

Laborator 03 - Procese

Materiale ajutătoare

- [lab03-slides.pdf](#)
- [lab03-refcard.pdf](#)
- [Video Procese](#)

Nice to read

- TLPI - Chapter 6, Processes, Chapter 26 Monitoring Child Processes
- WSP4 - Chapter 6, Process Management

Prezentare concepte

Un **proces** este un program în execuție. Procesele sunt unitatea primitivă prin care sistemul de operare alocă resurse utilizatorilor. Orice proces are un spațiu de adrese și unul sau mai multe fire de execuție. Putem avea mai multe procese ce execută același program, dar oricare două procese sunt complet independente.

Informațiile despre procese sunt ținute într-o structură numită **Process Control Block (PCB)**, câte una pentru fiecare proces existent în sistem. Printre cele mai importante informații regăsim:

- PID - identificatorul procesului
- spațiu de adresă
- registre generale, PC (contor program), SP (indicator stivă)
- tabela de fișiere deschise
- [informații referitoare la semnale](#)
 - lista de semnale blocate, ignorate sau care așteaptă să fie trimise procesului
 - handler-ele de semnale
- informațiile referitoare la sistemele de fișiere (directorul rădăcină, directorul curent)

În momentul lansării în execuție a unui program, în sistemul de operare se va crea un proces pentru alocarea resurselor necesare rulării programului respectiv. Fiecare sistem de operare pune la dispoziție apeluri de sistem pentru lucrul cu procese: creare, terminare, așteptarea terminării. Totodată există apeluri pentru duplicarea descriptorilor de resurse între procese, ori închiderea acestor descriptori.

Procese pot avea o organizare:

- ierarhică - de exemplu pe Linux - există o structură arborescentă în care rădăcina este procesul `init` (`pid = 1`).
- neierarhică - de exemplu pe Windows.

În general, un proces rulează într-un mediu specificat printr-un set de **variabile de mediu**. O variabilă de mediu este o pereche `NUME = valoare`. Un proces poate să verifice sau să seteze valoarea unei variabile de mediu printr-o serie de apeluri de bibliotecă. ([Linux](#), [Windows](#))

Pipe-urile (canalele de comunicație) sunt mecanisme primitive de comunicare între procese. Un pipe poate conține o cantitate limitată de date. Accesul la aceste date este de tip FIFO (datele se scriu la un capăt al pipe-ului pentru a fi citite de la celălalt capăt). Sistemul de operare garantează sincronizarea între operațiile de citire și scriere la cele două capete. ([Linux](#), [Windows](#))

Există două tipuri de pipe-uri:

- pipe-uri **anonime**: pot fi folosite doar de procese **înrudite** (un proces părinte și un copil sau doi copii), deoarece sunt accesibile doar prin moștenire. Aceste pipe-uri nu mai există după ce procesele și-au terminat execuția.
- pipe-uri **cu nume**: au suport fizic - există ca fișiere cu drepturi de acces. Aceasta înseamnă că ele vor exista independent de procesul care le creează și pot fi folosite de procese neînrudite.

Procese în Linux

Lansarea în execuție a unui program presupune următorii pași:

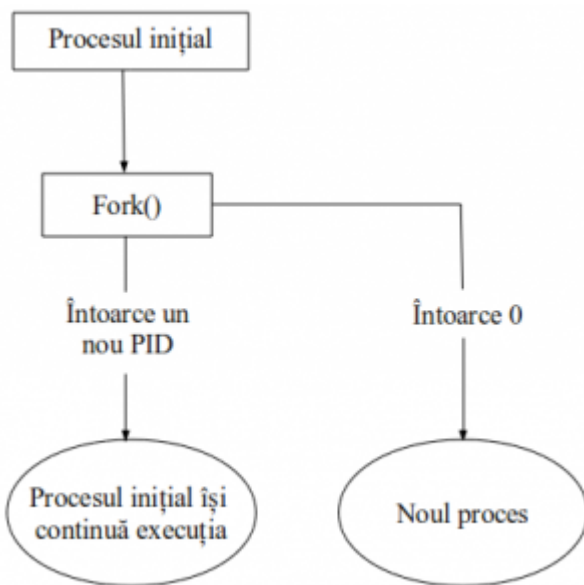
- Se creează un nou proces cu `fork` - procesul copil are o copie a resurselor procesului părinte.
- Dacă se dorește înlocuirea imaginii procesului copil aceasta poate fi schimbată prin apelarea unei funcții din familia `exec*`.

Crearea unui proces

În UNIX un proces se creează folosind apelul de sistem `fork`:

```
pid_t fork(void);
```

Efectul este crearea unui nou proces (procesul copil), copie a celui care a apelat `fork` (procesul părinte). Procesul copil primește un nou *process id* (PID) de la sistemul de operare.



Atenție! - această funcție este apelată o dată și se întoarce (în caz de succes) de două ori, așa cum se poate vedea în figura din dreapta.

Pentru aflarea PID-ului procesului curent ori al procesului părinte se va apela una din funcțiile de mai jos.

