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.

Processen med process-id 1 heter `init` och har i vårt system sökvägen `/sbin/init`.

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

Om en process ändrar på en enviromentvariabel i UNIX så kan endast processen själv och eventuella barnprocesser ta del av ändringen. Därför går det endast att skicka information "neråt" med hjälp av dessa.

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.

Systemet använder `sigaction` för att reglera vad som ska hända när en viss signal tas emot i en process. Detta fungerar inte för SIGKILL och SIGSTOP, vilket står tydligt i manualen. Det man får när man provkör programmet är ett felmeddelande: `sigaction() failed: Invalid argument.`

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

Child-processen har ingen nytta av att ges sitt eget process-id (detta kan hämtas genom ett anrop till `getpid` för sitt eget, och `getppid` för förälderns), det är parentprocessen som kan ha nytta av att känna till det, t.ex. för att kunna vänta på child-processen med `wait` eller döda den med `kill`. Värdet 0 är ett lämpligt returvärde för att indikera att vi är i child-processen, eftersom det lägsta PID:et är 1.

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?

Vi kan ej se något hinder för att filens offset sparas i varje process File Descriptor Table, annat än att många systemanrop måste göras om; exempelvis måste `ssize_t read(int fd, void *buf, size_t count);` utöver `fd` även ta emot offset som parameter. Dessutom kan det leda

till att två olika fildeskriptorer som pekar på samma fil (i File Table) har olika värden för offset (pekar på olika ställen i filen), vilket inte nödvändigtvis måste vara ett problem. Dock tappar man lite av operativsystemets fil-abstraktion mot programmeraren som skriver program för user-mode. Programmeraren kan ju i dagsläget använda `lseek(2)` för söka i filer, istället för att eventuellt mixtra med olika fildeskriptorer med olika offsets. Dessutom medför lagringen av offset-värdet i File Table att operativsystemet kan kontrollera hur user-program modifierar det.

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).

Vi råkade ut för just detta i vår implementation av labbuppgiften. Pagineraren väntade på EOF, men fick det aldrig, och väntade således för evigt.

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

Systemanropet `wait` blockerar tills en av förälderns barnprocesser har ändrat tillstånd. En ändring av tillstånd kan innebära att den antingen har avslutats normalt, eller just att processen har dött. Med hjälp av t.ex. `pid_t wait(int *status)` kan vi se på vilket sätt en process har avslutats. Det finns definierade makron för att tolka värden på denna variabel, som t.ex. `WIFEXITED`. Så ja, förälderprocessen kan upptäcka att barnprocessen plötsligt dör, genom att särskilja det från att den avslutas på normalt sätt.

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?

Om grep inte hittade något så avslutas programmet med exit-code (1). Vid felaktiga parameterar avslutas det med exit-code (2). Vi kan helt enkelt inspektera denna exit-code och på så sätt få reda på dessa saker. För att veta vilken process det handlar om så kan vi använda returvärdet från `wait`, och jämföra med undansparade process-id:n från ett tidigare `fork`-anrop.

Problembeskrivning

Uppgiften gick ut på att skriva ett program (`digenv.c`) som kan användas till att studera sina miljövariabler på ett smidigt sätt. Det skulle utnyttja operativsystemanrop för att styra in- och utmatningen av data mellan olika filter med hjälp av pipes. Ett filter karaktäriseras av att det läser från `stdin`, behandlar dataströmmen och skriver resultatet på `stdout`. De filter som, beroende på indatan till `digenv.c`, skulle anropas på olika sätt från programmet var `printenv`, `grep`, `sort` och `less`/`more` (`$PAGER`); `printenv` ska alltid anropas; om indataparametrar till programmet specificerats ska resultatet styras via `grep` (som ges indataparametrarna) till `sort`, annars ska resultatet gå direkt till `sort`; slutligen slås `$PAGER`-miljövariabeln upp för att se om en paginerare finns definierad, annars används `less` om möjligt och i övriga fall `more`. Det var ej tillåtet att använda C-bibliotekets anrop `system(3)` som kan utnyttja skalets funktionalitet.

