Laborator 6

```
biancapinghireac@vbox:~/SO/lab6/src$ make lib
gcc -Wall -g -O   -c -o error.o error.c
gcc -Wall -g -O   -c -o prexit.o prexit.c
gcc -Wall -g -O   -c -o tellwait.o tellwait.c
ar rcs liblab6.a error.o prexit.o tellwait.o
biancapinghireac@vbox:~/SO/lab6/src$ make
gcc -o fork1 fork1.c liblab6.a
gcc -o wait1 wait1.c liblab6.a
gcc -o fork2 fork2.c liblab6.a
gcc -o tellwait1 tellwait1.c liblab6.a
gcc -o tellwait2 tellwait2.c liblab6.a
gcc -o echoall echoall.c liblab6.a
gcc -o exec1 exec1.c liblab6.a
```

Crearea librariei.

```
biancapinghireac@vbox:~/SO/lab6/src$ ./fork1
a write to stdout
before fork
pid = 4454, glob = 7, var = 89
pid = 4453, glob = 6, var = 88
biancapinghireac@vbox:~/SO/lab6/src$ ./fork1 > temp.out
biancapinghireac@vbox:~/SO/lab6/src$ cat temp.out
a write to stdout
before fork
pid = 4472, glob = 7, var = 89
before fork
pid = 4471, glob = 6, var = 88
```

```c
#include <sys/types.h>
#include "ourhdr.h"

int  glob = 6;                  /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int main(void)
{
        int     var;            /* automatic variable on the stack */
        pid_t   pid;

        var = 88;
        if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
                err_sys("write error");
        printf("before fork\n");        /* we don't flush stdout */

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {            /* child */
                glob++;                 /* modify variables */
                var++;
        } else
                sleep(2);               /* parent */

        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
```

Fork1.c

Cu write() scrie direct in stdout, pentru ca nu da flush la buffer, cand apeleaza printf(), in terminal se va afisa o singura data(se da flush automat), in schimb in fisier nu se da flush buffer-ului si se compiaza atat in procesul parinte cat si in cel fiu prin fork().

Sleep() pentru parinte lasa timp procesului copil sa actioneze(increment-ul valorilor) fara a influenta si valorile din cel parinte.

```
biancapinghireac@vbox:~/SO/lab6/src$ ./wait1
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
wait error: No child processes
normal termination, exit status = 1
```

```c
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"
#include "prexit.h"

int main(void)
{
        pid_t   pid;
        int     status;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)              /* child */
                exit(7);

        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)              /* child */
                abort();                        /* generates SIGABRT */

        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)              /* child */
                status = 0;             /* divide by 0 generates SIGFPE */

        if (wait(&status) != pid)               /* wait for child */
                err_sys("wait error");
        pr_exit(status);                        /* and print its status */

        exit(0);
}
```

Wait1.c

Primul copil se termina normal cu **exit(7);**.

Al doilea copil se termina anormal prin semnal (**abort()** → **SIGABRT**).

Al treilea copil ar trebui sa produca o exceptie de tip diviziune la zero (**SIGFPE**), in acest caz codul zice doar **status = 0;** .

```
biancapinghireac@vbox:~/SO/lab6/src$ ./fork2
biancapinghireac@vbox:~/SO/lab6/src$ second child, parent pid = 1920
```

```c
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int main(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {            /* first child */
                if ( (pid = fork()) < 0)
                        err_sys("fork error");
                else if (pid > 0)
                        exit(0);        /* parent from second fork == first child */

                /* We're the second child; our parent becomes init as soon
                   as our real parent calls exit() in the statement above.
                   Here's where we'd continue executing, knowing that when
                   we're done, init will reap our status. */

                sleep(2);
                printf("second child, parent pid = %d\n", getppid());
                exit(0);
        }

        if (waitpid(pid, NULL, 0) != pid)       /* wait for first child */
                err_sys("waitpid error");

        /* We're the parent (the original process); we continue executing,
           knowing that we're not the parent of the second child. */
        exit(0);
```