Funcția `getpid` întoarce PID-ul procesului apelant:

```
pid_t getpid(void);
```

Funcția `getppid` întoarce PID-ul procesului părinte al procesului apelant:

```
pid_t getppid(void);
```

Înlocuirea imaginii unui proces

Familia de funcții `exec` va executa un nou program, înlocuind imaginea procesului curent, cu cea dintr-un fișier (executabil). Spațiul de adrese al procesului va fi înlocuit cu unul nou, creat special pentru execuția fișierului. De asemenea vor fi reinițializate registrele IP (contorul program) și SP (indicatorul stivă) și registrele generale. Măștile de semnale ignorate și blocate sunt setate la valorile implicite, ca și handler-ele semnalelor. PID-ul și descriptorii de fișiere care nu au setat flag-ul `CLOSE_ON_EXEC` rămân neschimbați (implicit, flag-ul `CLOSE_ON_EXEC` nu este setat).

```
int execl(const char *path, const char *arg, ...); | execl("/bin/ls", "ls", "-la", NULL);
int execv(const char *path, char *const argv[]); | char *const argvec[] = {"ls", "-la", NULL};
| execv("/bin/ls", argvec);
int execlp(const char *file, const char *arg, ...); | execlp("ls", "ls", "-la", NULL);
```

Se observă că primul argument este numele programului. Ultimul argument al listei de parametri trebuie să fie `NULL`, indiferent dacă lista este sub formă de vector (`execv*`) sau sub formă de argumente variabile (`execl*`).

`exec1` și `execv` nu caută programul dat ca parametru în `PATH`, astfel că acesta trebuie însoțit de calea completă. Versiunile `exec1p` și `execvp` caută programul și în `PATH`.

Toate funcțiile `exec*` sunt implementate prin apelul de sistem [execve](#)

Așteptarea terminării unui proces

Familia de funcții [wait](#) suspendă execuția procesului apelant până când procesul (procesele) specificat în argumente fie s-a terminat, fie a fost oprit (`SIGSTOP`).

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Starea procesului interogată se poate afla examinând `status` cu macrodefiniții precum [WEXITSTATUS](#), care întoarce codul de eroare cu care s-a încheiat procesul așteptat, evaluând cei mai semnificativi 8 biți.

Există o variantă simplificată, care așteaptă orice proces copil să se termine. Următoarele secvențe de cod sunt echivalente:

```
wait(&status); | waitpid(-1, &status, 0);
```

În caz că se dorește doar așteptarea terminării procesului copil, nu și examinarea statusului:

```
wait(NULL);
```

Terminarea unui proces

Pentru terminarea procesului curent, Linux pune la dispoziție apelul de sistem `exit`. Dintr-un program C există trei moduri de invocare a acestui apel de sistem:

1. apelul `_exit` ([POSIX.1-2001](#)):

```
void _exit(int status);
```

2. apelul `_Exit` din biblioteca standard C (conform [C99](#)):

```
void _Exit(int status);
```

3. apelul `exit` din biblioteca standard C (conform C89, C99):

```
void exit(int status);
```

`_exit(2)` și `_Exit(2)` sunt funcțional echivalente (doar că sunt definite de standarde diferite):

- procesul apelant se va termina imediat
- toți descriptorii de fișier ai procesului sunt închiși
- copiii procesului sunt "înfițați" de `init`
- părintelui procesului îi va fi trimis un semnal `SIGCHLD`. Tot acestuia îi va fi întoarsă valoarea `status`, ca rezultat al unei funcții de așteptare (`wait` sau `waitpid`).

În plus, `exit(3)`:

- va șterge toate fișierele create cu `tmpfile()`
- va scrie bufferele streamurilor deschise și le va închide

Notă: Conform ISO C, un program care se termină cu `return x` din `main()` va avea același comportament ca unul care apelează `exit(x)`.

Pentru terminarea unui alt proces din sistem, se va trimite un semnal către procesul respectiv prin intermediul apelului de sistem [kill](#). Mai multe detalii despre `kill` și semnale în [laboratorul de semnale](#).

Exemplu(my_system)

[my_system.c](#)

```
#include <stdlib.h>
#include <stdio.h>

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int my_system(const char *command)
```

```

{
    pid_t pid;
    int status;
    const char *argv[] = {command, NULL};

    pid = fork();
    switch (pid) {
    case -1:
        /* error forking */
        return EXIT_FAILURE;

    case 0:
        /* child process */
        execvp(command, (char *const *) argv);

        /* only if exec failed */
        exit(EXIT_FAILURE);

    default:
        /* parent process */
        break;
    }

    /* only parent process gets here */
    waitpid(pid, &status, 0);
    if (WIFEXITED(status))
        <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Child
%d terminated normally, with code %d\n",
        pid, WEXITSTATUS(status));

    return status;
}

int main(void) {
    my_system("ls");
    return 0;
}

```

Copierea descriptorilor de fișier

[dup](#) duplică descriptorul de fișier `oldfd` și întoarce noul descriptor de fișier, sau `-1` în caz de eroare:

```
int dup(int oldfd);
```

