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 execuţie. Procesele sunt unitatea primitivă prin care sistemul de operare alocă resurse utilizatorilor. Orice proces are un spaţiu de adrese şi unul sau mai multe fire de execuţie. Putem avea mai multe procese ce execută acelaşi program, dar oricare două procese sunt complet independente.

Informaţiile despre procese sunt ţinute într-o structură numită `Process Control Block` (PCB), câte una pentru fiecare proces existent în sistem. Printre cele mai importante informaţii regăsim:

- PID - identificatorul procesului
- spaţiu de adresă
- registre generale, `PC` (contor program), `SP` (indicator stivă)
- tabela de fişiere deschise
- informaţii referitoare la semnale
  - lista de semnale blocate, ignorate sau care aşteaptă să fie trimise procesului
  - handler-ele de semnale
- informaţiile referitoare la sistemele de fişiere (directorul rădăcină, directorul curent)

În momentul lansării în execuţie 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 dispoziţie apeluri de sistem pentru lucrul cu procese: creare, terminare, aşteptarea terminării. Totodată există apeluri pentru duplicarea descriptorilor de resurse între procese, ori închiderea acestor descriptori.

Procesele 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 comunicaţie) sunt mecanisme primitive de comunicare între procese. Un pipe poate conţine 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 operaţiile de citire şi 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 şi un copil sau doi copii), deoarece sunt accesibile doar prin moştenire. Aceste pipe-uri nu mai există după ce procesele şi-au terminat execuţia.
- pipe-uri **cu nume**: au suport fizic - există ca fişiere cu drepturi de acces. Aceasta înseamnă că ele vor exista independent de procesul care le creează şi pot fi folosite de procese neînrudite.

Procese în Linux

Lansarea în execuţie a unui program presupune următorii paşi:

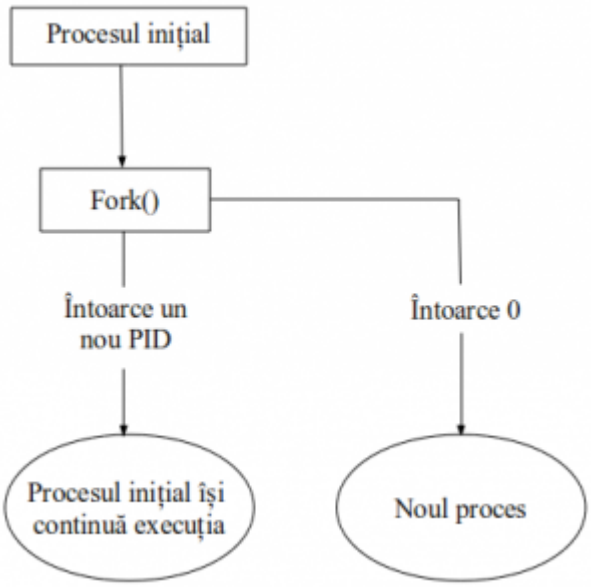
- Se creează un nou proces cu `fork` - procesul copil are o copie a resurselor procesului părinte.
- Dacă se doreşte înlocuirea imaginii procesului copil aceasta poate fi schimbată prin apelarea unei funcţii din familia `exec*`.

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 primeşte un nou *process id* (`PID`) de la sistemul de operare.



Atenţie! - acestă funcţie este apelată o dată şi se întoarce (în caz de succes) de două ori, aşa cum se poate vedea în figura din dreapta.

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

Funcţia getpid întoarce `PID`-ul procesului apelant:

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

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

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

Înlocuirea imaginii unui proces

Familia de funcţii exec va executa un nou program, înlocuind imaginea procesului curent, cu cea dintr-un fişier (executabil). Spaţiul de adrese al procesului va fi înlocuit cu unul nou, creat special pentru execuţia fişierului. De asemenea vor fi reiniţializate registrele `IP` (contorul program) şi `SP` (indicatorul stivă) şi registrele generale. Măştile de semnale ignorate şi blocate sunt setate la valorile implicite, ca şi handler-ele semnalelor. `PID`-ul şi descriptorii de fişiere care nu au setat flag-ul `CLOSE_ON_EXEC` rămân neschimbaţi (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*).

`execl` şi `execv` nu caută programul dat ca parametru în `PATH`, astfel că acesta trebuie însoţit de calea completă. Versiunile `execlp` şi `execvp` caută programul şi în `PATH`.

