# Digenv - processkommunikation med pipes

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

**Problembeskrivning**
Skapa ett program som använder sig utav pipes för kommunikation mellan processer.
Vår uppgift var att skriva ett program i C för att göra motsvarande detta kommando
”printenv | grep | sort | less”
i Unix liknande miljö.

**Programbeskrivning**
Vi börjar med att initiera de pipes (3 pipes) och variabler vi tänker använda. Därefter följer de fyra child processerna:
1. printenv
   Ändrar STDOUT till pipe env_grep, sen stäng pipen med pipecloser(env_grep).
   printenv kommer hämta miljövariablerna och skriva dessa till pipen.
2. grep
   Ändrar STDOUT till pipe grep_sort och STDIN till pipe env_grep, sen stänger pipes med pipecloser(env_grep) och pipecloser(grep_sort).
   grep filtrerar skrift från pipen efter filter givna från användaren.
   grep kommer dock ej köras om användare inte har specificerat några argument vid program start.
3. sort
   Ändrar STDOUT till pipe sort_less och STDIN till pipe grep_sort, sen stänger pipes med pipecloser(env_grep), pipecloser(grep_sort) och pipecloser(sort_less).
   sort sorterar text i bokstavsordning och skriver detta till pipe.
4. less
   Ändrar inte STDOUT eftersom vi vill att den nu ska skriva till terminalen men vi ändrar STDIN till pipe sort_less, sedan stänger vi samtliga pipes.

Sist i programmet stänger vi igen all pipes i parent processen och går in i en for-loop som väntar på att varje child-process är klar. Om man inte gör detta, hinner childprocesserna aldrig bli klara innan parent avslutas. Detta medför att childprocesserna blir liggande som zombies och programmet kommer inte fungera i sin helhet.

Vi valde att göra på detta sätt för att
2. det fungerade inte att initiera pipes allteftersom man behövde dem innan respektive fork(). Dessutom blir det mera överskådligt när alla pipes ligger på samma ställe i koden.
3. denna lösning är rakt på sak och vi använder lite extra funktioner som pipecloser för att få en mera lättläst kod. Man kan säga att detta var en enkel lösning på problemet.

De hjälpfunktioner vi använder är:
*choosePager()*
   Denna funktion kontrollerar vilken pager som finns fördefinierad i miljövariablerna.
   Finns det ingen definierad använder vi less.
   Senare i koden kontrollerar vi dessutom om less finns tillgängligt genom att fånga felmeddelande vid exekvering. Finns inte less, körs more.
*pipeCloser(int p[])*
   Denna funktion tar in en int-array. Denna array pekar på den filedescriptor som avses stängas.
   Det sker också en kontroll om det inte skulle gå att stänga en pipe, och skriver då ut ett felmeddelande.

**Filkatalog**

Ange i vilken filkatalog (t.ex. `~pelle/os/lab1/`) du sparat din källkod och ditt program. Ange också vad dina filer heter.
*OBS!* Se till så att filerna är läsbara för assistenterna (se 3.4 ).

**Utskrift med kompileringskommandon och körningar**
Utskrift från programkörningar kan finnas i samma mapp som programmet samt koden.
De filer som finns visar utskrifter vid körning av följande kommandon:

- ./digenv - normal
- ./digenv PATH - withargument
- ./digenv Pelle - withfalseargument

Kompilatorn vi har använt är ***gcc version 4.3.2 (Ubuntu 4.3.2-1ubuntu11)*** och kördes med flaggorna -***Wall -o digenv***.

**Verksamhetsberättelse och synpunkter på laborationen**
Vi arbetade cirka en vecka med labben och började med att läsa labPM, därefter förenkla uppgiften given med psudo kod och lite figurer som beskriver hur pipes och processer. Då vi trodde vi hade en stabil lösning på hur programmet skulle se ut jagade vi syntax errors i några dagar. Slutligen fick vi en rätt okej lösning men vi blev tvugna att kasta om pipes till början av koden och göra en rejäl kontroll på att vi stänger alla pipes som använts.