Programbeskrivning

I uppgiftsbeskrivningen framgick det att det finns två sätt att lösa problemet på – rättframt och något naivt eller elegant med rekursion. Eftersom vi ansåg att uppgiftens syfte var att göra oss bekanta med systemanrop och kommunikation via pipes valde vi att implementera programmet rättframt utan rekursion. Exekveringen av `digenv.c` sker därmed i stort sett rätlinjigt med tillägget av några `if`-satser och anrop av bekvämlighetsfunktioner samt en `for`-slinga i slutet av programmet för att invänta rätt antal döda barn. Eftersom programmet ser ut på det enkla viset finns det inga direkta algoritmer att tala om som skulle kunna bytas ut. Den ändring som skulle kunna göras är uppenbarligen att implementera programmet med rekursion.

De enda funktioner som vi definierat (utöver `main`) är `check_error`, `_check_error` och `close_pipes`. Dessa beskrivs närmare i procedurkommentarerna i koden med respektive argumentparametrar.

Filkatalog

Den senaste versionen av koden finns tillgänglig via GitHub: `https://github.com/joelpet/Digenv/tree/master/code`. Observera att vi arbetade tillsammans, vilket är anledningen till den ojämna fördelningen av commits.

En klon (möjligen förlegad) av källkodsförrådet på GitHub finns att tillgå via `~joelpet/kurser/opsys/lab1/Digenv/code`. Användaren `robertr` och gruppen `os` ska båda ha list- och läsrättigheter i den katalogen:

```
$ fs la ~joelpet/kurser/opsys/lab1/Digenv/code
Access list for /afs/nada.kth.se/home/k/u1ad1pck/kurser/opsys/lab1/
Digenv/code is
```

```
Normal rights:
  joelpet:remote-users rlidwka
  robertr:os rl
  system:administrators rlidwka
  system:anyuser l
  joelpet rlidwka
```

Utskrift med kompileringskommandon och körningar

Körning utan parametrar

```
$ make digenv
$ ./digenv
COLORTERM=gnome-terminal
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
RGeVPPFtxO,guid=ce50c76a6801c2072e6713fc4db6
add7
DESKTOP_SESSION=default
DISPLAY=:0.0
…
```

Programmet listar environmentvariablerna sorterade i en paginerare, som förväntat.

Körning med giltig parameter

```
$ make digenv
$ ./digenv --v (visar versionsnumret för grep)
Copyright (C) 2009 Free Software Foundation, Inc.
GNU grep 2.5.4
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
There is NO WARRANTY, to the extent permitted by law.
This is free software: you are free to change and redistribute it.
(END)
```

Programmet skriver ut versionsinformation om grep sorterat, i en paginerare, som förväntat.

Alternativ körning med giltig parameter

```
$ make digenv
$ ./digenv -i user
USER=joelpet
XUSERFILESEARCHPATH=/usr/local/hacks/app-defaults/%N
```

```
(END)
```

Programmet skriver ut alla environmentvariabler vars namn innehåller “user” (skiftlägesokänsligt).

Körning med ogiltig parameter

```
$ make digenv
$ ./digenv --asdfg
(END)
```

Programmet skriver ut en tom sida i pagineraren, och programmet avslutas med exit-code (2), som förväntat.

