

ID2200 – Laboration 1: Digenv

Sam Henriksson
hensa@kth.se
910530-2773

Elvis Stansvik
stansvik@kth.se
831205-3971

23 april 2014

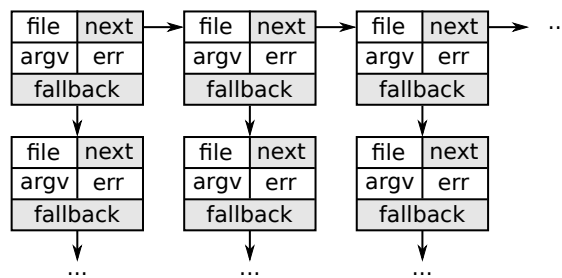
Innehåll

1	Lösning	2
2	Testkörningar	3
3	Förberedelsefrågor	3
4	Övrigt / Utvärdering	6
4.1	Tidsuppskattning	6
4.2	Betygssättning av lab-PM	6
4.3	Förslag på förbättringar	6
	Appendix A: Källkod	7
	Appendix B: Körningsutskrifter	10

1 Lösning

Uppgiften bestod i att skriva ett program `digenv` som simulerar UNIX-pipelinen `printenv | sort | less` eller, om en parameterlista angivits, `printenv | grep [parameterlista] | sort | less`. Som pager, det sista kommandot i pipelinen, ska först miljövariabeln `PAGER` testas, sedan `less` och sist `more`.

För att lösa uppgiften byggdes först en länkad datastruktur `command_t` för att hålla de kommandon som ska köras. Varje post i listan beskriver ett kommando (`file`), dess parametrar (`argv`), en felkod från senaste försöket att köra kommandot (`err`), nästa kommando i listan (`next`) samt ett eventuellt fallbackkommando (`fallback`) som kan ersätta kommandot om det saknas. Datastrukturens utseende visas i figur 1 nedan.



Figur 1: Datastrukturen `command_t`. Pekarfält indikeras med grått.

Sedan skrevs en funktion `run_pipeline` som tar en kommandolista (`command_t*`) och en fildeskriptor att läsa ifrån som argument, och sedan kör pipelinen som beskrivs av listan. Funktionen anropar först `pipe(2)` för att sätta upp två länkade fildeskriptorer för kommunikationen mellan första kommandot och nästa. Sedan anropas `fork(2)` för att skapa upp en ny barnprocess. I barnprocessen dirigeras standard input till fildeskriptorn för läsning som gavs som argument och standard output till pipens skrivdeskriptor med hjälp av `dup2(3)`. Slutligen exekveras kommandot i barnprocessen med hjälp av `execvp(3)`.

Om exekveringen lyckades, dvs barnprocessen avslutades med 0 som exit status, anropar funktionen `run_pipeline` sig själv med resten av kommandolistan och pipens läsdeskriptor som argument, för att sätta upp resten av pipelinen.

När exekveringen av kommandot sker i barnprocessen testas först det givna kommandot. Om det misslyckas testas sedan i tur och ordning fallbackkommandona som beskrivs av fallbackpekarna, tills något av dem lyckas. Misslyckas alla skrivs ett felmeddelande ut och rekursionen avbryts.

Efter att `run_pipeline` skrivits återstod bara att konstruera en lista som beskriver

```
printenv | sort | $PAGER -> less -> more
```

eller

```
printenv | grep [parameterlista] | sort | $PAGER -> less -> more
```

och anropa `run_pipeline` på denna.

Koden för programmet finns listad i Appendix A. Den finns också att hämta på <https://github.com/estan/ID2200> och kompileras på ett UNIX-system genom att skriva `make`.

2 Testkörningar

Programmet testades genom att köra `digenv` och `digenv HOME` med

- både `less` och `more` i `PATH` och `PAGER` satt till en tredje pager (`most`),
- både `less` och `more` i `PATH` men `PAGER` ej satt,
- bara `more` i `PATH`, men ej `less` och `PAGER` ej satt,
- varken `less` eller `more` i `PATH` och `PAGER` ej satt,
- varken `less` eller `more` i `PATH` och `PAGER` satt till en ej existerande fil (`foo`),
- felaktig parameter (`-y`) given till `grep`,

och visade sig i samtliga fall fungera som förväntat. Utskrifter från samtliga `digenv`-körningar återfinns i Appendix B. Körningarna av `digenv HOME` gav snarlikt resultat, men visade som förväntat bara miljövariabler som matchade "HOME".