**Förbedelsefrågor**
**1.** init
**2.** Nej, eftersom environmentalvariablerna ärvs från föräldrarprocessen, och man inte kan ärva åt andra hållet dvs parent från childprocessen.
**3.** Processen kommer inte göra mycket eftersom SIGKILL inte är en signal som går att fånga.
**4.** Anledningen till att `fork` skickar 0 till child och PID för child till parent, är för att parent ska kunna hantera och avsluta child på ett korrekt och smidigt vis.
**5.** File Table behövs för att systemet ska kunna ha kontroll över filer som är öppna och dess statusflaggor. Gällande offset i File Descritor Table, då kan man inte veta var i filen 2 olika processer behinner sig. Vilket gör skrivning svårt, och risken för överskrivning blir överhängande.
**6.** Ja. Så länge den inte har använts tidigare i någon process.
**7.** Ja, men inte per automatik. Det måste skrivas en funktion för kontroll av andra processer.
8. Programmet kan kontrollera om `grep` misslyckades genom att läsa exit-status för processen. I annat fall kan waitpid användas för att kontrollera om processen avslutades ordentligt.

```
#include <sys/types.h> /* For defining of type pid_t */
#include <sys/wait.h> /* For defining WIFEXITED */
#include <errno.h> /* Used for perror() */
#include <stdio.h> /* For defining of stderr */
#include <stdlib.h> /* definierar bland annaoch RAND_MAX */
#include <unistd.h> /* Used for defining of pipe() */
#define PIPE_READ_SIDE ( 0 )
#define PIPE_WRITE_SIDE ( 1 )

pid_t childpid; /* Child-process PID when started with fork() */

/*------PAGER START------*/
/* This function is used to check which pager we should use.
 * If there are no pager defined in environmental variables,
 * we will use less. */

char* choosePager(){
        char *pager, *test;
        test = getenv("PAGER");
        if(test == NULL)
                pager = "less";
        else
                pager = test;
        return pager;
}
/*------PAGER END-------*/
```

```
/*-------------pipeCloser START--------------*/
/* Function for closing the READ and WRITE side of the specified pipe */
void pipeCloser(int p[]){
	int return_value;
	return_value = close( p[ PIPE_READ_SIDE ] );
	if( -1 == return_value) {
		perror( "Cannot close READ side pipe " ); exit( 1 );
		}
	return_value = close( p[ PIPE_WRITE_SIDE ] );
	if( -1 == return_value) {
		perror( "Cannot close WRITE side of pipe " ); exit( 1 );
		}
}
/*-------------pipeCloser END--------------*/
int main(int argc, char **parameterlista)
{
	/* Variables for the program. Very usefull. */
	int return_value; /* Used to store the returnvalues from systemcalls. */
	int env_grep[2]; /* Our filedescriptors for the pipes. */
	int grep_sort[2];
	int sort_less[2];
	int status;
	char *pager = choosePager();


	/*-------------PIPE CREATION START--------------*/
	/* We create all pipes that we will use for the program */
	return_value = pipe( env_grep );
	if( -1 == return_value ) {
		perror( "Cannot create pipe env_grep" ); exit( 1 );
	}

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

	return_value = pipe( sort_less);
	if( -1 == return_value) {
		perror( "Cannot create pipe sort_less" ); exit( 1 );
	}
	/*-------------PIPE CREATION END--------------*/

	/*-------------ENV PROCESS START--------------*/
	/* Write the environmental varibales into our env_grep pipe */
	childpid = fork();
	if( 0 == childpid )
	{
		/* Duplicate the filedescriptor so that pipe is written to
		 * instead of stdout */
		return_value = dup2( env_grep[ PIPE_WRITE_SIDE ],
					STDOUT_FILENO ); /* STDOUT_FILENO == 1 */
		if( -1 == return_value ) {
			perror( "Cannot dup filedescriptor for printenv" ); exit( 1 );
		}

		pipeCloser(env_grep);

		execlp("printenv", "printenv", NULL);
		fprintf(stderr, "execlp of printenv failed "); exit( 1 ) ;
	}
	else if (childpid < 0) {
```

```
perror( "Cannot create env" ); exit( 1 );
}
/*---------------ENV PROCESS END--------------*/

/*-------------GREP PROCESS STARTS--------------*/
/* Redirect the STDIN to env_grep-pipe and STDOUT to grep_sort-pipe.
 * Takes the input values in this case the enviromental variables
 * and runs system call grep */
childpid = fork();

if( 0 == childpid )
{
        return_value = dup2( grep_sort[ PIPE_WRITE_SIDE ],
                                STDOUT_FILENO );
        if( -1 == return_value ) {
                perror( "Cannot dup filedescriptor for grep" ); exit( 1 );