Välkommenterad och strukturerad källkod

```
/*
 * NAME:
 *  digenv  -   study your environment variables
 *
 * SYNTAX:
 *  digenv [parameters]
 *
 * DESCRIPTION:
 *  Digenv displays your environment variables sorted in a pager, optionally
 *  filtered through `grep` with the given input parameters, if any present. If
 *  $PAGER is present, digenv will try to use that command as pager, otherwise
 *  it tries `less` and thereafter falls back to `more`
 *
 * OPTIONS:
 *  See grep(1). All parameters will be passed directly to `grep`.
 *
 * EXAMPLES:
 *  Simply display all environment variables sorted in a pager:
 *  $ digenv
 *
 *  Display all environment variables with a name containing "user" (-i means
 *  ignore case):
 *  $ digenv -i user
 *
 * ENVIRONMENT:
 *  PAGER    The command to execute for launching a pager.
 *
 * SEE ALSO:
 *   printenv(1), grep(1), sort(1), less(1), more(1)
 *
 * DIAGNOSTICS:
 *  The exit status is 0 if everything went fine, 1 if any system call
 *  failed, e.g. creating a pipe or executing a file, or if `grep` did not find
 *  anything, and 2 if `grep` failed or a child was terminated by a signal.
 *
 * NOTES:
 *  The exit statuses could be refined in order to better indicate exactly what
 *  went wrong.
 */
```

```
/* digenv
 *
 * This module contains the whole digenv program, including main() and
 * functions for error checking and closing pipes.
 */

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define PIPE_READ_SIDE 0
#define PIPE_WRITE_SIDE 1

/*
 * Define a pipe file descriptor type for neater code.
 */
typedef int pipe_fd_t[2];

void check_error(int, char*);
void _check_error(int, char*, int);
void close_pipes(int, pipe_fd_t*);


/* main
 *
 * main creates the necessary pipes and invokes the filters.
 */
int main(
      int argc,     /* number of given arguments */
      char** argv)  /* array of argument char arrays */
{
      pid_t childpid; /* placeholder for child process IDs */
      pipe_fd_t pipe_fds[3]; /* array of three pipe file descriptors */
      int num_filters = argc > 1 ? 4 : 3; /* number of filters to use */
      int num_pipes = num_filters - 1; /* number of pipes to use */
      int return_value; /* used for storing return values from syscalls */
      int status; /* used for getting status of exited child processes */
      int exit_code = 0; /* stores the code to exit with */
      int i; /* loop variable */
      int cur_pipe = 0; /* index of the pipe_fd that follows the current filter */
      char* pager; /* used to get the pager command name */

      /*
      * Create pipes.
      */
      for (i = 0; i < num_pipes; ++i) {
      return_value = pipe(pipe_fds[i]);
      check_error(return_value, "Could not initialize pipe.\n");
      }
```

```
        /*
        * Fork a child process and update its file descriptors to use a pipe for
        * its output. Then execute the `printenv` command, which will inherit the
        * File Descriptor Table.
        */
        childpid = fork();

        if (0 == childpid) {
        /*
        * This code is executed only in the child process.
        *
        * Overwrite stdout file descriptor with the one that points to the
        * write side of the current pipe, and close file descriptors to both
        * sides of the pipe, since we now have a file descriptor (stdout) that
        * points to the write side of the pipe. Then try to execute `printenv`
        * with "printenv" as the first argument, by convention.
        */

        return_value = dup2(pipe_fds[cur_pipe][PIPE_WRITE_SIDE], STDOUT_FILENO);
        check_error(return_value,  "printenv&: Could not duplicate write side of first
pipe.\n");

        return_value = close(pipe_fds[cur_pipe][PIPE_READ_SIDE]);
        check_error(return_value,  "Could not close read side of pipe.\n");

        return_value = close(pipe_fds[cur_pipe][PIPE_WRITE_SIDE]);
        check_error(return_value,  "Could not close write side of pipe.\n");

        (void) execlp("printenv", "printenv", (char *) 0);

        fprintf(stderr, "Could not execute printenv.\n");
        exit(1);
        }

        check_error(childpid,  "Could not fork printenv.\n");

        /*
        * If we get here, forking `printenv` went just fine, so increment cur_pipe