[dup2](#) duplică descriptorul de fișier `oldfd` în descriptorul de fișier `newfd`; dacă `newfd` există, mai întâi va fi închis. Întoarce noul descriptor de fișier, sau `-1` în caz de eroare:

```
int dup2(int oldfd, int newfd);
```

Descriptorii de fișier sunt, de fapt, indecși în tabela de fișiere deschise. Tabela este populată cu pointeri către structuri cu informațiile despre fișiere. Duplicarea unui descriptor de fișier înseamnă duplicarea intrării din tabela de fișiere deschise (adică 2 pointeri de la poziții diferite din tabelă vor indica spre aceeași structură din sistem, asociată fișierului). Din acest motiv, toate informațiile asociate unui fișier (lock-uri, cursor, flag-uri) sunt **partajate** de cei doi file descriptori. Aceasta înseamnă că operațiile ce modifică aceste informații pe unul din file descriptori (de ex. `lseek`) sunt vizibile și pentru celălalt file descriptor (duplicat). **Atentie!** Există și o excepție: flag-ul `CLOSE_ON_EXEC` nu este partajat (acest flag nu este ținut în structura menționată mai sus).

Moștenirea descriptorilor de fișier după operații fork/exec

Descriptorii de fișier ai procesului părinte se **moștenesc** în procesul copil în urma apelului `fork`. După un apel `exec`, descriptorii de fișier sunt păstrați, excepție făcând cei care au flag-ul `CLOSE_ON_EXEC` setat.

Variabile de mediu în Linux

În cadrul unui program se pot accesa variabilele de mediu, prin evidențierea celui de-al treilea parametru (opțional) al funcției `main`, ca în exemplul următor:

```
int main(int argc, char **argv, char **environ)
```

Acesta desemnează un vector de pointeri la șiruri de caractere, ce conțin variabilele de mediu și valorile lor. Șirurile de caractere sunt de forma `VARIABILA=VALOARE`. Vectorul e terminat cu `NULL`.

[getenv](#) întoarce valoarea variabilei de mediu denumite `name`, sau `NULL` dacă nu există o variabilă de mediu denumită astfel:

```
char* getenv(const char *name);
```

[setenv](#) adaugă în mediu variabila cu numele `name` (dacă nu există deja) și îi setează valoarea la `value`. Dacă variabila există și `replace` e `0`, acțiunea de setare a valorii variabilei e ignorată; dacă `replace` e diferit de `0`, valoarea variabilei devine `value`:

```
int setenv(const char *name, const char *value, int replace);
```

[unsetenv](#) șterge din mediu variabila denumită `name`:

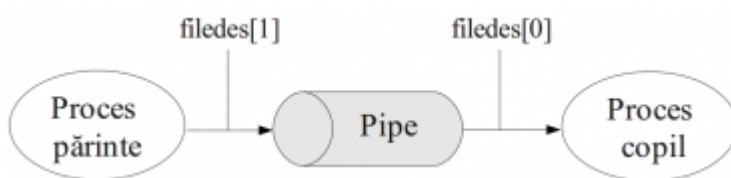
```
int unsetenv(const char *name);
```

Pipe-uri în Linux

Pipe-uri anonime în Linux

Pipe-ul este un mecanism de comunicare unidirecțională între două procese. În majoritatea implementărilor de UNIX, un pipe apare ca o zonă de memorie de o anumită dimensiune în spațiul nucleului. Procesele care comunică printr-un pipe anonim trebuie să aibă un grad de rudenie; de obicei, un proces care creează un pipe va apela după aceea `fork`, iar pipe-ul se va folosi pentru comunicarea între părinte și fiu. În orice caz, procesele care comunică prin pipe-uri anonime nu pot fi create de utilizatori diferiți ai sistemului.

Apelul de sistem pentru creare este [pipe](#):



```
int pipe(int filedes[2]);
```

Vectorul `filedes` conține după execuția funcției 2 descriptori de fișier:

- `filedes[0]`, deschis pentru citire;
- `filedes[1]`, deschis pentru scriere.

Mnemotehnică: `STDIN_FILENO` este 0 (citire), `STDOUT_FILENO` este 1 (scriere).

Observații:

- citirea/scrierea din/în pipe-uri este atomică dacă nu se citesc/scriu mai mult de `PIPE_BUF` octeți.
- citirea/scrierea din/în pipe-uri se realizează cu ajutorul funcțiilor `read/write`.

Majoritatea aplicațiilor care folosesc pipe-uri închid în fiecare dintre procese capătul de pipe **neutilizat** în comunicarea unidirecțională. Dacă unul dintre descriptori este închis se aplică regulile:

- o citire dintr-un pipe pentru care descriptorul de **scriere** a fost închis, după ce toate datele au fost citite, va returna `0`, ceea ce indică sfârșitul fișierului. Descriptorul de scriere poate fi duplicat astfel încât mai multe procese să poată scrie în pipe. De regulă, în cazul pipe-urilor anonime există doar două procese, unul care scrie și altul care citește, pe când în cazul fișierelor pipe cu nume (FIFO) pot exista mai multe procese care scriu date.
- o scriere într-un pipe pentru care descriptorul de **citire** a fost închis cauzează generarea semnalului `SIGPIPE`. Dacă semnalul este captat și se revine din rutina de tratare, funcția de sistem `write` returnează eroare și variabila `errno` are valoarea `EPIPE`.