Toate funcţiile `exec*` sunt implementate prin apelul de sistem execve

Aşteptarea terminării unui proces

Familia de funcţii wait suspendă execuţia procesului apelant până când procesul (procesele) specificate în argumente fie s-au terminat, fie au fost oprite (`SIGSTOP`).

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

Starea procesului interogat se poate afla examinând `status` cu macrodefiniţii precum WEXITSTATUS, care întoarce codul de eroare cu care s-a încheiat procesul aşteptat, evaluând cei mai nesemnificativi 8 biţi.

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

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

În caz că se doreşte doar aşteptarea terminării procesului copil, nu şi examinarea statusului:

```
wait(NULL);
```

Terminarea unui proces

Pentru terminarea procesului curent, Linux pune la dispoziţie 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)` şi `_Exit(2)` sunt funcţional echivalente (doar că sunt definite de standarde diferite):

- procesul apelant se va termina imediat
- toţi descriptorii de fişier ai procesului sunt închişi
- copiii procesului sunt "înfiaţi" de `init`
- părintelui procesului îi va fi trimis un semnal `SIGCHLD`. Tot acestuia îi va fi întoarsă valoarea `status`, ca rezultat al unei funcţii de aşteptare (`wait` sau `waitpid`).

În plus, `exit(3)`:

- va şterge toate fişierele create cu `tmpfile()`
- va scrie bufferele streamurilor deschise şi le va închide

Notă: Conform `ISO C`, un program care se termină cu `return x` din `main()` va avea acelaşi 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` şi 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 fişier

dup duplică descriptorul de fişier `oldfd` şi întoarce noul descriptor de fişier, sau `-1` în caz de eroare:

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

dup2 duplică descriptorul de fişier `oldfd` în descriptorul de fişier `newfd`; dacă `newfd` există, mai întâi va fi închis. Întoarce noul descriptor de fişier, sau `-1` în caz de eroare:

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

Descriptorii de fişier sunt, de fapt, indecşi în tabela de fişiere deschise. Tabela este populată cu pointeri către structuri cu informaţiile despre fişiere. Duplicarea unui descriptor de fişier înseamnă duplicarea intrării din tabela de fişiere deschise (adică 2 pointeri de la poziţii diferite din tabelă vor indica spre aceeaşi structură din sistem, asociată fişierului). Din acest motiv, toate informaţiile asociate unui fişier (lock-uri, cursor, flag-uri) sunt **partajate** de cei doi file descriptori. Aceasta înseamnă că operaţiile ce modifică aceste informaţii pe unul din file descriptori (de ex. `lseek`) sunt vizibile şi pentru celălalt file descriptor (duplicat). **Atentie!** Există şi o excepţie: flag-ul `CLOSE_ON_EXEC` nu este partajat (acest flag nu este ţinut în structura menţionată mai sus).

Moştenirea descriptorilor de fişier după operaţii fork/exec

Descriptorii de fişier ai procesului părinte se **moştenesc** în procesul copil în urma apelului `fork`. După un apel `exec`, descriptorii de fişier sunt păstraţi, excepţie 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 evidenţierea celui de-al treilea parametru (opţional) al funcţiei `main`, ca în exemplul următor:

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

Acesta desemnează un vector de pointeri la şiruri de caractere, ce conţin variabilele de mediu şi valorile lor. Şirurile 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) şi îi setează valoarea la `value`. Dacă variabila există şi `replace` e `0`, acţiunea 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 şterge din mediu variabila denumită `name`:

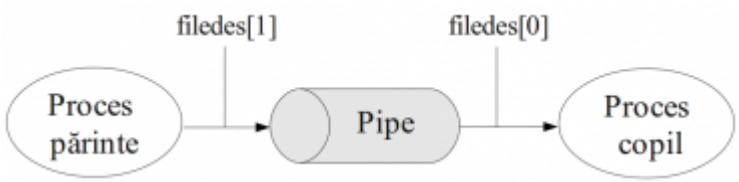
```
int unsetenv(const char *name);
```

Pipe-uri în Linux

Pipe-uri anonime în Linux

Pipe-ul este un mecanism de comunicare unidirecţională între două procese. În majoritatea implementărilor de UNIX, un pipe apare ca o zonă de memorie de o anumită dimensiune în spaţiul 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 şi fiu. În orice caz, procesele care comunică prin pipe-uri anonime nu pot fi create de utilizatori diferiţi ai sistemului.