        * counter.
        */

        ++cur_pipe;   /* is now unconditionally 1 */

        /*
        * If arguments where given, call `grep` with those.
        */
        if (argc > 1) {
        /*
        * Fork a child process and update its file descriptors to use a pipe for
        * input and output. Then execute the `grep` command, which will
        * inherit the File Descriptor Table.
        */
        childpid = fork();

        if (0 == childpid) {
```

```
            /*
            * This code is executed only in the child process.
            *
            * Same thing here as previously; replace (stdin and) stdout file
            * descriptors with those from the current pipes. Then close unused
            * sides of the pipes.
            */
            return_value = dup2(pipe_fds[cur_pipe-1][PIPE_READ_SIDE], STDIN_FILENO);
            check_error(return_value, "grep&: Could not duplicate read side of
pipe.\n");

            return_value = close(pipe_fds[cur_pipe-1][PIPE_WRITE_SIDE]);
            check_error(return_value, "grep&: Could not close write side of pipe.");

            return_value = dup2(pipe_fds[cur_pipe][PIPE_WRITE_SIDE], STDOUT_FILENO);
            check_error(return_value, "grep&: Could not duplicate write side of
pipe.\n");

            return_value = close(pipe_fds[cur_pipe][PIPE_READ_SIDE]);
            check_error(return_value, "grep&: Could not close read side of second
pipe.");


            argv[0] = "grep";   /* digenv is longer than "grep", so no buffer
overflow */
            (void) execvp("grep", argv);

            fprintf(stderr, "Could not execute grep.\n");
            exit(1);
      }

      check_error(childpid,  "Could not fork grep.\n");

      /*
      * Another filter done, and everything went fine this far; increment
      * cur_pipe counter and also close the pipes in parent process, since
      * it's not going to use them.
      */
      ++cur_pipe;
      close_pipes(cur_pipe, pipe_fds);
      }

      /*
      * Same forking procedure as earlier, but this time executing `sort`.
      */
      childpid = fork();

      if (0 == childpid) {
      /*
      * Same as before; duplicate file descriptors and close unused pipe
      * ends. Then execute `sort`.
      */

      return_value = dup2(pipe_fds[cur_pipe-1][PIPE_READ_SIDE], STDIN_FILENO);
      check_error(return_value, "sort&: Could not duplicate read side of first
```

```
pipe.\n");

        return_value = close(pipe_fds[cur_pipe-1][PIPE_WRITE_SIDE]);
        check_error(return_value, "sort&: Could not close write side of first pipe.");

        return_value = dup2(pipe_fds[cur_pipe][PIPE_WRITE_SIDE], STDOUT_FILENO);
        check_error(return_value, "sort&: Could not duplicate write side of second
pipe.\n");

        return_value = close(pipe_fds[cur_pipe][PIPE_READ_SIDE]);
        check_error(return_value, "sort&: Could not close read side of second pipe.");

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

        fprintf(stderr, "Could not execute sort.\n");
        exit(1);
        }

        check_error(childpid,  "Could not fork sort.\n");

        /*
        * Again, everything went fine, so increment cur_pipe counter and close pipes.
        */
        ++cur_pipe;
        close_pipes(cur_pipe, pipe_fds);

        /*
        * This is the last filter, but forking procedure is almost the same; first
        * take care of pipes, but instead of simply executing a file, we first
        * need to determine what file (pager) to execute.
        */
        childpid = fork();

        if (0 == childpid) {
        /*
        * Replace stdin with read side of previous pipe and close unused sides
        * of the current pipe.
        */
        return_value = dup2(pipe_fds[cur_pipe-1][PIPE_READ_SIDE], STDIN_FILENO);
        check_error(return_value, "less&: Could not duplicate read side of pipe.\n");

        return_value = close(pipe_fds[cur_pipe-1][PIPE_WRITE_SIDE]);
        check_error(return_value, "less&: Could not close write side of pipe.\n");

        return_value = close(pipe_fds[cur_pipe-1][PIPE_READ_SIDE]);
        check_error(return_value, "less&: Could not close write side of pipe.\n");

        /*