Atentie! Cea mai frecventă **greșeală**, relativ la lucrul cu pipe-urile, provine din neglijarea faptului că nu se trimite `EOF` prin pipe (citirea din pipe nu se termină) decât dacă sunt închise **TOATE** capetele de scriere din **TOATE** procesele care au deschis descriptorul de scriere în pipe (în cazul unui `fork`, nu uitați să închideți capetele pipe-ului în procesul părinte).

Alte funcții utile: [popen](#), [pclose](#).

Pipe-uri cu nume în Linux

Elimină necesitatea ca procesele care comunică să fie înrudite. Astfel, fiecare proces își poate deschide pentru citire sau scriere fișierul pipe cu nume (FIFO), un tip de fișier special, care păstrează în spate caracteristicile unui pipe. Comunicația se face într-un sens sau în ambele sensuri. Fișierele de tip FIFO pot fi identificate prin litera p în primul câmp al drepturilor de acces (`ls -l`).

Apelul de bibliotecă pentru crearea pipe-urilor de tip FIFO este [mkfifo](#):

```
int mkfifo(const char *pathname, mode_t mode);
```

După ce pipe-ul FIFO a fost creat, acestuia i se pot aplica toate funcțiile pentru operații obișnuite pentru lucrul cu fișiere: `open`, `close`, `read`, `write`.

Modul de comportare al unui pipe FIFO după deschidere este afectat de flagul `O_NONBLOCK`:

Atunci când se închide ultimul descriptor de fișier al capătului de scriere pentru un FIFO, se generează un "sfârșit de fișier" ? EOF ? pentru procesul care citește din FIFO.

Debanarea unui proces

Informații suplimentare legate de depanarea unui proces se găsesc [aici](#)

Procese în Windows

Crearea unui proces

În Windows, atât crearea unui nou proces, cât și înlocuirea imaginii lui cu cea dintr-un program executabil se realizează prin apelul funcției [CreateProcess](#).

<pre> BOOL CreateProcess(LPCTSTR lpApplicationName, LPCTSTR lpCommandLine, LPSECURITY_ATTRIBUTES lpProcessAttributes, LPSECURITY_ATTRIBUTES lpThreadAttributes, BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment, LPCTSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupInfo, LPPROCESS_INFORMATION lpProcessInformation); </pre>	<pre> BOOL bRes = CreateProcess(NULL, // No module name "notepad.exe" // Command line NULL, // Process handle not inheritable NULL, // Thread handle not inheritable FALSE, // Set handle inheritance to false 0, // No creation flags NULL, // Use parent's environment block NULL, // Use parent's starting directory &si, // Pointer to STARTUPINFO structure &pi // Pointer to PROCESS_INFORMATION); // structure </pre>
--	--

API-ul Windows mai pune la dispoziție câteva funcții înrudite precum [CreateProcessAsUser](#), [CreateProcessWithLogonW](#) ori [CreateProcessWithTokenW](#), care permit crearea unui proces într-un context de securitate diferit de cel al utilizatorului curent.

Pentru a se obține un handle al unui proces, cunoscându-se PID-ul procesului respectiv, se va apela funcția [OpenProcess](#):

```
HANDLE OpenProcess(
```

```

    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);

```

iar pentru a obține un handle al procesului curent se va apela [GetCurrentProcess](#):

```
HANDLE GetCurrentProcess(void);
```

Pentru a obține PID-ul procesului curent se va apela [GetCurrentProcessId](#):

```
DWORD GetCurrentProcessId(void);
```

Spre deosebire de Linux, în Windows nu se impune o ierarhie a proceselor în sistem. Teoretic există o ierarhie implicită din modul cum sunt create procesele. Un proces deține handle-uri ale proceselor create de el, însă handle-urile pot fi duplicate între procese ceea ce duce la situația în care un proces deține handle-uri ale unor procese care nu sunt create de el, deci ierarhia implicită dispare.

Așteptarea inițializării procesului creat

Așteptarea terminării unui proces

Pentru a suspenda execuția procesului curent până când unul sau mai multe alte procese se termină, se va folosi una din funcțiile de așteptare [WaitForSingleObject](#) ori [WaitForMultipleObjects](#).

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

Exemplul următor așteaptă nedefinit terminarea procesului reprezentat de `hProcess`.

```

DWORD dwRes = WaitForSingleObject(hProcess, INFINITE);
if (dwRes == WAIT_FAILED)
    // handle error

```

Funcțiile de așteptare sunt folosite în cadrul mai general al mecanismelor de sincronizare între procese și vor fi prezentate în detaliu în [laboratorul de sincronizare între procese](#).