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.*

Svar: `init` (alternativt `systemd` eller `upstart` på nyare Linux-system).

2. Kan *environment*-variabler användas för att kommunicera mellan föräldra- och barnprocess? Åt bägge hållen?

Svar: Miljövariabler satta i föräldraprocessen är synliga i barnprocessen och föräldraprocessen kan således kommunicera till barnprocessen genom att sätta miljövariabler. Det omvända är inte möjligt: de miljövariabler barnprocessen sätter är endast synliga i barnprocessen själv och dess egna barnprocesser, inte i föräldraprocessen. Om barnprocessen gör anrop till funktioner ur `exec`-familjen så måste dessutom miljövariablerna ha satts innan anropet, eftersom de flesta av dessa funktioner skapar kopior av den aktuella miljön – miljövariabler som satts i föräldraprocessen efter `exec`-anropet propagerar ej till programmet som startats av `exec`.

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.

Svar: Enligt specifikationen så tillåts man inte hantera signalerna **SIGKILL** eller **SIGSTOP**, eller ignorera dem med **SIG_IGN**. Faktum är att enligt dokumentationen i **sigaction(3P)** så ska varje försök att göra detta tyst ignoreras, men vi testade följande program på Linux och anropet till **sigaction** misslyckas högljutt genom att returnera -1 och sätta **errno** till **EINVAL**.

Källkod 1: `killtest.c`

```
1 #include <signal.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5
6 static void handler(int signum) { }
7
8 int main(int argc, char *argv[]) {
9     struct sigaction sa;
10    sa.sa_handler = handler;
11    sigemptyset(&sa.sa_mask);
12
13    /* Try to handle SIGKILL */
14    if (sigaction(SIGKILL, &sa, NULL) == -1) {
15        perror("sigaction");
16        exit(EXIT_FAILURE);
17    }
18
19    while (1) {
20        sleep(2);
21    }
22 }
```

```
$ ./killtest
sigaction: Invalid argument
```

4. Varför returnerar **fork** 0 till child-processen och child-PID till parent-processen, i stället för tvärtom?

Svar: För att man oftast är intresserad av att i föräldraprocessen ha tillgång till barnprocessens PID, till exempel för att vänta på att den ska avslutas med **waitpid(2)** eller skicka signaler till den med **kill(2)**. Barnprocessen kan fortfarande enkelt få tag i sitt eget och föräldraprocessens PID med **getpid(3P)** respektive **getppid(3P)** om den skulle behöva dessa.

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?*

Svar: En barnprocess ärver en kopia av sin förälders *File Descriptor Table*. Om den globala *File Table* slopades, och positionerna in i de öppna filerna istället lagrades i den process-specifika *File Descriptor Table* skulle det få som effekt att skrivningar och läsningar i barnprocessen inte förflyttar positionerna i föräldraprocessen och vice versa, vilket skulle vara lite förvirrande. Till exempel vill man kanske starta upp en uppsättning barnprocesser som turas om att skriva till samma fil, vilket skulle bli väldigt komplicerat när kärnan inte längre centralt håller reda på skriv-/läspositionerna in i den öppna filen.

Om filen öppnas på nytt, ifrån samma process eller ifrån en annan, så skapar kärnan dock en ny post i den centrala *File Table*. Positionen är på så sätt kopplad till `open(2)`-anropet.

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)`.*

Svar: Om pipens skrivdeskriptor inte stängs kommer läsaren aldrig se EOF, utan stå och vänta på mer data. Vi testade beteendet genom att kommentera ut `close(2)`-anropet som görs på rad 103 i `pipe.c` (källkod 4 i Appendix A) i vår lösning – programmet tog sig då aldrig vidare utan stod och väntade i första steget i pipelinen.

7. *Vad händer om en av processerna plötsligt dör? Kan den andra processen upptäcka detta?*

Svar: Föräldraprocessen meddelas via signalen SIGCHLD när en barnprocess stoppas eller terminerar. Åt andra hållet är svårare: Barnprocessen skulle vid uppstart kunna spara undan föräldraprocessens PID och sedan regelbundet skicka en signal till denna med hjälp av `kill(2)`. Returvärde 0 från `kill(2)` skulle då indikera att föräldraprocessen fortfarande är igång. Alternativt kan man på Linux-system registrera en signal som ska skickas när föräldraprocessen dör genom att anropa `prctl(2)` med `PR_SET_PDEATHSIG`, men detta är inte en standardmekanism på andra UNIX-system.