Fork2.c

In exemplul de mai sus se creaza un proces orfan(second child) care va fi preluat de systemd/init

```
biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait1
oouuttppuutt  ffrroomm  pcahrielndt

biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait1
output from parent
output from child
biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait1
output from parent
output from child
biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait1
output from child
output from parent
```

```c
#include <sys/types.h>
#include "ourhdr.h"

static void charatatime(char *);

int main(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {
                charatatime("output from child\n");
        } else {
                charatatime("output from parent\n");
        }
        exit(0);
}

static void charatatime(char *str)
{
        char    *ptr;
        int     c;

        setbuf(stdout, NULL);                   /* set unbuffered */
        for (ptr = str; c = *ptr++; )
                putc(c, stdout);
```

Tellwait1.c

In exemplul de mai sus se oberva lipsa sincronizarii proceselor care incearca sa modifice aceeasi zona de memorie (buffer ul stdout). Din acest motiv trebuie sincronizate.

```
biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait2
output from parent
output from child
```

```c
#include <sys/types.h>
#include "ourhdr.h"

static void charatatime(char *);

int main(void)
{
        pid_t   pid;

        TELL_WAIT();

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {
                WAIT_PARENT();          /* parent goes first */
                charatatime("output from child\n");
        } else {
                charatatime("output from parent\n");
                TELL_CHILD(pid);
        }
        exit(0);
}

static void charatatime(char *str)
{
        char    *ptr;
        int     c;

        setbuf(stdout, NULL);                   /* set unbuffered */
        for (ptr = str; c = *ptr++; )
                putc(c, stdout);
}
```

Tellwait2.c

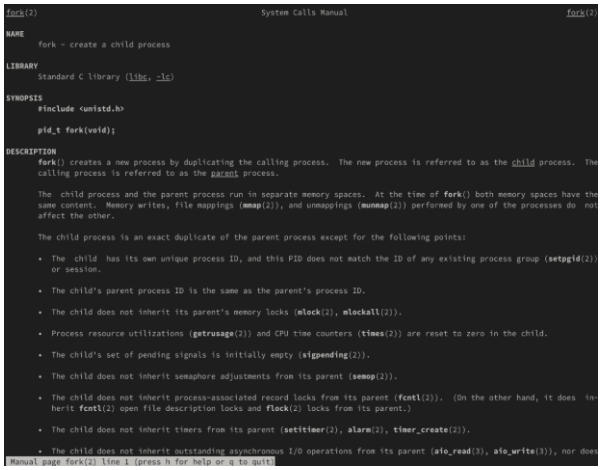
Aici se oberva sincronizarea, deoarece copilul asteasta ca parintele sat ermine procesul, apoi incele si el.

```
biancapinghireac@vbox:~/SO/lab6/src$ ./exec1
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
execlp error: No such file or directory
```

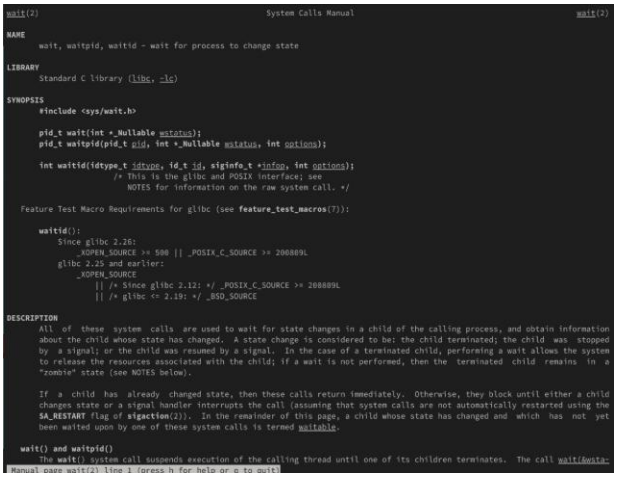
```c
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {    /* specify pathname, specify environment */
                if (execle("./echoall",
                                "echoall", "myarg1", "MY ARG2", (char *) 0,
                                env_init) < 0)
                        err_sys("execle error");
        }
        if (waitpid(pid, NULL, 0) < 0)
                err_sys("wait error");

        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0) {    /* specify filename, inherit environment */
                if (execlp("echoall",
                                "echoall", "only 1 arg", (char *) 0) < 0)
                        err_sys("execlp error");
        }
        exit(0);
```

Exec1.c

Programul creeaza doi copii cu fork. Primul foloseste **execle** pentru a rula **./echoall** cu argumente si un mediu (environment) propriu, unde PATH este doar **/tmp**. Functioneaza deoarece se da calea explicita spre executabil.
Al doilea copil foloseste **execlp** cu numele programului, dar nu il gaseste, pentru ca in PATH exista doar **/tmp**, iar acolo nu e **echoall**. De aceea apare eroare.
Exemplul arata diferenta dintre **execle** (cale si environment explicit) si **execlp** (moşteneste environment si cauta in PATH).