Aflarea codului de terminare a procesului așteptat

Pentru a determina codul de eroare cu care s-a terminat un anumit proces, se va apela funcția [GetExitCodeProcess](#):

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);
```

Dacă procesul `hProcess` nu s-a terminat încă, funcția va întoarce în `lpExitCode` codul de terminare `STILL_ACTIVE`. Dacă procesul s-a terminat, se va întoarce codul său de terminare care poate fi:

- parametrul transmis uneia din funcțiile [ExitProcess](#) sau [TerminateProcess](#) (exit din libc)
- valoarea returnată de funcția `main` sau `WinMain` a procesului
- codul de eroare al unei excepții netratate care a cauzat terminarea procesului
- dacă procesul se termină cu succes valoarea întoarsa în `lpExitCode` va fi 0 sau 1 în caz de eroare.

Terminarea unui proces

Pentru terminarea procesului curent, Windows API pune la dispoziție funcția [ExitProcess](#).

```
void ExitProcess(UINT uExitCode);
```

Procesul apelant și toate firele sale de execuție se vor termina imediat. Toate DLL-urile de care era atașat procesul sunt notificate și se apelează metode de distrugere a resurselor alocate de acestea în spațiul de adresă al procesului. Toți descriptorii de resurse (handle) ai procesului sunt închiși.

Atenție! `ExitProcess` nu se ocupă de eliberarea resurselor bibliotecii standard C. Pentru a asigura o finalizare corectă a programului trebuie apelat `exit`.

Pentru terminarea unui alt proces din sistem se va apela funcția [TerminateProcess](#).

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Se va iniția terminarea procesului `hProcess` și a tuturor firelor sale de execuție și se vor revoca operațiile de intrare/ieșire neterminate după care funcția [TerminateProcess](#) va întoarce imediat. Toți descriptorii de resurse (handle) ai procesului sunt închiși. Funcția [TerminateProcess](#) este **periculoasă** și se recomandă folosirea ei doar în cazuri extreme, deoarece ea nu notifică DLL-urile de care este atașat procesul `hProcess` asupra detașării acestuia, lăsând astfel alocate eventualele date rezervate de DLL în spațiul de adrese al procesului.

Terminarea unui proces NU implică terminarea proceselor create de acesta.

Exemplu

[exec.c](#)

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "utils.h"

void CloseProcess(LPPROCESS_INFORMATION lppi) {
    CloseHandle(lppi->hThread);
    CloseHandle(lppi->hProcess);
}

int main(void)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD dwRes;
    BOOL bRes;
    CHAR cmdLine[] = "mspaint";

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* Start child process */
    bRes = CreateProcess(
        NULL,          /* No module name (use command line) */
        cmdLine,      /* Command line */
        NULL,         /* Process handle not inheritable */
        NULL,         /* Thread handle not inheritable */
        FALSE,        /* Set handle inheritance to FALSE */
        0,            /* No creation flags */
        NULL,         /* Use parent's environment block */
        NULL,         /* Use parent's starting directory */
        &si,          /* Pointer to STARTUPINFO structure */
        &pi           /* Pointer to PROCESS_INFORMATION structure */
    );
    DIE(bRes == FALSE, "CreateProcess");

    /* Wait for the child to finish */
    dwRes = WaitForSingleObject(pi.hProcess, INFINITE);
    DIE(dwRes == WAIT_FAILED, "WaitForSingleObject");

    bRes = GetExitCodeProcess(pi.hProcess, &dwRes);
    DIE(bRes == FALSE, "GetExitCode");

    return 0;
}
```

Moștenirea handle-urilor la CreateProcess

După un apel [CreateProcess](#), handle-urile din procesul părinte pot fi **moștenite** în procesul copil.

Atenție! Pentru ca un handle să poată fi moștenit în procesul creat, trebuie îndeplinite 2 condiții:

- membrul `bInheritHandle`, al structurii `SECURITY_ATTRIBUTES`, transmise lui [CreateFile](#), trebuie să fie `TRUE`
- parametrul `bInheritHandles`, al lui [CreateProcess](#), trebuie să fie `TRUE`.

Handle-urile moștenite sunt valide doar în contextul procesului copil.

Cei 3 descriptori speciali de fișier pot fi obținuți apelând funcția [GetStdHandle](#):

```
HANDLE GetStdHandle(DWORD nStdHandle);
```

cu unul din parametrii:

- `STD_INPUT_HANDLE`

- `STD_OUTPUT_HANDLE`
- `STD_ERROR_HANDLE`

Pentru **redirectarea** handle-urilor standard în procesul copil puteți folosi membrii `hStdInput`, `hStdOutput`, `hStdError` ai structurii [STARTUPINFO](#), transmise lui [CreateProcess](#). În acest caz, membrul `dwFlags` al aceleiași structurii trebuie setat la `STARTF_USESTDHANDLES`. Dacă se dorește ca anumite handle-uri să rămână implicite, li se poate atribui handle-ul întors de [GetStdHandle](#).

```
STARTUPINFO si;
...
/* initialize process startup info structure */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);

/* setup flags to allow handle inheritance (redirection) */
si.dwFlags |= STARTF_USESTDHANDLES;
```

Atenție! Pentru a realiza redirectarea corespunzător câmpurile `hStdInput`, `hStdOutput`, `hStdError` din structura `STARTUPINFO` trebuie inițializate.