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`?*

Svar: `grep` lämnar 0 som exit status om en förekomst hittades och icke-0 om inget hittades eller felaktiga flaggor gavs. Koden i vår lösning låter därför föräldraprocessen vänta på att barnprocessen avslutas med `wait(2)` och inspekterar därefter dess exit status med makrot `WEXITSTATUS`. Detta görs på rad 111–112 i `pipe.c` (källkod 4 i Appendix A).

4 Övrigt / Utvärdering

4.1 Tidsuppskattning

Vi uppskattar att vi jobbat cirka 3-4 timmar med förberedelse av labben och rapportskrivning, och 3-4 timmar med att skriva och testa koden. Så totalt cirka 6-8 timmar nedlagt arbete.

4.2 Betygssättning av lab-PM

Vi ger lab-PM en 4:a på en skala 1–5. Instruktionerna är detaljerade och har många bra tips. Det som kan förbättras är en del svengelska som “environmentvariabler” (“miljövariabler” är nog mer vedertaget) och en del korrekturläsning, men i övrigt är lab-PM bra.

4.3 Förslag på förbättringar

Vi tycker inte riktigt det är realistiskt att skriva `digenv` som ett C-program. Som ingenjörer skulle vi skrivit `digenv` som ett enkelt shellskript istället. För att öka labbens realism föreslår vi att uppgiften ändras så att `digenv` ska kunna sätta samman godtyckliga pipelines istället. Den uppgiften är mer realistisk eftersom det är ett problem vi skulle kunna ställas inför om vi skulle skriva vårt eget kommandoskal.

Appendix A: Källkod

Källkod 2: digenv.c

```
1 #include "pipe.h"
2
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 int main(int argc, char *argv[]) {
7     char *pager_env = getenv("PAGER");
8
9     /* Create pipeline      file      argv      err  next  fallback */
10    command_t more      = {"more",      (char *[]){ "more", NULL},    0, NULL,  NULL};
11    command_t less      = {"less",      (char *[]){ "less", NULL},    0, NULL,  &more};
12    command_t pager     = {pager_env,   (char *[]){ pager_env, NULL},  0, NULL,  &less};
13    command_t sort      = {"sort",      (char *[]){ "sort", NULL},    0, &pager, NULL};
14    command_t grep      = {"grep",      argv,                      0, &sort,  NULL};
15    command_t printenv   = {"printenv", (char *[]){ "printenv", NULL}, 0, &sort,  NULL};
16
17    argv[0] = "grep"; /* Since we reuse argv for grep */
18
19    if (argc > 1) {
20        /* Arguments given: Run grep as well */
21        printenv.next = &grep;
22    }
23
24    /* Run pipeline */
25    run_pipeline(&printenv, STDIN_FILENO);
26
27    return 0;
28 }
```

