Sisteme de operare
Tema 6

Exercițiul 1

Rulați toate programele prezentate. Asigurați-vă că le-ați înțeles funcționarea. Folosiți-vă de pagina 2 de manual (comanda `man`).

```
[user@fedora destination]$ 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
[user@fedora destination]$ 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
```

Am creat bilbioteca liblab6.a .

```
[user@fedora destination]$ ./fork1
a write to stdout
before fork
pid = 9917, glob = 7, var = 89
pid = 9916, glob = 6, var = 88
[user@fedora destination]$ ./fork1 > temp.out
[user@fedora destination]$ cat temp.out
a write to stdout
before fork
pid = 9922, glob = 7, var = 89
before fork
pid = 9921, glob = 6, var = 88
```

Programul fork1.

Singura diferență constatată este la afișarea pid-ului. Prima dată are 9917 și 9916, după are 9922 și 9921.

```
[user@fedora destination]$ ./wait1
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
wait error: No child processes
normal termination, exit status = 1
```

Programul wait1 .

```
[user@fedora destination]$ ./fork2
[user@fedora destination]$ second child, parent pid = 1
```

Programul fork2.

```
[user@fedora destination]$ ./tellwait1
output from parent
outp[user@fedora destination]$ ut from child
```

Programul tellwait1.

```
[user@fedora destination]$ ./tellwait2
output from parent
o[user@fedora destination]$ utput from child
```

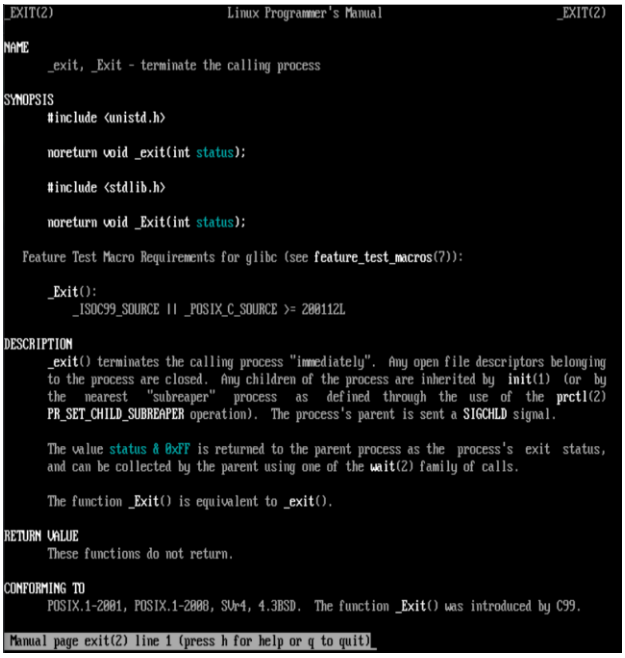
Programul tellwait2.

```
[user@fedora destination]$ ./exec1
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
[user@fedora destination]$ execlp error: No such file or directory
```

Programul exec1.

