Framsida för labrapport

Operativsystem, ID220 6

Period 2, läsår 20 13

Fyll i alla uppgifter!

Labnr.	Labnamn
1	Digenv - processkommunikation med pipes

Efternamn, förnamn	Personnummer	*Tydlig* datorpostadress
Cederlund, Mattias	920926-2410	mcede@kth.se

Inlämningsdatum: 2013-12-02

Kommentarer

För internt bruk

Godkänd	Komplettera	Meddelad	Datum	Signatur

Registrerad	Ny	Gammal	G/B/U

Problembeskrivning

Uppgiften bestod i att skriva ett program `digenv` för att lättare studera environmentvariabler.

För att visa environmentvariablerna kan man ta hjälp av anropet `printenv`. För att strukturera upp detta ytterligare kan man använda sig av `grep` för att plocka ut endast de rader man är intresserade av. `sort` kan användas för att sortera raderna i bokstavsordning och en pager såsom `less` eller `more` kan användas för att smidigt kunna läsa outputen så att den inte hinner hamna utanför skärmen. Vilken pager som ska användas kan man kontrollera genom att läsa av environmentvariabeln PAGER. Om denna inte finns ska programmet försöka använda `less`, och om inte den finns `more`. För att länka ihop programmen ska processerna kommunicera med hjälp av pipes.

Programmet ska kunna köras med eller utan parametrar och de parametrar man skickar in ska tolkas som parametrar till grep.
Om inga parametrar skickas till `digenv` ska följande pipeline exekveras:

```
printenv | sort | less
```

Om parametrar skickas till `digenv` ska istället följande exekveras:

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

Programbeskrivning

Programmet implementerades rättframt utan rekursion med en tre eller fyra-stegs pipeline beroende på om grep skulle användas eller inte.

När programmet startar undersöker den vilken pager som ska användas genom getenv(”PAGER”). Returnerar den null finns det ingen pager specificerad i environmentvariablerna och då används `less`. I pipeline-steget där pagern ska exekveras körs `more` om less misslyckades. Dessutom sätts en korrekt parameterlista till `grep` ihop ifall användaren skickat in parametrar till `digenv`.

Därefter skapas den första pipen, den första childprocessen forkas och sätts att exekvera `printenv`. Självklart dupliceras pipens fildeskriptorerer och pipens ändar stängs i alla childprocesser. Returvärden kontrolleras också från alla systemanrop.

Jag har valt att implementera min lösning på så sätt att parentprocessen väntar på att varje childprocess exekverat klart innan nästa startas. På så sätt förenklas felhantering och man kan enkelt avbryta programmet och ge användaren ett vettigt felmeddelande om någon process returnerat med felaktigt resultat.

När första childprocessen exekverat klart stängs de pipes som inte längre behövs för nästa steg i pipelinen och nya pipes skapas innan nya childprocesser forkas. Programmet fungerar på beskrivet sätt i alla 3-4 steg av pipelinen.

De enda funktionerna som jag använder är `close_pipes(int * pipe)` som stänger båda ändar av given pipe samt hanterar felmeddelanden och `wait_for_child()` som väntar på att någon childprocess avslutar och kontrollerar om den avslutats med någon felkod. Är så fallet hanterar den det och visar felmeddelanden. Dessa funktioner är till för att slippa upprepa kod.

De algoritmiska skillnader man hade kunnat implementera är att göra programmet rekursivt men eftersom målet är att skriva program som skapar processer som kommunicerar via pipes valdes en mer rättfram lösning.