Källkod 3: pipe.h

```
1 /**
2  * Pipeline functions / data structures.
3  */
4 #ifndef PIPE_H
5 #define PIPE_H
6
7 /**
8  * Represents a command in a pipeline.
9  *
10 * Commands are linked together to form a pipeline using the next field.
11 * A command can have a fallback command specified in the fallback field.
12 * The fallback command will be executed if execution of the command
13 * fails.
14 */
15 typedef struct command_s {
16     const char *file;          /* File to execute. */
17     char * const *argv;        /* NULL-terminated array of arguments. */
18     int err;                   /* Error from last execution attempt. */
19     struct command_s *next;    /* Next command in pipeline. */
20     struct command_s *fallback; /* Fallback command. */
21 } command_t;
22
23 void run_pipeline(command_t *command, int in);
24
25 #endif /* PIPE_H */
```



```

1  #include "pipe.h"
2
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7  #include <stdio.h>
8  #include <errno.h>
9  #include <string.h>
10
11 /**
12  * Turns the file descriptor new into a copy of old.
13  *
14  * If the redirection succeeds, old will be closed. If old and new is the same file
15  * descriptor, this function is a no-op. If redirection fails, the function prints
16  * an error and exits the process with EXIT_FAILURE.
17  */
18 void copy_fd(int old, int new) {
19     if (new != old) {
20         if (dup2(old, new) == -1) {
21             perror("dup2");
22             _exit(EXIT_FAILURE);
23         } else if (close(old)) {
24             perror("close");
25             _exit(EXIT_FAILURE);
26         }
27     }
28 }
29
30 /**
31  * Invokes a command.
32  *
33  * This function will try to run the given command using execvp(). If it fails,
34  * the fallback command is tried instead. If all fallbacks fails, the function
35  * prints an error to stderr and exits the process with EXIT_FAILURE.
36  *
37  * Commands with a NULL file field are ignored.
38  */
39 void invoke(command_t *command) {
40     /* Start with the given command, then successively try each fallback. */
41     command_t *current = command;
42     while (current) {
43         if (current->file) {
44             execvp(current->file, current->argv);
45             current->err = errno;
46         }
47         current = current->fallback;
48     }
49
50     /* All fallbacks failed. Print error and exit. */
51     fprintf(stderr, "Command failed, tried:\n");
52     current = command;
53     while (current) {
54         if (current->file) {
55             fprintf(stderr, "  %s: %s\n", current->file, strerror(current->err));
56         }
57         current = current->fallback;
58     }
59     exit(EXIT_FAILURE);
60 }

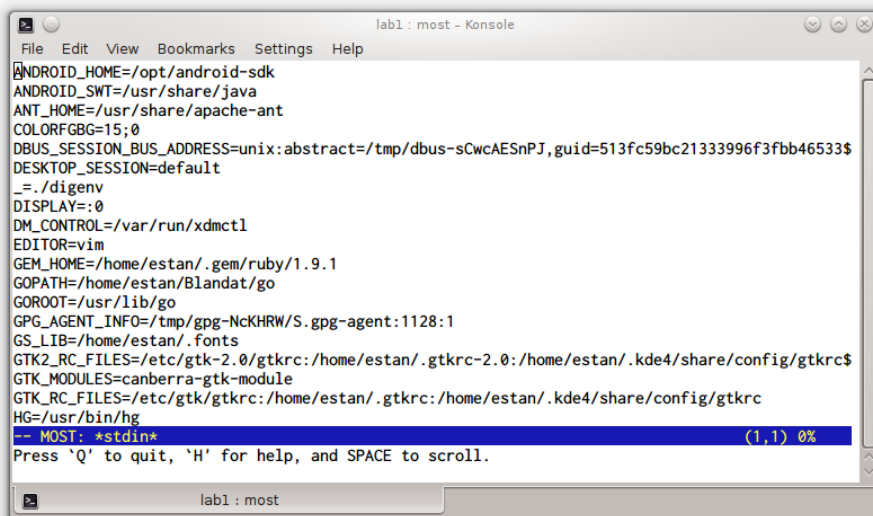
```

```

61
62 /**
63  * Runs a pipeline starting with the given command and input file descriptor.
64  *
65  * This function will construct a new pipe for communication between the given
66  * command and the next, fork a new child process in which the command is executed,
67  * then call itself recursively to set up the rest of the pipeline. Input for
68  * the pipeline is taken from the given file descriptor. If at some point an error
69  * is encountered, an error message is printed and the recursion stops.
70  */
71 void run_pipeline(command_t *command, int in) {
72     int fd[2];
73     pid_t pid;
74     int status;
75
76     if (command->next == NULL) {
77         /* Last command, stop forking. */
78         copy_fd(in, STDIN_FILENO);
79         invoke(command);
80     } else {
81         /* Create pipe. */
82         if (pipe(fd) == -1) {
83             perror("pipe");
84             exit(EXIT_FAILURE);
85         }
86
87         /* Fork a child process. */
88         pid = fork();
89         if (pid == -1) {
90             perror("fork");
91             exit(EXIT_FAILURE);
92         } else if (pid == 0) {
93             /* Running in child. */
94             if (close(fd[0])) {
95                 perror("close");
96                 _exit(EXIT_FAILURE);
97             }
98             copy_fd(in, STDIN_FILENO);
99             copy_fd(fd[1], STDOUT_FILENO);
100             invoke(command);
101         } else {
102             /* Running in parent. */
103             if (close(fd[1])) {
104                 perror("close");
105                 exit(EXIT_FAILURE);
106             }
107             if (close(in)) {
108                 perror("close");
109                 exit(EXIT_FAILURE);
110             }
111             wait(&status);
112             if (!WEXITSTATUS(status)) {
113                 /* Child exited normally, run rest of pipeline */
114                 run_pipeline(command->next, fd[0]);
115             }
116         }
117     }
118 }

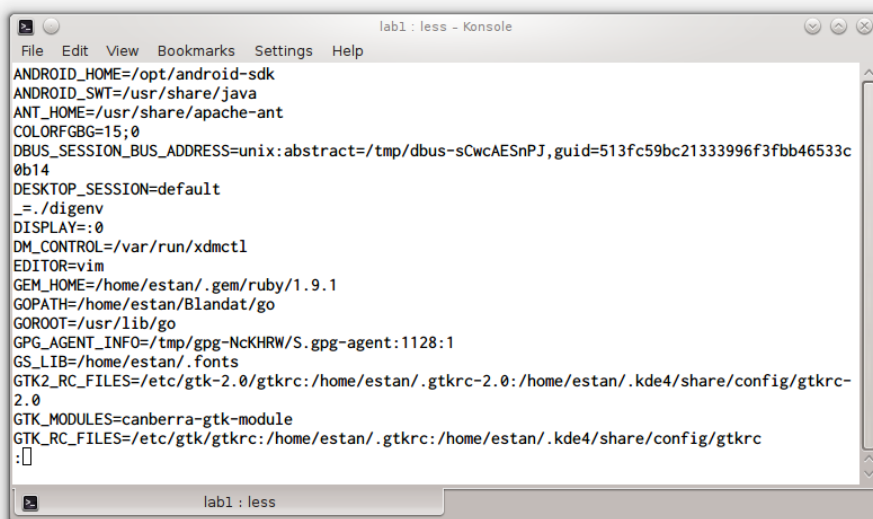
```