Apelul de sistem pentru creare este pipe:




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

Vectorul `filedes` conţine după execuţia funcţiei 2 descriptori de fişier:

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

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

Observaţii:

- citirea/scrierea din/în pipe-uri este atomică dacă nu se citesc/scriu mai mult de `PIPE_BUF`[1] octeţi.
- citirea/scrierea din/în pipe-uri se realizează cu ajutorul funcţiilor `read`/`write`.

Majoritatea aplicaţiilor care folosesc pipe-uri închid în fiecare dintre procese capătul de pipe **neutilizat** în comunicarea unidirecţională. 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 sfârşitul fişierului. 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 şi altul care citeşte, pe când în cazul fişierelor 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 şi se revine din rutina de tratare, funcţia de sistem `write` returnează eroare şi variabila `errno` are valoarea `EPIPE`.

Atentie! Cea mai frecventă **greşeală**, 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 uitaţi să închideţi capetele pipe-ului în procesul părinte).

Alte funcţii utile: popen, pclose.

Pipe-uri cu nume în Linux

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

Apelul de sistem 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 funcţiile pentru operaţii obişnuite pentru lucrul cu fişiere: `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 fişier al capătului de scriere pentru un FIFO, se generează un ?sfârşit de fişier" ? `EOF` ? pentru procesul care citeşte din FIFO.

Depanarea unui proces

Informaţii suplimentare legate de depanarea unui proces se g?sesc aici

Procese în Windows

Crearea unui proces

În Windows, atât crearea unui nou proces, cât şi înlocuirea imaginii lui cu cea dintr-un program executabil se realizează prin apelul funcţiei CreateProcess.

```
BOOL CreateProcess(
     LPCTSTR               lpApplicationName,
     LPTSTR                lpCommandLine,
     LPSECURITY_ATTRIBUTES lpProcessAttributes,
     LPSECURITY_ATTRIBUTES lpThreadAttributes,
     BOOL                  bInheritHandles,
     DWORD                 dwCreationFlags,
     LPVOID                lpEnvironment,
     LPCTSTR               lpCurrentDirectory,
     LPSTARTUPINFO         lpStartupInfo,
     LPPROCESS_INFORMATION lpProcessInformation
);
```

```
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
```

API-ul Windows mai pune la dispoziţie câteva funcţii 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 obţine un handle al unui proces, cunoscându-se `PID`-ul procesului respectiv, se va apela funcţia OpenProcess:

```
HANDLE OpenProcess(
```

```
        DWORD dwDesiredAccess,
        BOOL  bInheritHandle,
        DWORD dwProcessId
);
```

iar pentru a obţine un handle al procesului curent se va apela GetCurrentProcess:

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

Pentru a obţine 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 deţine handle-uri ale proceselor create de el, însă handle-urile pot fi duplicate între procese ceea ce duce la situaţia în care un proces deţine handle-uri ale unor procese care nu sunt create de el, deci ierarhia implicită dispare.

Aşteptarea iniţializării procesului creat

Aşteptarea terminării unui proces

Pentru a suspenda execuţia procesului curent până când unul sau mai multe alte procese se termină, se va folosi una din funcţiile de aşteptare WaitForSingleObject ori WaitForMultipleObjects.

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

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

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

Funcţiile de aşteptare sunt folosite în cadrul mai general al mecanismelor de sincronizare între procese şi vor fi prezentate în detaliu în laboratorul de sincronizare între procese.

Aflarea codului de terminare a procesului aşteptat

Pentru a determina codul de eroare cu care s-a terminat un anumit proces, se va apela funcţia GetExitCodeProcess:

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

Dacă procesul `hProcess` nu s-a terminat încă, funcţia 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 funcţiile ExitProcess sau TerminateProcess (`exit` din `libc`)
- valoarea returnată de funcţia `main` sau `WinMain` a procesului
- codul de eroare al unei excepţii netratate care a cauzat terminarea procesului.

Terminarea unui proces

Pentru terminarea procesului curent, Windows API pune la dispoziţie funcţia ExitProcess.

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

Procesul apelant şi toate firele sale de execuţie se vor termina imediat. Toate `DLL`-urile de care era ataşat procesul sunt notificate şi se apelează metode de distrugere a resurselor alocate de acestea în spaţiul de adresă al procesului. Toţi descriptorii de resurse (handle) ai procesului sunt închişi.

Atenţie! `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 funcţia TerminateProcess.