Förberedelsefrågor

1. Den första processen som skapas heter init.
2. Med getenv(3) och setenv(3) så kan båda processer sätta och läsa environmentvariabler. Dock är de lokala för den process de blir satta i. Childprocesser ärver parentprocessens environmentvariabler och därmed går det endast att kommunicera från parent till child och inte tvärt om.
3. Sigaction gäller för alla signaler utom SIGKILL och SIGSTOP, därför fungerar det inte att ha en sådan process.
4. För att parent-processen kan vilja "styra" sina childprocesser och det är inte möjligt om den inte vet pid till dem.
5. Jag ser inga problem med det annat än att OSet tappar viss kontroll över filsystemet vilket kan tänkas problematiskt om t.ex. flera processer vill åt att skriva i samma fil.
6. Parentprocessen väntar för evigt på att childprocessen avlsutas, men det kommer inte ske eftersom den inte ser EOF.
7. Parentprocessen kan kolla exit-status på sina childprocesser genom wait och WIFEXITED. På så sätt ser den om någon child-process dött.
8. Du kan till wait skicka med en pekare för att spara ner returvärden från child-processerna. Från denna kan man sedan med WIFEXITED ta reda på vilket värde childprocessen avslutades med. För grep så gäller att exitstatus är 0 om man hittat lines med eftersökt värde och 1 annars. Errors ger exitstatus 2.

Resultat från testkörningar

Körning utan argument på shell.it.kth.se:

```
[mcede@avril lab1]$ ./digenv
}
CVS_RSH=ssh
_=./digenv
EDITOR=emacs
EMAIL=mcede@kth.se
G_BROKEN_FILENAMES=1
GCONF_LOCAL_LOCKS=1
HISTSIZE=1000
HOME=/afs/ict.kth.se/home/m/c/mcede
HOSTNAME=avril.it.kth.se
INFOPATH=/usr/share/info
INPUTRC=/etc/inputrc
KDEDIR=/usr
KDE_IS_PRELINKED=1
KDE_NO_IPV6=1
KRB5CCNAME=FILE:/tmp/krb5cc_1010014_HaB5S0
LANG=en_GB.ISO-8859-15
LESSOPEN=|/usr/bin/lesspipe.sh %s
_LMFILES_=/etc/site/modulefiles/Default:/usr/local/vol/modulefiles
/fvwm2-
local/current:/usr/local/vol/modulefiles/matlab/7.4.0:/usr/local/v
ol/modulefiles/maple/10.05:/usr/local/vol/modulefiles/comsol/3.4
LOADEDMODULES=Default:fvwm2-
local/current:matlab/7.4.0:maple/10.05:comsol/3.4
LOGNAME=mcede
:
```

Körning med argument på shell.it.kth.se:
[mcede@avril lab1]$ ./digenv PATH

```
INFOPATH=/usr/share/info
MANPATH=/usr/local/vol/matlab/7.4.0/man:/usr/kerberos/man:/usr/loc
al/share/man:/usr/share/man/en:/usr/share/man:/usr/man
MODULEPATH=/usr/local/vol/modulefiles:/pkg/modules/modulefiles:/et
c/site/modulefiles:/usr/share/Modules/modulefiles:/etc/modulefiles
PATH=/usr/lib/heimdal/bin:/usr/local/vol/comsol/3.4/bin:/usr/local
/vol/maple/10.05/bin:/usr/local/vol/matlab/7.4.0/bin:/usr/kerberos
/bin:/usr/local/bin:/bin:/usr/bin:/opt/real/RealPlayer
WMMENUPATH=/usr/local/vol/maple/menudef/10.05:/usr/local/vol/fvwm2
/menudefs/emacs:/usr/local/vol/fvwm2/menudefs/acroread
(END)
```