Appendix B: Körningsutskrifter



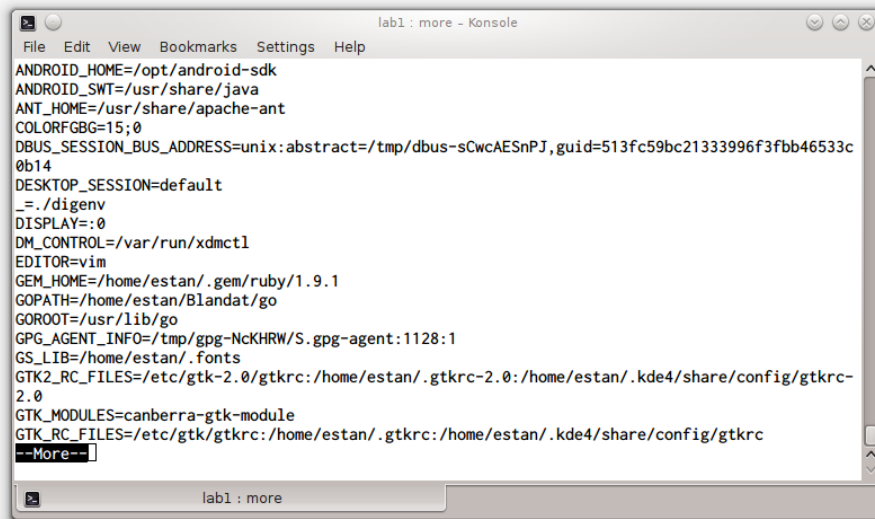
```
lab1 : most - Konsole
File Edit View Bookmarks Settings Help
ANDROID_HOME=/opt/android-sdk
ANDROID_SWT=/usr/share/java
ANT_HOME=/usr/share/apache-ant
COLORFGBG=15;0
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-sCwcAESnPJ,guid=513fc59bc21333996f3fbb46533$
DESKTOP_SESSION=default
_=./digenv
DISPLAY=:0
DM_CONTROL=/var/run/xdmctl
EDITOR=vim
GEM_HOME=/home/estan/.gem/ruby/1.9.1
GOPATH=/home/estan/Blandat/go
GOROOT=/usr/lib/go
GPG_AGENT_INFO=/tmp/gpg-NckHRW/S.gpg-agent:1128:1
GS_LIB=/home/estan/.fonts
GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/home/estan/.gtkrc-2.0:/home/estan/.kde4/share/config/gtkrc$
GTK_MODULES=canberra-gtk-module
GTK_RC_FILES=/etc/gtk/gtkrc:/home/estan/.gtkrc:/home/estan/.kde4/share/config/gtkrc
HG=/usr/bin/hg
-- MOST: *stdin* (1,1) 0%
Press 'Q' to quit, 'H' for help, and SPACE to scroll.
```

Figur 2: Både less och more i PATH och PAGER satt till en tredje pager (most).