Alte proprietăți ale procesului părinte care pot fi moștenite sunt variabilele de mediu și directorul curent. Nu vor fi moștenite handle-uri ale unor zone de memorie alocate de procesul părinte și nici pseudo-descriptori precum cei întorși de funcția [GetCurrentProcess](#).

Handle-ul din procesul părinte și cel moștenit în procesul copil vor referi același obiect, exact ca în cazul duplicării. De asemenea, handle-ul moștenit în procesul copil are aceeași valoare și aceeași drepturi de acces ca și handle-ul din procesul părinte. Pentru a folosi handle-ul moștenit, procesul copil va trebui să-i cunoască valoarea și ce obiect referă. Aceste informații trebuie să fie pasate de părinte printr-un mecanism extern (IPC etc).

Variabile de mediu în Windows

Pentru a afla valoarea unei variabile de mediu se va apela funcția [GetEnvironmentVariable](#):

```
DWORD GetEnvironmentVariable(
    LPCTSTR lpName,
    LPTSTR lpBuffer,
    DWORD nSize
);
```

care va umple `lpBuffer`, de dimensiune `nSize`, cu valoarea variabilei `lpName`.

Pentru a seta o variabilă de mediu se va apela [SetEnvironmentVariable](#):

```
BOOL SetEnvironmentVariable(
    LPCTSTR lpName,
    LPCTSTR lpValue
);
```

care va seta variabila `lpName` la valoarea specificată de `lpValue`. Funcția se va folosi și pentru ștergerea unei variabile de mediu prin transmiterea unui parametru `lpValue = NULL`. [SetEnvironmentVariable](#) are efect doar asupra variabilelor de mediu ale utilizatorului și nu poate modifica variabile de mediu globale.

În Windows există un set de variabile de mediu globale, valabile pentru toți utilizatorii. În plus, fiecare utilizator în parte are asociat un set propriu de variabile de mediu. Împreună, cele două seturi formează `Environment Block`-ul utilizatorului respectiv. Acest `Environment Block` este similar cu variabila `environ`, din Linux. Mai multe detalii aici:

Pipe-uri in Windows

Pipe-uri anonime în Windows

Ca și pe Linux, pipe-urile anonime de pe Windows sunt unidirecționale. Fiecare pipe are două capete reprezentate de câte un handle: un handle de citire și un handle de scriere. Funcția de creare a unui pipe este [CreatePipe](#):

<pre> BOOL CreatePipe(PHANDLE hReadPipe, PHANDLE hWritePipe, LPSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize); </pre>	<pre> CreatePipe(&hReadPipe, &hWritePipe, &sa, //pentru moștenire sa.bInheritHandle=TRUE 0 //dimensiunea default pentru pipe); </pre>
--	---

Atenție! Pentru a **moșteni** un pipe anonim, este nevoie ca parametrul `bInheritHandle` din structura [LPSECURITY_ATTRIBUTES](#) să fie setat pe `TRUE`.

`CreatePipe` creează atât pipe-ul, cât și handler-urile folosite pentru scriere/citire din/în pipe cu ajutorul funcțiilor [ReadFile](#) și [WriteFile](#).

[ReadFile](#) se termină în unul din cazurile: o operație de scriere a luat sfârșit la capătul de scriere în pipe, numărul de octeți cerut a fost citit sau a apărut o eroare.

[WriteFile](#) se termină atunci când toți octeții au fost scriși. Dacă bufferul pipe-ului este plin înainte ca toți octeții să fie scriși, [WriteFile](#) rămâne blocat până când alt proces sau thread folosește [ReadFile](#) pentru a face loc în buffer.

Pipe-urile anonime sunt implementate folosind un pipe cu nume unic. De aceea se poate pasa un handle al unui pipe anonim unei funcții care cere un handle al unui pipe cu nume.

Pipe-uri cu nume în Windows

În Windows, un pipe cu nume este un pipe unidirecțional (inbound ori outbound) sau bidirecțional ce realizează comunicația între un server pipe și unul sau mai mulți clienți pipe. Se numește *server pipe* procesul care creează un pipe cu nume și *client pipe* procesul care se conectează la pipe.

Pentru a face posibilă comunicarea între server și mai mulți clienți prin același pipe, se folosesc instanțe ale pipe-ului. O instanță a unui pipe folosește același nume, dar are propriile handle-uri și buffere.