Väl kommenterad kod

```
/*
 *
 * NAME:
 *   digenv - a program for easily showing environment variables
 *
 * SYNTAX:
 *   digenv [grep arguments]
 *
 * DESCRIPTION:
 *    Digenv can be run with or without arguments. All arguments
 *    will be passed on to grep. Passing bad arguments to digenv
 *    will cause grep to terminate with an error and by that way
 *    terminate digenv with an error.
 *
 *    If run without arguments, digenv will execute the following:
 *    printenv | sort | less
 *
 *    If run with arguments, digenv will execute the following:
 *    printenv | grep [arguments] | sort | less
 *
 * EXAMPLES:
 *  digenv
 *    digenv PATH
 *
 * SEE ALSO:
 *   printenv(1), grep(1), sort(1), less(1)
 *
 */

#include <sys/types.h> /*definierar typen pid_t*/
#include <sys/wait.h> /*definierar bland annat WIFEXITED*/
#include <errno.h> /*definierar errno*/
#include <stdio.h> /*definierar bland annat stderr*/
#include <stdlib.h> /*definierar bland annat rand() och RAND_MAX*/
#include <unistd.h> /*definierar bland annat pipe() och STDIN_FILENO*/

#define PIPE_READ_SIDE (0)
#define PIPE_WRITE_SIDE (1)
#define DEBUG (0)

pid_t childpid; /*för child-processens PID vid fork()*/
pid_t grep_pid; /*För att spara greps pid, för att skriva vettigt errormsg om
fel där*/
```

```
/*close_pipes
 *
 *close_pipes closes both ends of a given pipe
 *
 *int * pipe - the filedescriptor for the pipe to close
 *
 */
void close_pipes(int * pipe) {
      int return_value;

      /*Stäng pipens läs-sida och kolla output från anropet*/
      return_value = close(pipe[ PIPE_READ_SIDE ]);
      if (return_value == -1) {
            perror("Cannot close read end");
            exit(1);
      }

      /*Stäng pipens skriv-sida och kolla output från anropet*/
      return_value = close(pipe[PIPE_WRITE_SIDE]);
      if (return_value == -1) {
            perror("Cannot close write end");
            exit(1);
      }
}

/*wait_for_child
 *
 *close_pipes waits for a child process to terminate
 *If an error occurs wait_for_child will print information to STDERR
 *
 */
void wait_for_child() {
      int status; /*för returvärden från child-processer*/

      childpid = wait(&status);
      if (DEBUG) {
            fprintf(stderr, "Process (pid %ld) exited\n", (long int) childpid);
      }
      if (childpid == -1) {
            perror("wait() failed unexpectedly");
            exit(1);
      }

      /*Child-processen har kört klart*/
      if (WIFEXITED(status)) {
            int child_status = WEXITSTATUS(status);
            /*Om child-processen hade problem*/
            if (child_status != 0) {
                  fprintf(stderr, "Child (pid %ld) failed with exit code %d\n",
                              (long int) childpid, child_status);
                  if (childpid == grep_pid) {
                        fprintf(stderr, "Grep could not find anything with that
input!\n");
                  }
                  exit(0);
            }
      }
      else {
            /*Om child-processen avbröts av signal*/
            if (WIFSIGNALED(status)) {
                  int child_signal = WTERMSIG(status);
                  fprintf(stderr, "Child (pid %ld) was terminated by signal no.
%d\n",
                              (long int) childpid, child_signal);
```

```
            }
      }
}

int main(int argc, char **argv, char **envp) {
      int pipe_getenv_grep[2]; /*för fildeskriptorer från pipe(2)*/
      int pipe_getenv_sort[2];
      int pipe_grep_sort[2];
      int pipe_sort_less[2];
      int return_value; /*för returvärden från systemanrop*/
      int use_grep = 0; /*Flagga som sätts till 1 om vi ska använda grep*/
      char* grep_args[argc+1]; /*Behållare för argumenten till grep*/
      char* pager; /*Behållare för pager*/

      /*Läs pager från environment-variablerna. Om Pager inte finns, använd
less*/
      pager = getenv("PAGER");
      if (pager == NULL) {
            pager = "less";
      }

      /*Om det finns parametrar till grep*/
      if (argc > 1) {
            use_grep = 1;                              /*Sätt flaggan för att
använda grep*/
            int i;
            grep_args[0] = "grep";              /*Första argumentet är namnet på
programmet som ska köras*/
            for (i = 1; i < argc; i++) {
                  grep_args[i] = argv[i];       /*Kopiera över argumenten*/
            }
            grep_args[argc] = (char *) 0;       /*Avsluta argument-arrayen med
nollställd byte*/
      }

      /*Skapa pipe beroende på om grep ska köras*/
      if (use_grep) {
            return_value = pipe(pipe_getenv_grep); /*Skapa en pipe*/
      }
      else {
            return_value = pipe(pipe_getenv_sort);
      }

      if (return_value == -1) { /*Om pipe() misslyckades*/
            perror("Cannot create pipe");
            exit(1);
      }

      childpid = fork();      /*Skapa ny process med fork*/
      if (DEBUG) {
            fprintf(stderr, "PRINTENV forked (pid %ld)\n", (long int) childpid);
      }
      /*Kod som körs av child-processen*/
      if (0 == childpid) {
            if (use_grep) {
                  /*Dupicera fildeskriptor så att processen skriver till
pipe_getenv_grep istället för stdout, kolla returvärde*/
                  return_value = dup2(pipe_getenv_grep[ PIPE_WRITE_SIDE ],
STDOUT_FILENO);
                  if (return_value == -1) {
                        perror("Cannot dup printenv_grep write");
                        exit(1);
                  }