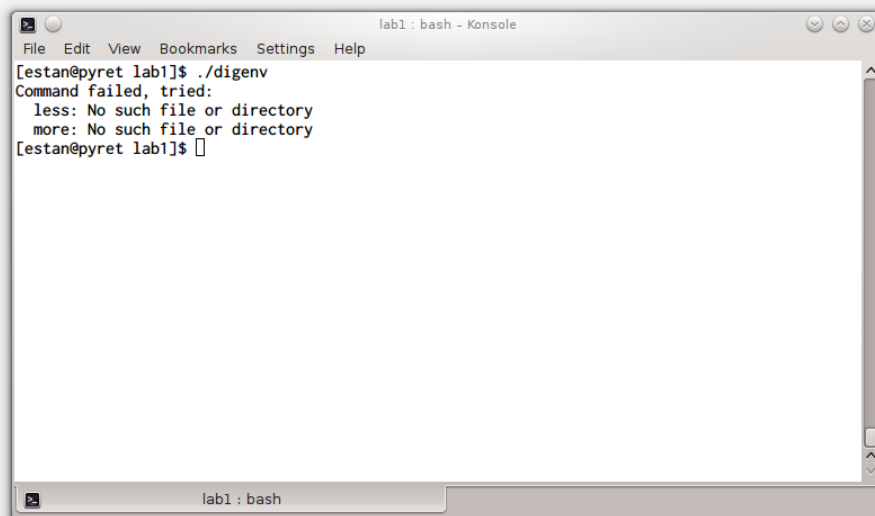
```
lab1 : less - Konsole
File Edit View Bookmarks Settings Help
ANDROID_HOME=/opt/android-sdk
ANDROID_SWT=/usr/share/java
ANT_HOME=/usr/share/apache-ant
COLORFGBG=15;0
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-sCwcAESnPJ,guid=513fc59bc21333996f3fbb46533c
0b14
DESKTOP_SESSION=default
_=./digenv
DISPLAY=:0
DM_CONTROL=/var/run/xdmctl
EDITOR=vim
GEM_HOME=/home/estan/.gem/ruby/1.9.1
GOPATH=/home/estan/Blandat/go
GOROOT=/usr/lib/go
GPG_AGENT_INFO=/tmp/gpg-NckHRW/S.gpg-agent:1128:1
GS_LIB=/home/estan/.fonts
GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/home/estan/.gtkrc-2.0:/home/estan/.kde4/share/config/gtkrc-
2.0
GTK_MODULES=canberra-gtk-module
GTK_RC_FILES=/etc/gtk/gtkrc:/home/estan/.gtkrc:/home/estan/.kde4/share/config/gtkrc
:[]
```

Figur 3: Både less och more i PATH men PAGER ej satt.