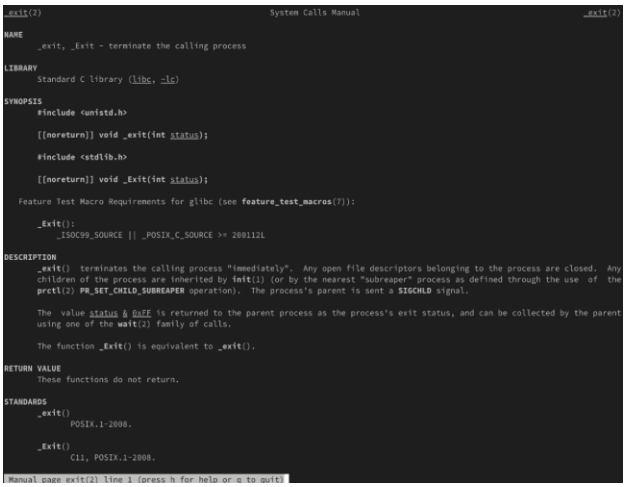


Fork



```
wait(2)                          System Calls Manual                          wait(2)

NAME
       wait, waitpid, waitid - wait for process to change state

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/wait.h>

       pid_t wait(int *_Nullable wstatus);
       pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                       /* This is the glibc and POSIX interface; see
                          NOTES for information on the raw system call. */

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       waitid():
           Since glibc 2.26:
                   _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L
               glibc 2.25 and earlier:
                   _XOPEN_SOURCE
                       || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
                       || /* glibc <= 2.19: */ _BSD_SOURCE

DESCRIPTION
       All  of  these  system  calls  are used to wait for state changes in a child of the calling process, and obtain information
       about the child whose state has changed.  A state change is considered to be: the child terminated; the child  was  stopped
       by  a signal; or the child was resumed by a signal.  In the case of a terminated child, performing a wait allows the system
       to release the resources associated with the child; if a wait is not performed, then the  terminated  child  remains  in  a
       "zombie" state (see NOTES below).

       If  a  child  has  already  changed state, then these calls return immediately.  Otherwise, they block until either a child
       changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the
       SA_RESTART flag of sigaction(2)).  In the remainder of this page, a child whose state has changed and  which  has  not  yet
       been waited upon by one of these system calls is termed waitable.

   wait() and waitpid()
       The wait() system call suspends execution of the calling thread until one of its children terminates.  The call wait(&wsta-
Manual page wait(2) line 1 (press h for help or q to quit)
```

Wait



```
_exit(2)                         System Calls Manual                          _exit(2)

NAME
       _exit, _Exit - terminate the calling process

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <unistd.h>

       [[noreturn]] void _exit(int status);

       #include <stdlib.h>

       [[noreturn]] void _Exit(int status);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       _Exit():
           _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION
       _exit()  terminates the calling process "immediately".  Any open file descriptors belonging to the process are closed.  Any
       children of the process are inherited by init(1) (or by the nearest "subreaper" process as defined through the use  of  the
       prctl(2) PR_SET_CHILD_SUBREAPER operation).  The process's parent is sent a SIGCHLD signal.

       The  value status & 0xFF is returned to the parent process as the process's exit status, and can be collected by the parent
       using one of the wait(2) family of calls.

       The function _Exit() is equivalent to _exit().

RETURN VALUE
       These functions do not return.

STANDARDS
       _exit()
              POSIX.1-2008.

       _Exit()
              C11, POSIX.1-2008.

Manual page exit(2) line 1 (press h for help or q to quit)
```

