp av nedanstående kodsnitt:

```
fprintf(stderr, "mish%% ");  
fflush(stderr);
```

7 Testkörning

Eftersom man på denna lab, om man gör fel, kan få till ett program som gör att man måste starta om datorn som man kör på så ber vi er undvika testkörningar på **salt** och **peppar** och istället testköra på någon av datorerna **itchy** eller **scratchy**.

Dessa båda maskiner är endast till för testkörningar och således borde inte så många bli “arga” om de måste starta om. Editeringen av koden sker lämpligen på någon annan dator (tex den lokala Unix/linux-maskinen eller på **salt/peppar**) medan kompilering och testen körs på **itchy** eller **scratchy**.

För att begränsa problemen för andra användare (och sig själv) kan man göra följande när man ska testköra: Om man kör **tsch** (default)

- Ena inloggningen (på **itchy** eller **scratchy**), där man testkör: Kör kommandot (innan man börjar testkörningarna) **limit maxproc 50** Då begränsas den inloggningen till max 50 processer, fast andra inloggnings har kvar default 1024.
- Andra inloggningen (på samma maskin som ovan): kör t.ex **top** för att se hur det ser ut just nu. För att ta bort sina egna **mish**-processer (vid behov), från andra inloggningen: **kill -STOP -u mitt-username mish** **kill -9 -u mitt-username mish** Dvs först skicka ‘frys alla mina processer som heter mish’, så de inte kan yngla av sig mer. Sen större hammare.

8 Exempelsession

Här följer en exempelkörning av hur ett färdigt program kan se ut, anropas och köras.

```
itchy:~/edu/sysprog/lab3> ./mish
mish% cd ..
mish% ls
lab1 lab2 lab3 lab4
mish% echo hej
hej
mish% cd lab3
mish% cat mish.h | tail -5 > apa
mish% cat apa
int externCommands(command cmds[], int argc);
void rem(pid_t pid, int proc[], int nrp);
void echo(int argc, char *argv[]);
int cd(int argc, char *argv[]);
#endif
```

9 Filer

Alla filer ska finnas i katalogen `~/edu/sysprog/lab3/` och vara läsbara för oss labrättare. De ska heta **mish.c**, **mish.h**, **execute.c**, **execute.h**, **parser.c**,

`parser.h`, `sighant.c`, `sighant.h` samt Makefile (ja, du måste använda en Makefile). Din lösning ska gå att kompilera, köra och rätta utan att vi ska behöva kopiera några andra filer.

9.1 Relevanta avsnitt i kurslitteraturen

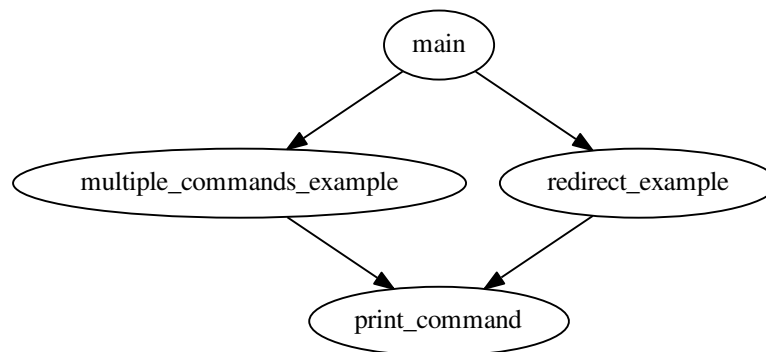
Figur 1.7 kan tjäna som en lämplig utgångspunkt för uppgiften, då den redogör för ett *väldigt* minimalt shell.

10 Rapport och källkod

Rapporten och källkoden skall följa riktlinjerna på [denna sida](#).

Systembeskrivning är en rubrik som kan variera relativt mycket mellan olika labbar hur den bör se ut. På just denna lab skall systembeskrivningen (förutom vad som nämns i de mer generella riktlinjerna) innehålla åtminstone följande:

- Anropsdiagram som visar vilka funktioner som anropar vilka, se nedan för ett exempel på hur ett sådant kan se ut. Figuren är ett anropsdiagram för [parser_examples.c](#).
- Gör en figur som illustrerar hur processer kommunicerar under körning. Lämpligen gör man någon typ av bild med processer och pipor utmarkerade, med pilar mellan dem som visar hur data skickas.¹ Exempel finns i tidigare tentors lösningsförslag, [bland annat i tentan från 2002](#).



Figur 1: Anropsdiagram för [parser_examples.c](#)

¹Denna figur behöver inte vara väldigt detaljerad. Ni behöver tex inte markera ut varenda fildeskriptor i processerna, utan det räcker att visa vilken process som läser/skriver till vilken pipa. Nu kan även utelämnas 'pid'.