Pipe-urile cu nume au următoarele caracteristici care le diferențiază de cele anonime:

- sunt orientate pe mesaje - se pot transmite mesaje de lungime variabilă (nu numai byte stream);
- sunt bidirecționale - două procese pot schimba mesaje pe același pipe;
- pot exista mai multe instanțe ale aceluiași pipe
- poate fi accesat din rețea - comunicația între două procese aflate pe mașini diferite este aceeași cu cea între procese aflate pe aceeași mașină.

Mod de lucru - Server Pipe

Serverul creează un pipe cu funcția [CreateNamedPipe](#).

<pre>HANDLE CreateNamedPipe(LPCTSTR lpName, DWORD dwOpenMode, DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOutBufferSize, DWORD nInBufferSize, DWORD nDefaultTimeOut, LPSECURITY_ATTRIBUTES lpSecurityAttributes);</pre>	<pre>HANDLE hNamedPipe = CreateNamedPipe("\\.\pipe\mypipe", // name PIPE_ACCESS_DUPLEX, // read/write access PIPE_TYPE_BYTE PIPE_WAIT, // byte stream PIPE_UNLIMITED_INSTANCES, // max. instances BUFSIZE, // output buffer size BUFSIZE, // input buffer size 0, // default time out NULL // default security // attribute);</pre>
--	---

Funcția returnează un handle către capătul serverului la pipe. Acest handle poate fi transmis funcției [ConnectNamedPipe](#) pentru a aștepta conectarea unui proces client la o instanță a unui pipe.

```
BOOL ConnectNamedPipe(HANDLE hNamedPipe, LPOVERLAPPED lpOverlapped);
```

Mod de lucru - Client Pipe

Un client se conectează transmițând numele pipe-ului la una din funcțiile [CreateFile](#) sau [CallNamedPipe](#) - ultima mai utilă pentru transmiterea de mesaje.

Un exemplu funcțional folosind pipe-uri cu nume se află [aici](#)

Mai multe detalii despre moștenirea pipe-urilor se pot găsi [aici](#).

Detalii extra despre pipe-urile cu nume

Exerciții

În rezolvarea laboratorului folosiți arhiva de sarcini [lab03-tasks.zip](#)

Observații: Pentru a vă ajuta la implementarea exercițiilor din laborator, în directorul `utils` din arhivă există un fișier `utils.h` cu funcții utile.

LINUX

- (1.5 puncte)** Intrați în directorul `1-system`
 - Programul `system.c` execută o comandă transmisă ca parametru, folosind funcția de bibliotecă [system](#).
 - Compilați (folosind `make`)
 - Rulați programul dând ca parametru o comandă. Exemplu: `./system pwd`
 - Cum procedați pentru a trimite mai mulți parametrii unei comenzi? (ex: `ls -la`)
 - Cum funcționează [system](#)?
 - Pentru a vedea câte apeluri de sistem [execve](#) se realizează, rulați: `strace -e execve -f ./system ls`
 - Explicați apariția fiecărui apel [execve](#)
 - Hints:
 - [strace](#) afișează apelurile de sistem făcute de un executabil. De ce este nevoie de parametrul `-f` ?
 - Revedeți secțiunea [Înlocuirea imaginii unui proces](#) și pagina de manual pentru [execve](#)
- (1 punct)** Intrați în directorul `2-orphan`
 - Inspectați sursa `orphan.c`.
 - Compilați programul.
 - Rulați: `./orphan & watch ps -al`
 - De ce, pentru procesul indicat de executabilul `orphan` (coloana `CMD`), `pid`-ul procesului părinte (coloana `PPID`) devine `1`?
- Tiny-Shell**
 - Intrați în directorul `3-tiny`
 - Următoarele exerciții au ca scop implementarea unui shell minimal, care oferă suport pentru execuția unei *singure* comenzi externe cu argumente multiple și redirectări. Shell-ul trebuie să ofere suport pentru folosirea și setarea variabilelor de mediu.
 - Observație:* Pentru a ieși din tiny shell folosiți `exit` sau `CTRL+D`
- (1 punct)** Execuția unei comenzi simple
 - Creați un nou proces care să execute o comandă simplă.
 - Funcția `simple_cmd` primește ca argument un vector de șiruri ce conține comanda și parametrii acesteia
 - Hints:
 - Citiți exemplul [my_system](#).

- Urmăriți în cod comentariile cu **TODO-1**
- Pentru testare puteți folosi comenzile: `./tiny`
 - > `pwd`
 - > `ls -al`
 - > `exit`
- 2. (**1 punct**) Adăugare suport pentru setarea și expandarea variabilelor de mediu
 - Trebuie să completați funcțiile `set_var` și `expand`; acestea sunt apelate deja atunci când se face parsarea liniei de comandă.
 - Verificarea erorilor trebuie făcută în aceste funcții.
 - Hints:
 - Urmăriți în cod comentariile cu **TODO-2**
 - Citiți secțiunea [Variabile de mediu in Linux](#)
 - Pentru testare puteți folosi comenzile: `./tiny`
 - > `echo $HOME`
 - > `name=Makefile`
 - > `cat $name`
- 3. (**1 punct**) Redirectarea ieșirii standard
 - Tiny-shell trebuie să suporte redirectarea output-ului unei comenzi (`stdout`) într-un fișier.
 - Completați funcția `do_redirect`.
 - Dacă fișierul indicat de `filename` nu există, va fi creat. Dacă există, trebuie trunchiat.
 - Hints:
 - Urmăriți în cod comentariile cu **TODO-3**
 - Citiți secțiunea [Copierea descriptorilor de fișier](#)
 - Pentru testare puteți folosi comenzile: `./tiny`
 - > `ls -al > out`