Exit

EXERCITIUL 2:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define BUFFER_SIZE 1024

void main(){
        char command[BUFFER_SIZE];
        int status;
        while(1){
                printf("$ ");
                fgets(command, BUFFER_SIZE, stdin);
                command[strlen(command)-1]='\0';

                if(strcmp(command,"exit")==0)
                        break;

                pid_t pid = fork(); //proces nou
                if(pid==-1){ //procesul nu s a creat corespunzator
                        exit(20); //20 fork
                }else if(pid ==0){
                        if(execlp(command,command,(char *) NULL) ==-1){
                                exit(21); //21 command error
                        }
                }
                else{
                        waitpid(pid,&status,0);
                }
        }
        exit(1);
}
```

Citeste comenzile de la tastatura si le ruleaza intr-un proces copil folosind fork si execlp. Daca scrii exit, se opreste. Daca sunt erori la fork sau la exec, programul iese cu un cod de eroare special.te

```
biancapinghireac@vbox:~/SO/lab6/src$ ./shell
$ tree
.
├── echoall
├── echoall.c
├── error.c
├── error.o
├── exec1
├── exec1.c
├── fork1
├── fork1.c
├── fork2
├── fork2.c
├── liblab6.a
├── Makefile
├── ourhdr.h
├── prexit.c
├── prexit.h
├── prexit.o
├── shell
├── shell.c
├── tellwait1
├── tellwait1.c
├── tellwait2.c
├── tellwait.c
├── tellwait.o
├── temp.out
├── wait1
└── wait1.c

1 directory, 26 files
$
```

EXERCITIUL 3:

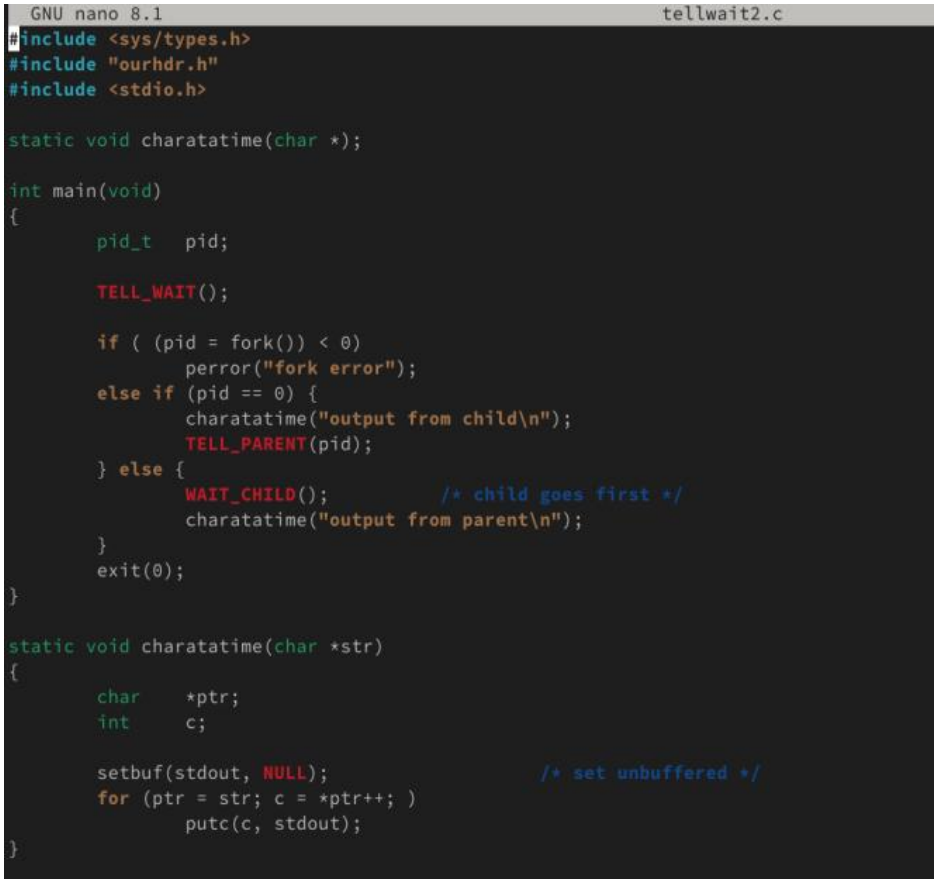
```
biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait2
output from parent
output from child
```

Nemodificat

```
biancapinghireac@vbox:~/SO/lab6/src$ gcc tellwait2.c -o tellwait2
biancapinghireac@vbox:~/SO/lab6/src$ ./tellwait2
output from child
output from parent
```

Modificat

Cod:



Am inversat ordinea proceselor(acum procesul parinte asteapta terminarea procesului fiu).