KTH ICT

Laboration 1

Digenv - processkommunikation med pipes v1.1

ID2206/ID2200 (fd 2G1520/2G1504)
Operativsystem

uppdaterad 2011

KTH ICT SCS

Digenv - processkommunikation med pipes

ID2206/ID2200 (fd 2G1520/2G1504) Operativsystem

1.0 Introduktion

1.1 Mål

Efter laborationen ska du kunna:

- skriva program som skapar processer som kommunicerar via pipes på Unix-liknande system
- skriva C-program som uppfyller rimliga krav på tydlighet, felhantering och dokumentation

1.2 Unix-versioner

Denna laboration kan köras på alla system som följer Posix-standarden, exempelvis Sun Solaris, de flesta Linux-distributioner, Microsoft Windows med *Cygwin* (`www.cygwin.com`). Alla system har dock inte manual-texterna; i så fall finns en användbar webbsida:
`http://www.freebsd.org/cgi/man.cgi`

Det finns två huvudgrenar av Unix: System V (SysV) och Berkeley (BSD). Gnu/Linux-system har ofta valbart SysV-stil eller BSD-stil. Laborationen följer alltså Posix, men ibland behövs olika kod eller olika kommandon för SysV-dialekt och BSD-dialekt. I vårt fall gäller dock att programmet skall gå att kompilera och köra på IMIT's eller NADA's SUN-datorer

2.0 Laborationsuppgift

Uppgiften går ut på att skriva ett litet program/kommando för att lättare kunna studera de environmentvariabler varje shell (kommandotolk) i UNIX har. Environmentvariabler används bland annat för att identifiera ditt användarnamn, vilket directory som är hem-directory och för sökvägar till directoryn som skall sökas igenom då du startar ett nytt program/ger ett kommando.

2.1 Environmentvariabler

I de flesta UNIX kommandotolkar (shell) finns ett inbyggt kommando **`printenv(1)`** som skriver ut processens environmentvariabler på skärmen/i fönstret. `tcsh` och `bash` är två kommandotolkar som har `printenv` inbyggt. Har den kommandotolk du kör inte `printenv` inbyggt finns kommandot ofta att tillgå som separat kommando i något bibliotek, t.ex i GNU distributionen som du normalt hittar under `/opt/gnu/bin/` Som vi också visat på övningarna kan man enkelt själv skriva ett program med motsvarande funktionalitet i C.

Kör du bara kommandot:

```
dator> printenv
```

Så får du en listning av alla environmentvariabler. Ger man kommandot `printenv` *`VARIABEL`* så får du se värdet på variabeln. Till exempel kan du studera vilken sökväg som din kommandotolk använder när den skall hitta ett kommando/program kan du titta på environmentvariabeln PATH med hjälp av kommandot:

```
dator> printenv PATH
```

Environmentvariablerna kan förändras av olika kommandon. Till exempel lägger kommandon som `module add gnu` till directoryn till din sökväg (`PATH`).

Environmentvariablerna ärvs normalt från föräldraprocessen. För ett C-program innebär det att de läggs på stacken och kan accessas från `main()` via en tredje parameter som brukar kallas `envp`, som är en vektor av pekare till textsträngar. Textsträngarna innehåller environmentvariabelnamnet direkt följt av ett likhetstecken som följs av värdet på variabeln. Användaren kan själv definiera nya environmentvariabler via t.ex `set(1)` eller `setenv(1)`.

Studera dina egna environmentvariabler via `printenv(1)` och genom att provköra följande C-program.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv, char **envp)
{
  int i;
  for(i=0; envp[i]!= NULL; i++)
   printf("%2d:%s\n",i, envp[i]);
}
```

Figur 1. Utskrift av environmentvariabler

2.2 Filter

I Unix/Linux använder man ofta användarkommandon som kallas för filter. Utmärkande för ett filter är att de läser från `stdin`, behandlar strömmen av tecken och skriver resultatet på `stdout` om inget annat anges. I enlighet med UNIX designfilosofi så gör oftast ett filter en enkel, väl avgränsad uppgift. Genom att koppla ihop flera filter med hjälp av pipes får man en enkel och flexibel, men ändå kraftfull möjlighet att utföra komplexa uppgifter.

I den här uppgiften skall du använda filtren `grep(1)`, `sort(1)` och `less(1)` (eller `more(1)`)

`grep(1)` används för att söka efter en textsträng eller ett reguljärt uttryck. `grep(1)` söker antingen i indata som läses via `stdin` eller frånen eller flera filer. För t.ex. se var din sökväg, `PATH`, sätts upp kan du prova att köra följande kommando i ditt hemdirectory:

```
dator> grep PATH .*
```

Har du lite otur så ramlar det förbi en massa rader på skärmen som du inte hinner med att läsa. Då kan man lämpligen skicka utdata till en *pager/browser* som fyller en skärmsida i taget. Två vanliga sådana är `less(1)` och `more(1)` (bägge kan avslutas genom att trycka <q>-tangenten). `less(1)` är mer avancerad och tillåter att man rullar tillbaka i texten och att man kan söka i texten.

```
dator> grep PATH .* | less
```

Vilken pager som används som default när du t.ex tittar på en man-sida bestäms av environmentvariabeln `PAGER`.

När man kör `printenv` kommandot utan parametrar får man en lista av alla environmentvariabler i den ordning de har definierats. Vill man bringa lite mer struktur i det hela kan man sortera listan i t.ex alfabetisk ordning genom att köra utmatningen från `printenv` genom ett sorteringsfilter, `sort(1)`.

```
dator> printenv | sort
```

3.0 Uppgiften

Din uppgift är att skriva ett litet program/kommando, **`digenv.c`**, som man kan använda för att enklare studera sina environmentvariabler. Syntaxen för att använda `digenv` skall vara

```
digenv [parameterlista]
```

Det här betyder att man skall kunna köra `digenv` med eller utan en parameterlista. Kör man `digenv` utan parametrar skall det motsvara att köra (om man använder `less(1)` som pager):

```
printenv | sort | less
```

Vilken pager som skall användas skall ditt program kontrollera genom att läsa av environmentvariabeln `PAGER`. Finns ingen sådan variabel skall ditt program använda `less(1)`, finns inte `less(1)` skall du pröva med `more(1)`. Environmentvariabler kan man läsa av från C-program med `getenv(3)`.

Ger man parametrar till `digenv` skall de tolkas som parametrar till `grep(1)` och då skall följande pipe exekveras:

