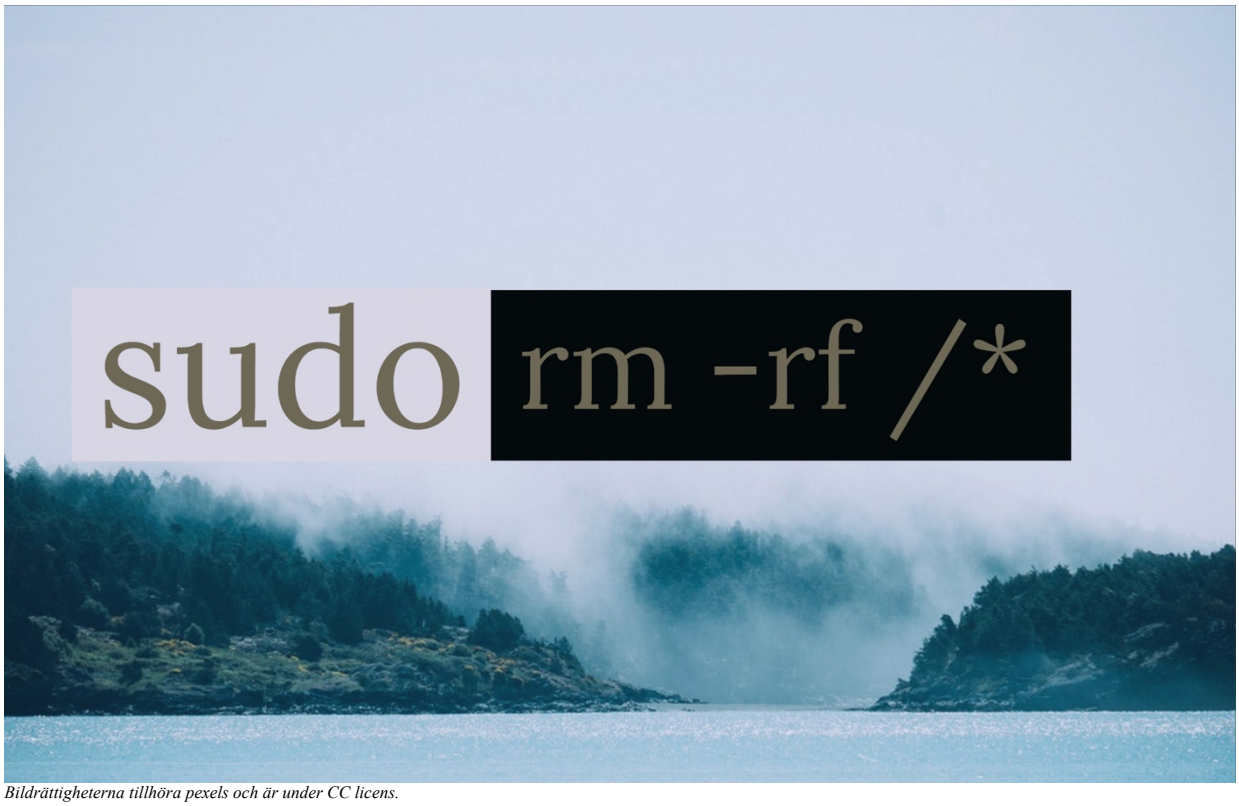
Systemnära programmering 5DV088HT19

Obligatorisk uppgift 3 MISH



Bildrättigheterna tillhöra pexels och är under CC licens.

Max Malmer(c17mmr)
Kontakt: c17mmr@cs.umu.se

2019-10-10, v.1.0

Innehållsförteckning

1. Introduktion

1.1 Bakgrund

Detta är en rapport som beskriver ett program och en implementation av ett minimal unix shell för kursen Systemnära Programmering 5DV088HT19.

1.2 Problembeskrivning

I denna uppgift så ska ett minimal unix shell implementeras. Det vill säga ett program som skriver ut en prompt och sedan läser in kommandon från användaren som vid en ''vanlig'' terminal körning som i exempelvis bash. Dom kommandon som ska implementeras internt är cd och echo. Att notera är att dom inte behöver vara lika avancerade som deras motparter i t.ex. bash. Men grundläggande funktion ska finnas. Bland annat så ska t.ex. cd utan argument sätta användaren i ''HOME'' mappen. Resterande kommandon ska exikveras externt. Dessa kommandon ska både kunna ta och skriva till filer samt ta pipes som fildeskriptorer mellan varandra.

2. Användarhandledning

2.1 Programmet

Programmet är indelat i 5 filer varav 4 .c filer med tillhörande .h filer och en makefil. Filerna är som följer: mish.c, execute.c, parser.c, sighant.c och makefile. Tecknet `$` används för början av en ny rad i valfri terminal, alt `mish%` vid fallet att mish är startat.