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

Se va iniţia terminarea procesului `hProcess` şi a tuturor firelor sale de execuţie şi se vor revoca operaţiile de intrare/ieşire neterminate după care funcţia TerminateProcess va întoarce imediat. Toţi descriptorii de resurse (handle) ai procesului sunt închişi. Funcţia TerminateProcess este **periculoasă** şi se recomandă folosirea ei doar în cazuri extreme, deoarece ea nu notifică `DLL`-urile de care este ataşat procesul `hProcess` asupra detaşării acestuia, lăsând astfel alocate eventualele date rezervate de `DLL` în spaţiul 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;
}
```

Moştenirea handle-urilor la CreateProcess

După un apel CreateProcess, handle-urile din procesul părinte pot fi **moştenite** în procesul copil.

Atenţie! Pentru ca un handle să poată fi moştenit în procesul creat, trebuie îndeplinite 2 condiţii:

- membrul `bInheritHandle`, al structurii `SECURITY_ATTRIBUTES`, transmise lui CreateFile, trebuie să fie `TRUE`
- parametrul `bInheritHandles`, al lui CreateProcess, trebuie să fie `TRUE`.

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

Cei 3 descriptori speciali de fişier pot fi obţinuţi apelând funcţia 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 puteţi folosi membrii `hStdInput`, `hStdOutput`, `hStdError` ai structurii STARTUPINFO, transmise lui CreateProcess. În acest caz, membrul `dwFlags` al aceleiaşi structuri trebuie setat la `STARTF_USESTDHANDLES`. Dacă se doreşte 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 inheritence (redirection) */
si.dwFlags |= STARTF_USESTDHANDLES;
```

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

Alte proprietăţi ale procesului părinte care pot fi moştenite sunt variabilele de mediu şi directorul curent. Nu vor fi moştenite handle-uri ale unor zone de memorie alocate de procesul părinte şi nici pseudo-descriptori precum cei întorşi de funcţia GetCurrentProcess.

Handle-ul din procesul părinte şi cel moştenit în procesul copil vor referi acelaşi obiect, exact ca în cazul duplicării. De asemenea, handle-ul moştenit în procesul copil are aceeaşi valoare şi aceleaşi drepturi de acces ca şi handle-ul din procesul părinte. Pentru a folosi handle-ul moştenit, procesul copil va trebui să-i cunoască valoarea şi ce obiect referă. Aceste informaţii 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 funcţia 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`. Funcţia se va folosi şi pentru ştergerea unei variabile de mediu prin transmiterea unui parametru `lpValue = NULL`. SetEnvironmentVariable are efect doar asupra variabilelor de mediu ale utilizatorului şi nu poate modifica variabile de mediu globale.

În Windows există un set de variabile de mediu globale, valabile pentru toţi 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 şi pe Linux, pipe-urile anonime de pe Windows sunt unidirecţionale. Fiecare pipe are două capete reprezentate de câte un handle: un handle de citire şi un handle de scriere. Funcţia de creare a unui pipe este CreatePipe:

```
BOOL CreatePipe(
     PHANDLE                hReadPipe,
     PHANDLE                hWritePipe,
     LPSECURITY_ATTRIBUTES lpPipeAttributes,
     DWORD                  nSize
);
```

```
CreatePipe(
    &hReadPipe,
    &hWritePipe,
    &sa,        //pentru moştenire sa.bInheritHandle=TRUE
    0           //dimensiunea default pentru pipe
);
```

Atenţie! Pentru a **moşteni** un pipe anonim, este nevoie ca parametrul `bInheritHandle` din structura LPSECURITY_ATTRIBUTES să fie setat pe TRUE.

`CreatePipe` creează atât pipe-ul, cât şi handler-urile folosite pentru scriere/citire din/în pipe cu ajutorul funcţiilor ReadFile şi WriteFile.

ReadFile se termină în unul din cazurile: o operaţie de scriere a luat sfârşit la capătul de scriere în pipe, numărul de octeţi cerut a fost citit sau a apărut o eroare.

WriteFile se termină atunci când toţi octeţii au fost scrişi. Dacă bufferul pipe-ului este plin înainte ca toţi octeţii să fie scrişi, WriteFile rămâne blocat pâna ce alt proces sau thread foloseşte 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 funcţii care cere un handle al unui pipe cu nume.