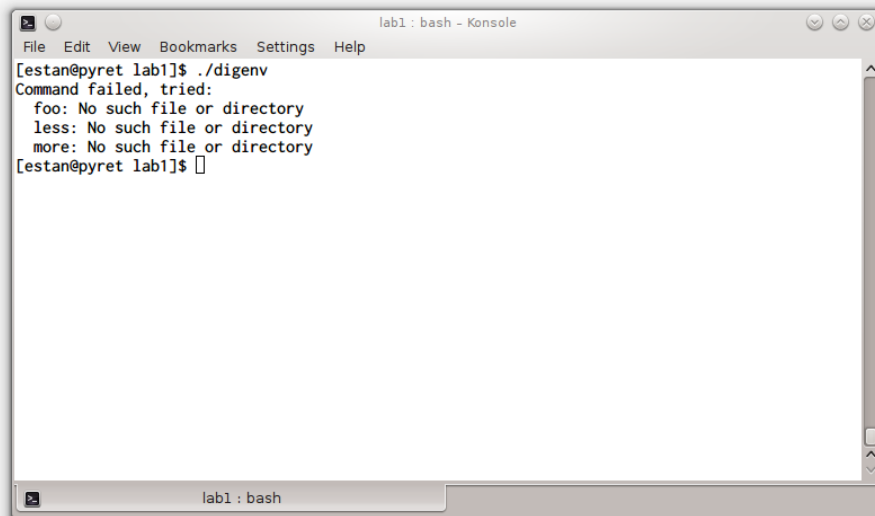
```
lab1 : more - Konsole
File Edit View Bookmarks Settings Help
ANDROID_HOME=/opt/android-sdk
ANDROID_SWT=/usr/share/java
ANT_HOME=/usr/share/apache-ant
COLORFGBG=15;0
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-sCwcAESnPJ,guid=513fc59bc21333996f3fbb46533c0b14
DESKTOP_SESSION=default
_=/digenv
DISPLAY=:0
DM_CONTROL=/var/run/xdmctl
EDITOR=vim
GEM_HOME=/home/estan/.gem/ruby/1.9.1
GOPATH=/home/estan/Blandat/go
GOROOT=/usr/lib/go
GPG_AGENT_INFO=/tmp/gpg-NckHRW/S.gpg-agent:1128:1
GS_LIB=/home/estan/.fonts
GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/home/estan/.gtkrc-2.0:/home/estan/.kde4/share/config/gtkrc-2.0
GTK_MODULES=canberra-gtk-module
GTK_RC_FILES=/etc/gtk/gtkrc:/home/estan/.gtkrc:/home/estan/.kde4/share/config/gtkrc
--More--
```

Figur 4: Bara more i PATH, men ej less och PAGER ej satt.



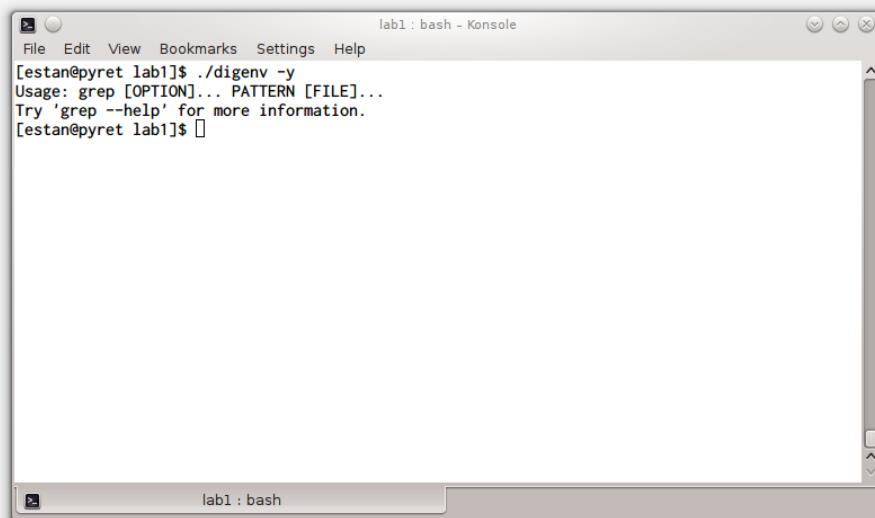
```
lab1 : bash - Konsole
File Edit View Bookmarks Settings Help
[estan@pyret lab1]$ ./digenv
Command failed, tried:
  less: No such file or directory
  more: No such file or directory
[estan@pyret lab1]$
```

Figur 5: Varken less eller more i PATH och PAGER ej satt.



```
lab1 : bash - Konsole
File Edit View Bookmarks Settings Help
[estan@pyret lab1]$ ./digenv
Command failed, tried:
  foo: No such file or directory
  less: No such file or directory
  more: No such file or directory
[estan@pyret lab1]$
```

Figur 6: Varken `less` eller `more` i `PATH` och `PAGER` satt till en ej existerande fil (`foo`).



```
lab1 : bash - Konsole
File Edit View Bookmarks Settings Help
[estan@pyret lab1]$ ./digenv -y
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
[estan@pyret lab1]$
```

Figur 7: Felaktig parameter (`-y`) given till `grep`.