Programmet kompileras med följande kommando:

```
$make all
```

För att köra programmet använd följande kommando:

```
$./mish
```

När programmet startat så kommer du se följande prompt:

```
mish%
```

Programmet är då startat och mish prompten tar kommandon ungefär som en vanlig terminal.

För att avsluta programmet så skriv som instruerat exit eller använd Ctrl + D, annars så kan ett exempel på en indata du kan skriva vara:

```
cd ..
```

Ändrar din directory till den som är ovanför mappen du är i.

```
echo hejhej
```

Skriver ut ’’hej hej’’ till shellet.

```
ls | wc
```

Skriver ut antalet ord i de listade filerna i mappen du är i.

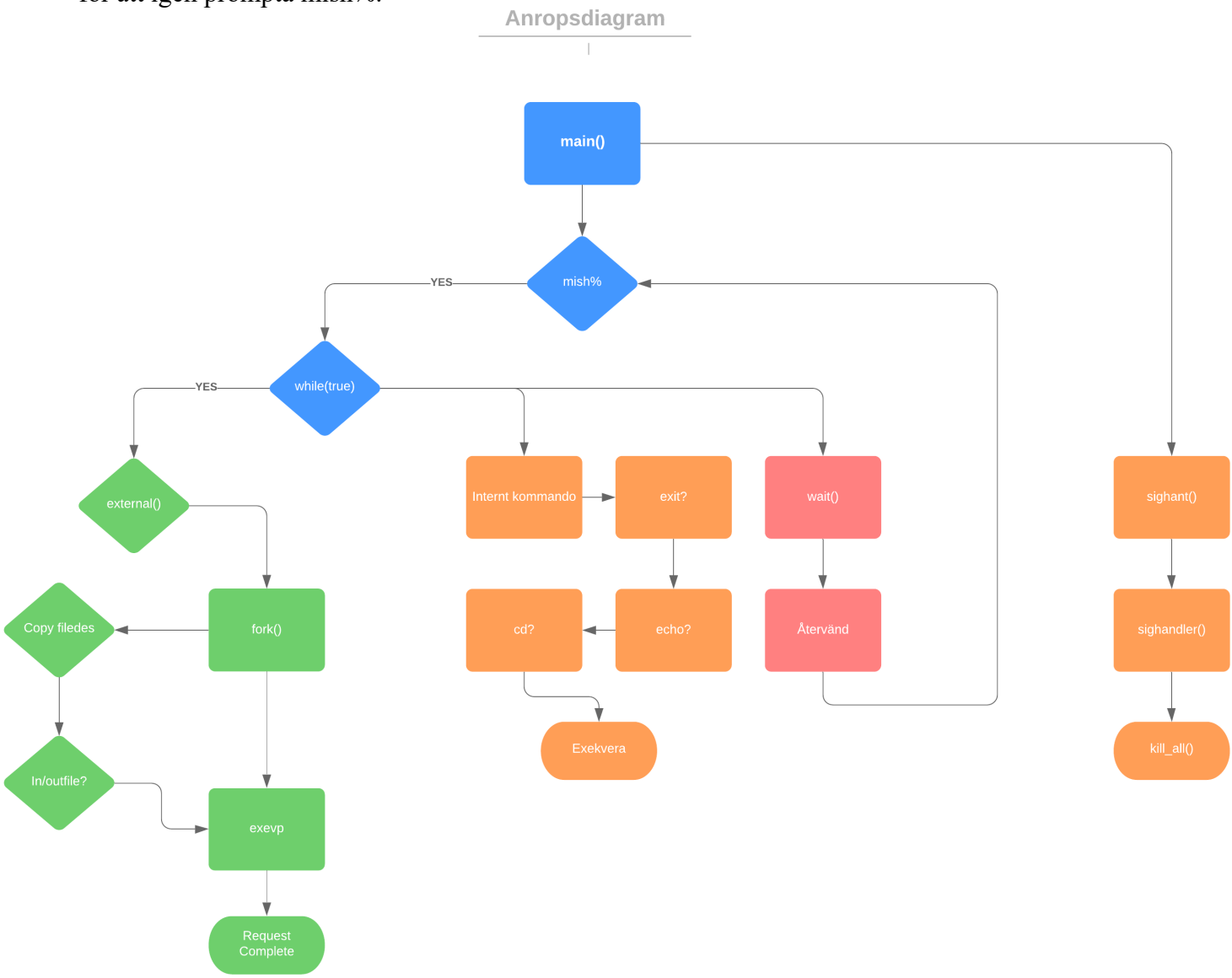
```
ls > filmedfiler.fil
```

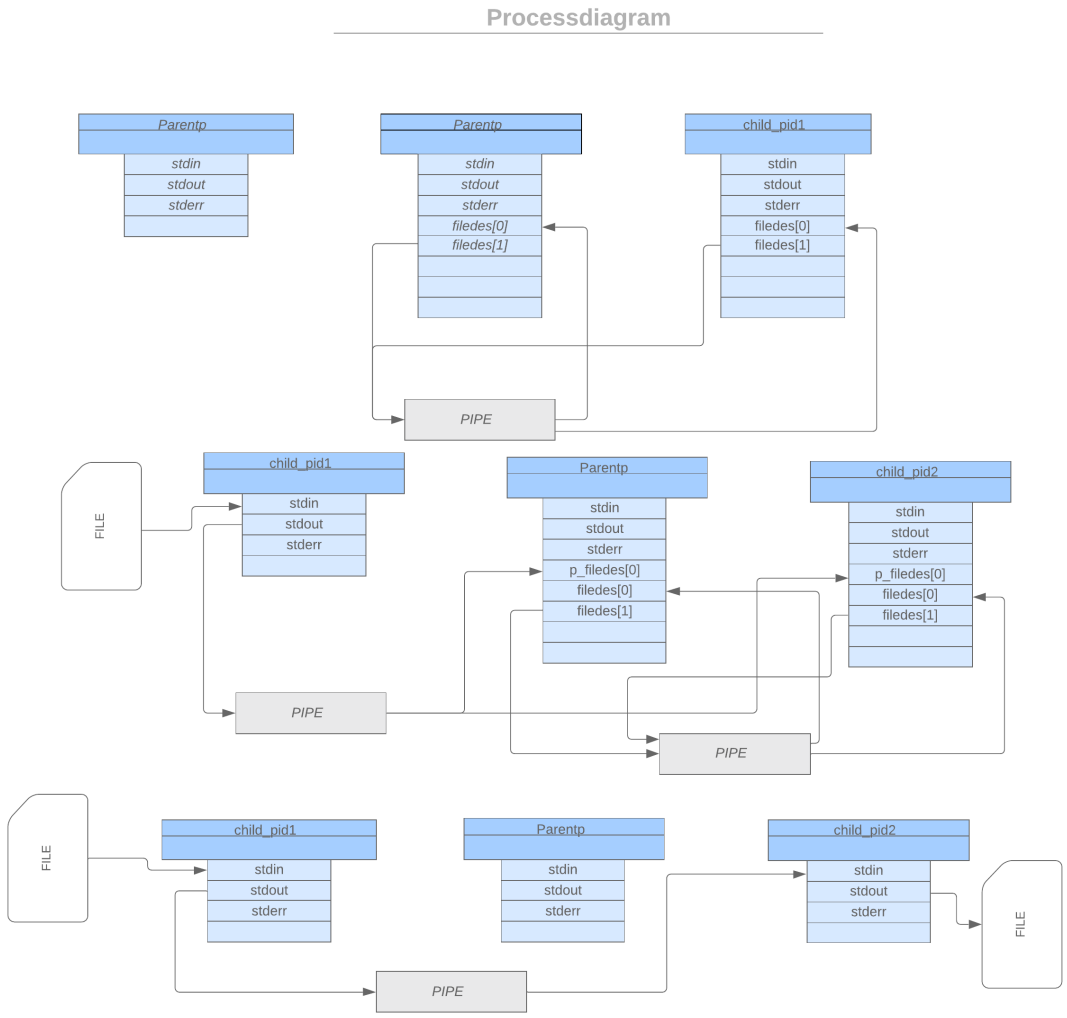
Skriver in ditt directory till en fil som heter filmedfiler.fil.

3. Systembeskrivning

3.1 Programmet

Programmet startar genom att initialisera nödvändiga variabler och sen promta mish%. Därefter så väntar den på användarens indata i form av kommandon. När en buffer från användaren lästs in så använder programmet ett parser för att dela upp kommandot i en statisk array av strukter där antalet argument, argumentet och om kommandot hade några in eller utfiler. Mish fortsätter sedan och kollar om kommandot är ett av dom interna (cd, exit eller echo). Den går om inte det vidare och kör dom externa kommandon som har angivits. Detta görs genom att använda funktionen fork() för att skapa en process som kör kommandot med execvp() för att sedan dö. Vid mer än ett kommando med pipes mellan så skrivs dessutom STDOUT över för de kommandon som inte är sist. För dom kommandon som inte är först så skrivs STDIN över till fildeskriptorn för detta förra kommandot algoritmen itererade över. Sedan så väntar huvudprogrammet på att dess barn ska avsluta. Programmet återvänder sedan tillbaka till början för att igen prompta mish%.