                  close_pipes(pipe_getenv_grep);      /*Stäng pipens ändar*/
```

```
            }
            else {
                  /*Om vi inte ska använda grep - duplicera annan pipe, kolla
returvärde*/
                  return_value = dup2(pipe_getenv_sort[ PIPE_WRITE_SIDE ],
STDOUT_FILENO);
                  if (return_value == -1) {
                        perror("Cannot dup printenv_sort write");
                        exit(1);
                  }

                  close_pipes(pipe_getenv_sort);
            }
            (void) execlp("printenv", "printenv", (char *) 0);   /*Exekvera
printenv*/
            perror("Cannot exec printenv");
            exit(1);

      }
      /*Om fork misslyckades*/
      if (childpid == -1) {
            perror("Cannot fork()");
            exit(1);
      }

      wait_for_child();       /*Vänta på att printenv blir klar*/

      /*Körs endast om vi har angett parametrar till grep*/
      if (use_grep) {
            return_value = pipe(pipe_grep_sort);

            if (return_value == -1) {
                  perror("Cannot create pipe");
                  exit(1);
            }

            childpid = fork();
            if (DEBUG) {
                  fprintf(stderr, "GREP forked (pid %ld)\n", (long int)
childpid);
            }
            if (0 == childpid) {
                  /*Dupicera fildeskriptor så att processen läser från
pipe_getenv_grep istället för stdin*/
                  return_value = dup2(pipe_getenv_grep[ PIPE_READ_SIDE ],
STDIN_FILENO);
                  if (return_value == -1) {
                        perror("Cannot dup getenv_grep read");
                        exit(1);
                  }

                  close_pipes(pipe_getenv_grep);
                  /*Dupicera fildeskriptor så att processen skriver till
pipe_grep_sort istället för stdout*/
                  return_value = dup2(pipe_grep_sort[ PIPE_WRITE_SIDE ],
STDOUT_FILENO);
                  if (return_value == -1) {
                        perror("Cannot dup grep_sort write");
                        exit(1);
                  }

                  close_pipes(pipe_grep_sort);
                  (void) execvp("grep", grep_args);  /*Exekvera grep med
argumenten som lästes in som argument*/
```

```
                perror("Cannot exec grep");
                exit(1);
          }
          else {
                grep_pid = childpid;
          }

          if (childpid == -1) {
                perror("Cannot fork()");
                exit(1);
          }
          close_pipes(pipe_getenv_grep);
    }

    /*Vänta på grep endast om vi använde grep*/
    if (use_grep) {
          wait_for_child();
    }

    return_value = pipe(pipe_sort_less);  /*Skapa pipe för att använda mellan
sort och less*/

    if (return_value == -1) {
          perror("Cannot create pipe");
          exit(1);
    }

    childpid = fork();
    if (DEBUG) {
          fprintf(stderr, "SORT forked (pid %ld)\n", (long int) childpid);
    }
    if (0 == childpid) {
          /*Använd olika pipes för läsning beroende på om vi använder grep
eller inte*/
          if (use_grep) {
                return_value = dup2(pipe_grep_sort[ PIPE_READ_SIDE ],
STDIN_FILENO);
                if (return_value == -1) {
                      perror("Cannot dup grep_sort read");
                      exit(1);
                }

                close_pipes(pipe_grep_sort);
          }
          else {
                return_value = dup2(pipe_getenv_sort[ PIPE_READ_SIDE ],
STDIN_FILENO);
                if (return_value == -1) {
                      perror("Cannot dup getenv_sort read");
                      exit(1);
                }

                close_pipes(pipe_getenv_sort);
          }

          return_value = dup2(pipe_sort_less[ PIPE_WRITE_SIDE ],
STDOUT_FILENO);
          if (return_value == -1) {
                perror("Cannot dup sort_less write");
                exit(1);
          }

          close_pipes(pipe_sort_less);
```

```
            (void) execlp("sort", "sort", (char *) 0);

            perror("Cannot exec sort");
            exit(1);

      }

      if (childpid == -1) {
            perror("Cannot fork()");
            exit(1);
      }

      /*Stäng de pipes vi är klara med, beroende om vi använt grep eller inte*/
      if (use_grep) {
            close_pipes(pipe_grep_sort);
      }
      else {
            close_pipes(pipe_getenv_sort);
      }

      wait_for_child(); /*Vänta på sort*/

      childpid = fork();
      if (DEBUG) {
            fprintf(stderr, "LESS forked (pid %ld)\n", (long int) childpid);
      }
      if (0 == childpid) {
            return_value = dup2(pipe_sort_less[ PIPE_READ_SIDE ], STDIN_FILENO);
            if (return_value == -1) {
                  perror("Cannot dup sort_less read");
                  exit(1);
            }

            close_pipes(pipe_sort_less);

            (void) execlp(pager, pager, (char *) 0);

            perror("Cannot exec pager, will retry with more");

            (void) execlp("more", "more", (char *) 0);      /*Om vald pager
misslyckades - exekvera more istället*/

            perror("Cannot exec more");
            exit(1);

      }

      if (childpid == -1) {
            perror("Cannot fork()");
            exit(1);
      }

      close_pipes(pipe_sort_less); /*Stäng pipen mellan sort och less*/

      wait_for_child(); /*Vänta på less*/

      exit(0);  /*Avsluta parent-processen på normalt sätt*/
}
```