Pipe-uri cu nume în Windows

În Windows un pipe cu nume este un pipe unidirecţional (inbound ori outbound) sau bidirecţional ce realizează comunicaţia între un server pipe şi unul sau mai mulţi clienţi pipe. Se numeşte *server pipe* procesul care creează un pipe cu nume şi *client pipe* procesul care se conectează la pipe.
Pentru a face posibilă comunicarea între server şi mai mulţi clienţi prin acelaşi pipe se folosesc instanţe ale pipe-ului. O instanţă a unui pipe foloseşte acelaşi nume, dar are propriile handle-uri şi buffere.

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

- sunt orientate pe mesaje - se pot transmite mesaje de lungime variabilă (nu numai byte stream);
- sunt bidirecţionale - două procese pot schimba mesaje pe acelaşi pipe;
- pot exista mai multe instanţe ale aceluiaşi pipe - Exemplu (bla bla);
- poate fi accesat din reţea - comunicaţia între două procese aflate pe maşini diferite este aceeaşi cu cea între procese aflate pe aceeaşi maşină.

Mod de lucru - Server Pipe

Serverul creează un pipe cu funcţia CreateNamedPipe.

```
HANDLE CreateNamedPipe(
   LPCTSTR               lpName,
   DWORD                 dwOpenMode,
   DWORD                 dwPipeMode,
   DWORD                 nMaxInstances,
   DWORD                 nOutBufferSize,
   DWORD                 nInBufferSize,
   DWORD                 nDefaultTimeOut,
   LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