4. Algoritmbeskrivningar

Nedan beskrivs alla algoritmer i implementationen som inte är triviala eller som kommer med uppgiftens beskrivning.

4.1 Mish

`main()`: Är algoritmen som startar programmet. Den börjar med att skriva ut prompten och fortsätter sedan med att läsa in kommandot från användaren för att sedan skicka det till parsern. Efter det så kommer den om det är ett internt eller extern kommando och agerar därefter.

`internal_command()`: Är en algoritm som kollar om ett kommando är ett av dom interna (cd, echo, exit) och kör i sådana fall den algoritmen. Annars så gör den inget.

`external_command()`: En algoritm som exekverar externa kommandon genom att skapa separata processer för dom. Om det är mer än ett kommando så skapas även pipes mellan processerna sådant att dom förstkommande kommandona kan skicka data till dom efterkommande.

`file_redirect()`: Kollar om det finns en in eller utfil för kommandot. Om det är det sista eller första kommandot så kommer STDIN eller STDOUT fildescriptorn skrivas över med filens descriptor.

`cd()`: En enkel implementation av kommandot cd. Det byter directory för den nuvarande processen till det som anges till algoritmen. Om inget argument ges så återvänder kommandot en till ''HOME'' mappen.

`echo()`: Är en förenklad implementation av kommandot echo. Den tar in ett kommdo och skriver ut den fullständiga texten på det som kommer efteråt.

`kill_all()`: Stänger ned alla processer efter att ha fått en inputsignal.

4.2 Sighant

`sighandler();` En algoritm som väntar på att få en signal och därefter utför kill_all() funktionen om den får signalen SIGINT.

`sighant();` En algoritm som med startar hanterare som under programmets körning väntar på olika signaler från antingen användaren eller systemet.

5. Testkörningar

Som testkörning för att stresstesta programmet så använde jag mig av valgrind. Det finns ett kommando till Makefile som kör programmet i valgrind och det är:

```
make valgrind
```

Jag körde sen alla mina tester i en session är jag kallat ./mish i valgrind 4 gånger. I det läget så klarade jag dom enligt specifikationen rekommenderade testerna och många fler. Bland annat så testade jag gränsen på programmet genom att köra:

```
w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w
```

Som jag märkt så finns det helt klart vissa begräsningar med den här implementationen. Bland annat så vad jag har märkt så kommer det bli problematiskt om det skapas mer än ”MAX_COMMANDS”. Då kommer programmet hänga sig. Programmet kan heller inte köra mer än ”MAXLINELEN” i längd på inputsträngen, då krashar det med Segfault.
De system jag provat att köra min implementation på är: MacOS 10 High Sierra, Ubuntu 18.4 och Windows 10.

Jag kan inte garantera att implementationen kommer fungera felfritt på något annat än en maskin som kör antingen Debian 9 eller 10 som OS.

6. Reflektioner

Överlag så har detta varit en enormt rolig uppgift. Jag tror att det är en av dom roligaste sen jag kom till universitetet. Mycket som jag behövt förstå i och med att jag gjort uppgiften är saker som är uppenbara att jag faktiskt kommer ha användning för. Det var intressant att se vilka begränsningar sån här systemnära Unix programmering kan skapa, jag tycker att det ofta blir så att man löser ett problem på ett helt nytt sätt än vad jag tänkt mig. Sen så för min egen del så känner jag också att jag har lite svårt att förstå hur allt händer ibland. Särskilt i signalhanteraren så har jag mest gått efter dokumentation och testat mig fram till något som verkar vettigt. Men jag vet vad den abstrakt gör och det känns som att det för nu är nog bra.

Jag tror också att jag skulle kunnat göra både en bättre rapport och ett bättre kodat shell om jag hade haft mer tid till uppgiften. Den är mycket stor rent tidsmässigt och jag har haft mycket med andra kurser samtidigt.

7. Referenser

1. https://www.geeksforgeeks.org/piping-in-unix-or-linux/#targetText=A%20pipe%20is%20a%20form,program%2Fprocess%20for%20further%20processing. 2019-10-10
2. https://www.gnu.org/home.en.html 2019-10-10
3. https://unix.stackexchange.com/questions/7138/fast-interrupts-in-linux 2019-10-10
4. https://www.tldp.org/LDP/lkmpg/2.4/html/x1210.html 2019-10-10
5. https://www.cambro.umu.se/access/content/group/57250HT19-1/Laboration%203/mish-specification.pdf 2019-10-10
6. https://www.cambro.umu.se/portal/site/57250HT19-1/ 2019-10-10