```
printenv | grep parameterlista | sort | less
```

Uppgiften går alltså ut på att kontrollera om man fått några kommandoradsparametrar och därefter koppla upp en pipeline med tre eller fyra steg.

Observera att det inte är tillåtet att använda C-bibliotekets anrop `system(3)` som utnyttjar kommandotolkens (shells) funktionalitet.

3.1 Krav på ditt program

Ditt program `digenv` ska uppfylla följande krav. Du ska ha mycket goda skäl, dokumenterade i kommentarer, om ditt program inte uppfyller alla kraven.

Ditt program ska alltid kontrollera returvärden från systemanrop.

Ditt program ska hantera fall där en fas i pipelineningen misslyckas — till exempel om något syntaxfel uppstår i en fas. Då ska programmet avbrytas på ett snyggt sätt. Använd exit-status från processerna som indikation på om något gått fel. Observera att det är svårt att provocera dessa program att indikera fel (det vill säga lämna någon annan exit-status än 0), enklast sättet att provocera fram fel är att skicka felaktiga flaggor till `grep`.

Ditt program får aldrig lämna kvar oavslutade barn-processer efter en körning. Kontrollera med `ps -el` eller `ps -aux`.

Ditt program bör inte ta bort barn-processer med systemanropet `kill` annat än i undantagsfall, dvs om du upptäcker något allvarligt fel.. Barn-processerna avslutas normalt av sig själva om du gjort rätt med dina pipes.

Ditt program får aldrig lämna kvar temporära datafiler efter en körning. Använd `unlink` för att ta bort ev. sådana filer.

Ditt program skall kunna kompileras med **`gcc -Wall`** utan några varningar alls.

Programmet skall fungera utan parametrar, med flera korrekta parametrar och hantera felaktiga parametrar på ett kontrollerat sätt.

Ditt program ska följa de dokumentationskrav som ges i *CDOK*, se kursens webbsidor. Kraven innebär bland annat att:

- Det ska finnas en längre beskrivande kommentar i början av programmet.
- Delar du upp ditt program i flera filer (moduler) så ska varje fil dessutom ha en längre beskrivande kommentar som förklarar vad den aktuella modulen gör.

- Det ska finnas en längre beskrivande kommentar före varje funktion eller procedur.
- Det ska finnas en kort förklarande kommentar för varje parameter och för varje deklarerad variabel.
- Det bör finnas en förklarande kommentar före en sekvens med programrader som tillsammans gör något icke-trivialt.
- Det bör finnas upplysande och begripliga kommentarer invid programrader som gör något icke-trivialt.

3.2 Rekommendationer och tips

Läs igenom häftet "Användbara systemanrop och biblioteksfunktioner" innan du börjar arbeta med labben (eller försöker svara på förberedelsefrågorna).

Det kan vara lättare om du skriver och testar en fas i taget och lägger till nästa fas när du klarat av de föregående.

Kolla regelbundet att programmet inte lämnar efter sig barn-processer.

Om programmet inte avslutas ordentligt, eller envisas med att lämna efter sig barn-processer, kontrollera mycket noga att du verkligen stänger alla oanvända ändar av alla pipes med `close`.

Kom ihåg att du inte kan använda dubbla snedstreck (`//`) för kommentarer i C. Du måste använda `/*` och `*/` i stället.

Det är inte säkert att ditt program verkligen måste kopiera innehållet i de textsträngar som utgör kommandoradsargumenten.

Elegantaste lösningen som enkelt kan utvidgas till att klara att koppla ihop ett godtyckligt antal filter (processer) är att använda rekursion. För att göra en sådan lösning måste man utnyttja att barnprocesserna påbörjar sin exekvering då de skapas och att föräldraprocessen läggs i tillståndet ready. Dock skall man vara på det klara med att det är konceptuellt enklare, men genererar mer kod, att lösa uppgiften mer rättframt utan rekursion.

3.3 Förberedelsefrågor

1. När en maskin bootar med UNIX skapas en process som har PID=1 och den lever så länge maskinen är uppe. Från den här processen skapas alla andra processer med `fork`. Vad heter denna process?
   Tips: Kommandot `ps -el` (SysV) eller `ps -aux` (BSD) ger en lista med mycket information om alla processer i systemet.
2. Kan environmentvariabler användas för att kommunicera mellan föräldra- och barnprocess? Åt bägge hållen?
3. Man kan tänka sig att skapa en odödlig child-process som fångar alla `SIGKILL`-signaler genom att registrera en egen signalhanterare kill_handler som bara struntar i `SIGKILL`. Processen ska förstås ligga i en oändlig loop då den inte exekverar signalhanteraren. Testa! Skriv ett programmet med en sådan signalhanterare, kompilera och provkör. Vad händer? Läs mer i manualtexten om `sigaction` för att förklara resultatet.
4. Varför returnerar `fork` 0 till child-processen och child-PID till parent-processen, i stället för tvärtom?
5. UNIX håller flera nivåer av tabeller för öppna filer, både en användarspecifik "File Descriptor Table" och en global "File Table". Behövs egentligen File Table? Kan man ha offset i File Descriptor Table istället?
6. Kan man strunta i att stänga en pipe om man inte använder den? Hur skulle programbeteendet påverkas? Testa själv. Läs mer i `pipe(2)`.

7. Vad händer om en av processerna plötsligt dör? Kan den andra processen upptäcka detta?
8. Hur kan du i ditt program ta reda på om `grep` misslyckades? Dvs om grep inte hittade någon förekomst av det den skulle söka efter eller om du gett felaktiga parametrar till `grep`?

4.0 Om rapporten

Rapporten skall vara utformad i enlighet med *"Allmänna instruktioner för LabPM"* som du hittar på kursens hemsida. Speciellt bör du tänka på att ha med:

- Ifyllt försättsblad
- En relativt utförlig beskrivning av uppgiften, hur du valt att lösa den och ditt program
- Svar på *samtliga* förberedelsefrågor från detta labPM
- Resultat från testkörningar
- Väl kommenterad källkod
- Uppgift om var källkoden finns och hur den skall kompileras (se till att vi kan accessa den!)

För att vi skall kunna förbättra labben och labbkursen vill vi också att du ägnar lite tid åt att också skriva in följande i din labbrapport:

- En uppskattning på hur mycket tid du lagt ned på att göra labben
- Betygssätt labPM, svårighetsgrad på labben och vad du fått ut av den på en skala 1-5
- Kan vi förbättra labben? Kom gärna med konstruktiv kritik!