WINDOWS

1. (**0.5 puncte**) Bomb
 - Deschideți proiectul (fișierul `.sln`) și compilați primul subproiect: `1-bomb`
 - Inspectați sursa `1-bomb.c`. Ce credeți că face?
 - Înainte de a rula `1-bomb.exe`, porniți Task Manager (CTRL + ALT + DEL; În mașina virtuală se va folosi CTRL + ALT + Insert)
 - Rulați `1-bomb.exe`. Ce s-a întâmplat?
2. **Tiny-Shell on Windows**
 - Ne propunem să continuăm implementarea de **Tiny-Shell**
 - **Important:** Compilarea se va realiza din Visual Studio sau din command-prompt-ul de Visual Studio, iar rularea executabilului `tiny.exe` se va realiza din **Cygwin**.
 - Pentru a ajunge din Cygwin pe Desktop: `$ cd c:`
`$ cd Users/Student/Desktop/`
1. (**0.5 puncte**) Executarea unei comenzi simple
 - Partea de executare a unei comenzi simple și a variabilelor de mediu este deja implementată.
 - Urmăriți în sursă funcția `RunSimpleCommand`.
 - Testați funcționalitatea prin comenzi de tipul: `./tiny`
 - > `ls -al`
 - > `exit`
2. (**1.5 puncte**) Redirectare
 - Realizați redirectarea tuturor `HANDLE`-relor
 - Hints:
 - Completați funcția `RedirectHandle`.
 - **Atenție!** Trebuie inițializate toate handle-rele structurii `STARTUPINFO`.
 - Urmăriți în cod comentariile cu **TODO 1**
 - Revedeți secțiunea [Moștenirea handle-urilor](#).
 - Atenție la metoda de moștenire a handle-relor.
 - Pentru testare puteți folosi comenzile: `./tiny`
 - > `ls -al > out`
 - > `exit`
3. (**2 puncte**) Implementarea unei comenzi cu pipe-uri
 - Shell-ul vostru trebuie să ofere suport pentru o comandă de forma `' comanda_simpla | comanda_simpla '`
 - Hints:
 - Urmăriți în cod comentariile cu **TODO 2**
 - Completați funcția `PipeCommands`.
 - **Atenție!** În procesul părinte, trebuie închise capetele pipe-urilor.
 - Pentru redirectari, folosiți-vă de funcția `RedirectHandle`.

- Revedeți secțiunea despre [Pipe-uri anonime în Windows](#)
- Pentru testare puteți folosi comenzile: `./tiny`
 - > `cat Makefile | grep tiny`
 - > `exit`

BONUS

1. **(1 so karma)** Pipe-uri cu nume (Linux/Windows)
 - Realizați două programe, denumite `server` și `client`, care interacționează printr-un **pipe cu nume**.
 - FIFOul se numește `myfifo`. Dacă nu există, este creat de server.
 - Serverul trebuie rulat înaintea clientului.
 - Clientul citește de la intrarea standard un mesaj care va fi transmis serverului.
 - Serverul va afișa mesajul primit la ieșirea standard.
 - **Hints:**
 - Linux:
 - Citiți secțiunea [Pipe-uri cu nume în Linux](#)
 - Windows:
 - Citiți secțiunea [Pipe-uri cu nume în Windows](#)
 - Puteți porni de la [exemplul](#) din documentația `CreateNamedPipe`.
 - **Atenție:** Dacă `ReadFile` întoarce `FALSE`, iar mesajul de eroare (ce întoarce `GetLastError()`) este `ERROR_BROKEN_PIPE`, înseamnă că au fost închise toate capetele de scriere.
2. **(1 so karma)** Magic
 - Intrați în directorul `lin/5-magic` și deschideți sursa `magic.c`
 - Completați **doar** condiția instrucțiunii `if` pentru a obține la rulare mesajul "Hello World".
 - Nu sunt permise alte modificări în funcția `main`.

Soluții

[lab03-sol.zip](#)

Resurse utile

1. [Fork - Wikipedia](#)
2. [About Fork and Exec](#)
3. [Fork, Exec and Process Control - YoLinux Tutorial](#)
4. [Windows Handles and Data Types - Wikipedia](#)
5. [MSDN: Processes and Threads](#)
6. [C++ CreateProcess example](#)
7. [Windows XP and 2003 Server Boot.ini options](#)

¹ limită globală setată implicit pe Linux la 4096 bytes

From:

<http://elf.cs.pub.ro/so/wiki/> - **Sisteme de Operare**

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-03>

Last update: **2012/03/20 23:30**