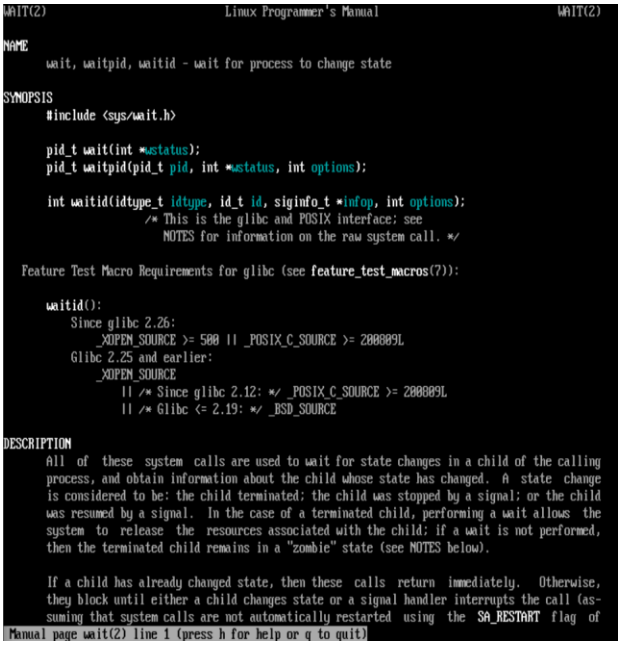
```
FORK(2)                    Linux Programmer's Manual                    FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a new process by duplicating the calling process.  The new process is re-
       ferred to as the child process.  The calling process is referred to as the parent process.

       The child process and the parent process run in separate memory spaces.  At  the  time  of
       fork()  both memory spaces have the same content.  Memory writes, file mappings (mmap(2)),
       and unmappings (munmap(2)) performed by one of the processes do not affect the other.

       The child process is an exact duplicate of the parent process  except  for  the  following
       points:

       *  The  child has its own unique process ID, and this PID does not match the ID of any ex-
          isting process group (setpgid(2)) or session.

       *  The child's parent process ID is the same as the parent's process ID.

       *  The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

       *  Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset
          to zero in the child.

       *  The child's set of pending signals is initially empty (sigpending(2)).

       *  The child does not inherit semaphore adjustments from its parent (semop(2)).

       *  The  child does not inherit process-associated record locks from its parent (fcntl(2)).
Manual page fork(2) line 1 (press h for help or q to quit)
```

Funcția fork().



```
_EXIT(2)                        Linux Programmer's Manual                       _EXIT(2)

NAME
       _exit, _Exit - terminate the calling process

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.

CONFORMING TO
       POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.  The function _Exit() was introduced by C99.

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

Funcția exit.



```
WAIT(2)                         Linux Programmer's Manual                        WAIT(2)

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

SYNOPSIS
       #include <sys/wait.h>

       pid_t wait(int *wstatus);
       pid_t waitpid(pid_t pid, int *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 (as-
       suming that system calls are not automatically restarted  using  the  SA_RESTART  flag  of
Manual page wait(2) line 1 (press h for help or q to quit)
```

Funcția wait.

Exercițiul 2

Scrieți un program care să implementeze un shell minimal. Shell-ul va trebui să citească de la standard input comenzi și să le execute.

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

#define BUFFER_SIZE 1024

int 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();
		if(pid==-1)
		{
			printf("Fork failed\n");
			exit(EXIT_FAILURE);
		}
		else if(pid==0)
		{
			if(execlp(command,command,(char *) NULL)==-1)
			{
				printf("Failed to execute command\n");
				exit(EXIT_FAILURE);
			}
		}
		else
		{
			waitpid(pid,&status,0);
		}
	}
	return 0;
}
```

Codul în C pentru programul nostru.

```
[user@desktop-5p6viv2 destination]$ make exercitiu2
gcc -Wall -g -O    exercitiu2.c   -o exercitiu2
[user@desktop-5p6viv2 destination]$ ./exercitiu2
$ ls
echoall    error.o  exercitiu2    fork1.c  liblab6.a  prexit.c  tellwait1    tellwait2.c  wait1
echoall.c  exec1    exercitiu2.c  fork2    Makefile   prexit.h  tellwait1.c  tellwait.c   wait1.c
error.c    exec1.c  fork1         fork2.c  ourhdr.h   prexit.o  tellwait2    tellwait.o
```

Am rulat programul creat și l-am verificat cu un input.

Exercițiul 3

Modificați programul `tellwait2` astfel încât procesul fiu să fie executat primul.

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

static sem_t semafor;

static void charatatime(char *);

int main(void)
{
        pid_t   pid;
        sem_init(&semafor,0,1);
        TELL_WAIT();

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

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

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

Programul modificat.

```
[user@fedora destination]$ ./tellwait2
output from parent
o[user@fedora destination]$ utput from child
```

L-am executat înainte de modificare.

```
[user@desktop-5p6viv2 destination]$ ./tellwait2
output from child
output from parent
```

L-am executat după modificare.