```
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
```

Funcţia returnează un handle către capătul serverului la pipe. Acest handle poate fi transmis funcţiei ConnectNamedPipe pentru a aştepta conectarea unui proces client la o instanţă a unui pipe.

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

Mod de lucru - Client Pipe

Un client se conectează transmiţând numele pipe-ului la una din funcţiile CreateFile sau CallNamedPipe - ultima mai utilă pentru transmiterea de mesaje.

Un exemplu funcţional folosind pipe-uri cu nume se află aici

Detalii extra despre pipe-urile cu nume

Exerciţii

În rezolvarea laboratorului folosiţi arhiva de sarcini lab03-tasks.zip

Observaţii: Pentru a vă ajuta la implementarea exerciţiilor din laborator, în directorul `utils` din arhivă există un fişier `utils.h` cu funcţii utile.

LINUX

1. (**1.5 puncte**) Intraţi în directorul `1-system`
   - Programul `system.c` execută o comandă transmisă ca parametru, folosind funcţia de bibliotecă system.
   - Compilaţi (folosiţi `make`) şi rulaţi programul dând ca parametru o comandă. Exemplu ./system pwd
   - Cum procedaţi pentru a trimite mai mulţi parametrii unei comenzi? (ex: `ls -la`)
   - Cum funcţionează system?
     - Pentru a vedea câte apeluri de sistem execve se realizează, rulaţi: strace -e execve -f ./system ls
     - Explicaţi apariţia fiecărui apel execve
     - Hints:
       - strace afişează apelurile de sistem făcute de un executabil. De ce este nevoie de parametrul `-f` ?
       - Revedeţi secţiunea Înlocuirea imaginii unui proces şi pagina de manual pentru execve
2. (**1 punct**) Intraţi în directorul `2-orphan`
   - Inspectaţi sursa `orphan.c` şi apoi compilaţi programul.
   - Rulaţi: ./orphan & watch ps -al
   - De ce, pentru procesul indicat de executabilul `orphan` (coloana `CMD`), pid-ul procesului părinte (coloana `PPID`) devine 1?
3. **Tiny-Shell**
   - Intraţi în directorul `3-tiny`
   - Următoarele exerciţii au ca scop implementarea unui shell minimal, care oferă suport pentru execuţia unei *singure* comenzi externe cu argumente multiple, redirectări şi pipe-uri. Shell-ul trebuie să ofere suport pentru folosirea şi setarea variabilelor de mediu.
   - *Observaţie:* Pentru a ieşi din tiny shell folosiţi `exit` sau `CTRL+D`
   1. (**1 punct**) Execuţia unei comenzi simple
      - Creaţi un nou proces care să execute o comandă simplă.
      - Funcţia `simple_cmd` primeşte ca argument un vector de şiruri ce conţine comanda şi parametrii acesteia
      - Hints:
        - Citiţi exemplul my_system.
        - Urmăriţi în cod comentariile cu `TODO-1`
      - Pentru testare puteţi folosi comenzile: ./tiny
        > pwd
        > ls -al
        > exit
   2. (**1 punct**) Adăugare suport pentru setarea şi expandarea variabilelor de mediu
      - Trebuie să completaţi funcţiile `set_var` şi `expand`; acestea sunt apelate deja atunci când se face parsarea liniei de comandă.
      - Verificarea erorilor trebuie făcută în aceaste funcţii.
      - Hints:
        - Urmăriţi în cod comentariile cu `TODO-2`
        - Citiţi secţiunea Variabile de mediu in Linux
      - Pentru testare puteţi folosi comenzile: ./tiny
        > echo $HOME
        > name=Makefile
        > cat $name
   3. (**1 punct**) Redirectarea ieşirii standard
      - Tiny-shell trebuie să suporte redirectarea output-ului unei comenzi (stdout) într-un fişier.
      - Completaţi funcţia `do_redirect`.

- Dacă fişierul indicat de `filename` nu există, va fi creat. Dacă există, trebuie trunchiat.
- Hints:
  - Urmăriţi în cod comentariile cu `TODO-3`
  - Citiţi secţiunea Copierea descriptorilor de fişier
- Pentru testare puteţi folosi comenzile: ./tiny
  > ls -al > out

WINDOWS

1. (**0.5 puncte**) Bomb
   - Deschideţi proiectul (fişierul `.sln`) şi compilaţi primul subproiect: `1-bomb`
   - Urmăriţi sursa `1-bomb.c`. Ce credeţi că face?
   - Înainte de a rula `1-bomb.exe`, porniţi `Task Manager` (CTRL + ALT + DEL)
   - Rulaţi `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 Documents\ and\ Settings/student/Desktop/
   1. (**0.5 puncte**) Executarea unei comenzi simple
      - Partea de executare a unei comenzi simple şi a variabilelor de mediu este deja implementată.
      - Urmăriţi în sursă funcţia `RunSimpleCommand`.
      - Testaţi funcţionalitatea prin comenzi de tipul: ./tiny
        > ls -al > out
        > exit
   2. (**1.5 puncte**) Redirectare
      - Realizaţi redirectarea tuturor HANDLE-relor
        - Hints:
          - Completaţi funcţia `RedirectHandle`.
          - **Atenţie!** Trebuie iniţializate toate handle-rele structurii STARTUPINFO.
          - Hints:
            - Revedeţi secţiunea Moştenirea handle-urilor.
            - Aveţi grijă cum se face moştenirea handle-relor.
          - Pentru testare puteţi folosi comenzile: ./tiny
            > ls -al > out
   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:
          - Completaţi funcţia `PipeCommands`.
          - **Atenţie!** În procesul părinte, trebuie închise capetele pipe-urilor.
          - Pentru redirectari, folosiţi-vă de funcţia `RedirectHandle`.
          - Revedeţi secţiunea despre Pipe-uri anonime în Windows
          - Pentru testare puteţi folosi comenzile: ./tiny
            > cat Makefile | grep tiny

BONUS

1. (**1 so karma**) Pipe-uri cu nume (Linux/Windows)
   - Realizaţi două programe, denumite `server` şi `client`, care interacţionează printr-un **pipe cu nume**.
   - FIFOul se numeşte `myfifo`. Dacă nu există, este creat de server.
   - Serverul trebuie rulat înaintea clientului.
   - Clientul primeşte ca argument, la rulare un mesaj care va fi transmis serverului.
   - Serverul va afişa mesajul primit la ieşirea standard.
   - **Hints**:
     - Linux:
       - Citiţi secţiunea Pipe-uri cu nume în Linux
     - Windows:
       - Citiţi secţiunea Pipe-uri cu nume în Windows
       - Puteţi porni de la exemplul din documentaţia `CreateNamedPipe`.
       - **Atenţie**: Dacă `ReadFile` întoarce FALSE, iar mesajul de eroare (ce întoarce GetLastError()) este

`ERROR_BROKEN_PIPE`, înseamnă că au fost închise toate capetele de scriere.

1. (**1 so karma**) Magic
   - Intraţi în directorul `lin/5-magic` şi deschideţi sursa `magic.c`
   - Completaţi **doar** în condiţia instrucţiunii `if` pentru a obţine la rulare mesajul "Hello World".
     - Nu sunt permise alte modificări în funcţia `main`.

Soluţii

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: **2011/03/04 18:27**