        }

        return_value = dup2( env_grep[ PIPE_READ_SIDE ],
                                STDIN_FILENO );
        if( -1 == return_value ) {
                perror( "Cannot dup filedescriptor for grep" ); exit( 1 );
        }

pipeCloser(grep_sort);
pipeCloser(env_grep);

        if( argc >= 2 ) {
                parameterlista[0] = "grep";
                execvp("grep", parameterlista);
                fprintf(stderr, "execvp of grep failed "); exit( 1 ) ;
        }
        else {
                execlp("grep", "grep", "", NULL);
                fprintf(stderr, "execlp of grep failed "); exit( 1 ) ;
        }
}
else if (childpid < 0) {
perror( "Cannot create grep" ); exit( 1 );
}
/*---------------GREP PROCESS END--------------*/

/*-------------SORT PROCESS STARTS--------------*/
/* Redirect the STDIN to grep_sort and STDOUT to sort_less, takes the input values
 * in this case the enviromental variables left after grep and does system call sort*/
childpid = fork();

if( 0 == childpid )
{
        return_value = dup2( sort_less[ PIPE_WRITE_SIDE ],
                                STDOUT_FILENO );
        if( -1 == return_value ) {
                perror( "Cannot dup filedescriptor for sort" ); exit( 1 );
        }

        return_value = dup2( grep_sort[ PIPE_READ_SIDE ],
                                STDIN_FILENO ); /* STDOIN_FILENO == 0 */
        if( -1 == return_value ) {
                perror( "Cannot dup" ); exit( 1 );
        }

        pipeCloser(grep_sort);
pipeCloser(env_grep);
```

```
pipeCloser(sort_less);

        execlp("sort", "sort", NULL);
        fprintf(stderr, "execlp of sort failed "); exit( 1 );
}
else if (childpid < 0) {
perror( "Cannot create sort" ); exit( 1 );
}
/*-------------SORT PROCESS END--------------*/

/*-------------LESS PROCESS STARTS--------------*/
/* Redirect the STDIN to sort_less, takes the input values
 * in this case the enviromental variables left after grep and put in the
 * correct order after sort. Doing system call less if less exits if not,
 * use more.*/
childpid = fork();

if( 0 == childpid ) {

        return_value = dup2( sort_less[ PIPE_READ_SIDE ],
        STDIN_FILENO );
        if( -1 == return_value ) {
                perror( "Cannot dup" ); exit( 1 );
        }

        pipeCloser(grep_sort);
pipeCloser(env_grep);
pipeCloser(sort_less);

        execlp(pager, pager, NULL);
        if(ENOENT == errno)
        {
                fprintf(stderr, "Pager not found, using 'less'");
                execlp("less", "less", NULL);
                if(ENOENT == errno)
                {
                        fprintf(stderr, "Pager not found, using 'less'");
                        execlp("more", "more", NULL);
                }
        }
        fprintf(stderr, "execlp of less failed "); exit( 1 );
}
else if (childpid < 0) {
perror( "Cannot create less" ); exit( 1 );
}
/*-------------LESS PROCESS END--------------*/

/*------------CLOSE UNUSED PIPES------------*/
/* Closing all pipes in parrent to make sure we don't wait for
 * any pipes to close. */
pipeCloser(grep_sort);
pipeCloser(env_grep);
pipeCloser(sort_less);
/*---------CLOSE UNUSED PIPES END-----------*/

/*---------WAIT START-----------*/
/* We need to wait for our child processes. */
int i;
for(i = 0; i <= 3; i++) {
        childpid = wait( &status );
        if( -1 == childpid ) {
                perror( "wait() failed unexpectedly" ); exit( 1 );
        }
```

```
                if( WIFEXITED( status ) ) {
                        int child_status = WEXITSTATUS( status );
                        if( 0 != child_status ) {
                                fprintf( stderr, "Child (pid %ld) failed with exit code %d\n",
                                (long int) childpid, child_status );
                        }
                }
                else {
                        if( WIFSIGNALED( status ) ) {
                                int child_status = WTERMSIG( status );
                                if( 0 != child_status ) {
                                fprintf( stderr, "Child (pid %ld) was terminated by signal no. %d\n",
                                (long int) childpid, child_status );
                                }       }       }       }
        /*---------WAIT END-----------*/
        exit( 0 );
}
```