Var källkoden finns

Källkoden finns på /afs/ict.kth.se/home/m/c/mcede/os/lab1 där programmet heter digenv.c.

Som backup finns även koden tillgänglig på
https://www.dropbox.com/s/vvvvoyovkxhxgu3/digenv.c

Koden kompileras med gcc -W -o digenv digenv.c.

Verksamhetsberättelse och utvärdering av laborationsuppgiften

Till en början utgick jag från exemplet duptest.c givet i dokumentet Användbara Systemanrop bestående av ett enkelt exempel där processer kommunicerar via pipes. Utifrån den byggde jag upp fler steg i pipelinen tills att alla delprocesser kunde köras. Det blev såklart like problem med att stänga ändarna vid rätt tillfälle men till slut löste det sig. Då implementerades parameterhantering och val av pager i nästa steg och till sist såg jag till att felhanteringen uppfyllde kraven.

Uppskattad tid lagd på laborationen: 10 timmar
Betyg för labPM: 4
Svårighetsgrad för laborationen: 3
Vad som fåtts ut av laborationen: 4
Vad som kan förbättras: Jag hade gärna sett att större vikt hade lagts på dokumentet ”Användbara systemanrop”. I nuvarande labPM nämns det under tips. För att tydliggöra anser jag att den snarare ses som en förberedelse, liggande först i labPM. I övrigt välskrivet och lättförstått labPM.