        * Lookup environment variable PAGER, to see if such exists and, if so,
        * execute it. The commands `less` and `more` are provided as fallback
        * pagers, in that order. Exit with status 1 if no pager could be
        * executed.
        */
        pager = getenv("PAGER");
        if (NULL != pager) {
```

```
        (void) execlp(pager, pager, (char *) 0);
    }
    (void) execlp("less", "less", (char *) 0);
    (void) execlp("more", "more", (char *) 0);

    fprintf(stderr, "Could not execute pager.\n");
    exit(1);
    }

    check_error(childpid, "Could not fork.\n");

    /*
    * Once again, everything went fine, so increment cur_pipe counter and
    * close previous pipe in parent process.
    */
    ++cur_pipe;
    close_pipes(cur_pipe, pipe_fds);

    /*
    * Wait for filter children to exit and check their exit statuses.
    */
    for (i = 0; i < num_filters; ++i) {
    childpid = wait(&status);
    check_error(childpid, "wait() failed unexpectedly.\n");

    if (WIFEXITED(status)) {
        /*
        * Child terminated normally, by calling exit(), _exit() or by
        * return from main().
        */
        int child_status = WEXITSTATUS(status);
        if (0 != child_status && 0 == exit_code) {
        /*
        * Child terminated with non-zero status, indicating some
        * problem. However, it does not have to be fatal; grep exits
        * with status 1 if no matches were found. Save exit status.
        */
        exit_code = child_status;
        }
    } else {
        if (WIFSIGNALED(status) && 0 == exit_code) {
        /*
        * Child was terminated by a signal (WTERMSIG(status)). Set
        * exit status to non-zero to indicate this.
        */
        exit_code = 2;
        }
    }
    }

    /*
    * Exit with first non-zero child exit status, or 0 if everything went fine.
    */
    exit(exit_code);
}
```

```
/* check_error
 *
 * check_error calls _check_error() with a predefined exit code.
 */
void check_error(
      int return_value,   /* return value to check for -1 value */
      char* error_prefix) /* short string to prefix the error message with */
{
      _check_error(return_value, error_prefix, 1);
}

/* _check_error
 *
 * _check_error checks if return_value is -1 and takes appropriate actions,
 * such as printing an error message and exiting with the given exit_code.
 */
void _check_error(
      int return_value,   /* return value to check for -1 value */
      char* error_prefix, /* short string to prefix the error message with */
      int exit_code)      /* the code to exit the program with */
{
      if (-1 == return_value) {
      perror(error_prefix);
      exit(exit_code);
      }
}

/* close_pipes
 *
 * close_pipes closes pipes that pressumably are not going to be used.
 */
void close_pipes(
      int cur_pipe,             /* index of the pipe in pipe_fds that is going
                                   to be initiated next */
      pipe_fd_t* pipe_fds)      /* array of pipe file descriptors */
{
      int return_value;
      if (cur_pipe >= 2) {
      return_value = close(pipe_fds[cur_pipe-2][PIPE_WRITE_SIDE]);
      check_error(return_value, "Could not close write side of pipe.\n");
      return_value = close(pipe_fds[cur_pipe-2][PIPE_READ_SIDE]);
      check_error(return_value, "Could not close read side of pipe.\n");
      }
}
```

Verksamhetsberättelse och synpunkter på laborationen

Vi började med att skriva programmet så att det antog att inga inparametrar hade givits det och att pagineraren som skulle användas var `less`. När det fungerade gick vi vidare och lät `printenv` skicka sin utdata till `sort` via en pipe som sedan skrev sin utdata till `stdout`. På så sätt byggde vi ut programmet undan för undan i små steg och bekräftade efter varje nytt steg att det fungerade som

förväntat. Vi erfor problem med våra pipes i början då vi inte stängde läs- och skrivsidorna ordentligt, vilket ledde till att pagineraren aldrig fick `EOF` och således väntade på mer indata för evigt. I övrigt flöt arbetet på bra och vi blev klara med koden på ungefär 4 timmar, inklusive trasslet med stängningen av pipes. Därefter har ungefär 2 timmar spenderats på att författa denna rapport. Laborationen gav en god introduktion till programmering med systemanrop.