Laborator 03 - Procese

Materiale ajutătoare

* [lab03-slides.pdf](http://elf.cs.pub.ro/so/res/laboratoare/lab03-slides.pdf)
* [lab03-refcard.pdf](http://elf.cs.pub.ro/so/res/laboratoare/lab03-refcard.pdf)
* [Video Procese](http://elf.cs.pub.ro/so/res/tutorial/lab-03-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](http://en.wikipedia.org/wiki/Process_control_block)), câte una pentru fiecare proces existent în sistem. Printre cele mai importante informații conținute de PCB regăsim:

* PID - identificatorul procesului
* spațiu de adresă
* registre generale, PC (contor program), SP (indicator stivă)
* tabela de fișiere deschise
* [informații referitoare la semnale](http://ocw.cs.pub.ro/courses/so/laboratoare-2013/laborator-06)
  + 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.

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](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#variabile_de_mediu_in_linux), [Windows](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#variabile_de_mediu_in_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](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#pipe-uri_in_linux), [Windows](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#pipe-uri_in_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. Prin urmare, 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

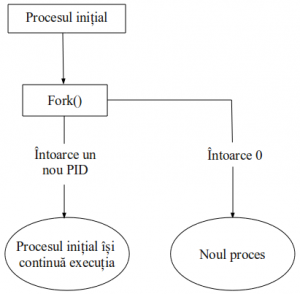
În UNIX un proces se creează folosind apelul de sistem [fork](http://linux.die.net/man/2/fork):

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.

Această funcție este apelată o dată și se întoarce (în caz de succes) de două ori:

* În părinte va întoarce pid-ul procesului nou creat (copil).
* În procesul copil apelul va întoarce 0.

[](http://ocw.cs.pub.ro/courses/_detail/so/laboratoare/fork.png?id=so%3Alaboratoare%3Alaborator-03)

Pentru aflarea PID-ului procesului curent și al procesului părinte se vor apela funcțiile de mai jos.

Funcția [getpid](http://linux.die.net/man/3/getpid) întoarce PID-ul procesului apelant:

pid\_t getpid(void);

Funcția [getppid](http://linux.die.net/man/3/getppid) întoarce PID-ul procesului părinte al procesului apelant:

pid\_t getppid(void);

Înlocuirea imaginii unui proces

Familia de funcții [exec](http://linux.die.net/man/3/exec) va executa un nou program, înlocuind imaginea procesului curent, cu cea dintr-un fișier (executabil). Acest lucru înseamnă:

* Spațiul de adrese al procesului va fi înlocuit cu unul nou, creat special pentru execuția fișierului.
* Registrele PC (contorul program), SP (indicatorul stivă) și registrele generale vor fi reinițializate.
* Măștile de semnale ignorate și blocate sunt setate la valorile implicite, ca și handler-ele semnalelor.
* PID-ul și descriptorii de fișier care nu au setat flag-ul CLOSE\_ON\_EXEC rămân neschimbați (implicit, flag-ulCLOSE\_ON\_EXEC nu este setat).

int execl(const char \*path, const char \*arg, ...);

int execv(const char \*path, char \*const argv[]);

int execlp(const char \*file, const char \*arg, ...);

Exemplu de folosire a funcțiilor de mai sus:

execl("/bin/ls", "ls", "-la", NULL);

char \*const argvec[] = {"ls", "-la", NULL};

execv("/bin/ls", argvec);

execlp("ls", "ls", "-la", NULL);

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

Toate funcțiile exec\* sunt implementate prin apelul de sistem [execve](http://linux.die.net/man/2/execve).

Așteptarea terminării unui proces

Familia de funcții [wait](http://linux.die.net/man/3/waitpid) 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 interogat se poate afla examinând status cu macrodefiniții precum [WEXITSTATUS](http://linux.die.net/man/3/waitpid), care întoarce codul de eroare cu care s-a încheiat procesul așteptat, evaluând cei mai nesemnificativi 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, se poate folosi:

wait(NULL);

Terminarea unui proces

Pentru terminarea procesului curent, Linux pune la dispoziție apelul de sistem exit.

void [exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(int status);

Apeluri exit

Dintr-un program C există trei moduri de invocare a acestui apel de sistem:

1. apelul [\_exit](http://linux.die.net/man/2/exit) ([POSIX.1-2001](http://linux.die.net/man/7/standards)):

void \_exit(int status);

2. apelul [\_Exit](http://linux.die.net/man/2/exit) din biblioteca standard C (conform [C99](http://en.wikipedia.org/wiki/C99)):

void \_Exit(int status);

3. apelul [exit](http://linux.die.net/man/3/exit) din biblioteca standard C (conform C89, C99), cel prezentat mai sus.

\_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 “înfiați” de init
* un semnal SIGCHLD va fi trimis către părintele procesului. 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

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

Un proces al cărui părinte s-a terminat poartă numele de **proces orfan**. Acest proces este adoptat automat de către procesul init, dar poartă denumirea de orfan în continuare deoarece procesul care l-a creat inițial nu mai există.

Un proces finalizat al cărui părinte nu a citit (încă) statusul terminării acestuia poartă numele de **proces zombie**. Procesul intră într-o stare de terminare, iar informația continuă să existe în tabela de procese astfel încât să ofere părintelui posibilitatea de a verifica codul cu care s-a finalizat procesul. În momentul în care părintele apelează funcția wait, informația despre proces dispare. Orice proces copil o să treacă prin starea de proces zombie la terminare.

Pentru terminarea unui alt proces din sistem, se va trimite un semnal către procesul respectiv prin intermediul apelului de sistem [kill](http://linux.die.net/man/2/kill). Mai multe detalii despre kill și semnale în[laboratorul de semnale](http://ocw.cs.pub.ro/courses/so/laboratoare-2013/laborator-06).

Exemplu (my\_system)

[my\_system.c](http://ocw.cs.pub.ro/courses/_export/code/so/laboratoare/laborator-03?codeblock=11)

*#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](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(EXIT\_FAILURE);

default:

*/\* parent process \*/*

**break**;

}

*/\* only parent process gets here \*/*

waitpid(pid, &status, 0);

if (WIFEXITED(status))

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("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](http://linux.die.net/man/2/dup) duplică descriptorul de fișier oldfd și întoarce noul descriptor de fișier, sau -1 în caz de eroare:

int dup(int oldfd);

[dup2](http://linux.die.net/man/2/dup) 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 dintre file descriptori (de ex. lseek) sunt vizibile și pentru celălalt file descriptor (duplicat).

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.

Detalii despre flagul CLOSE\_ON\_EXEC

**fcntl**

Pentru a seta flag-ul CLOSE\_ON\_EXEC se folosește funcția [fcntl](http://linux.die.net/man/3/fcntl), cu un apel de forma:

fcntl(file\_descriptor, F\_SETFD, FD\_CLOEXEC);

**O\_CLOEXEC**

fcntl poate activa flag-ul FD\_CLOEXEC doar pentru descriptori de fișier deja existenți. În aplicații cu mai multe fire de execuție, între crearea unui descriptor de fișier și un apel fcntl se poate interpune un apelexec pe un alt fir de execuție.

/ \* THREAD 1 \*/ |/ \* THREAD 2 \*/

fd = op\_creare\_fd() |

| exec(...)

fcntl(fd, F\_SETFD, FD\_CLOEXEC); |

Cum, implicit, descriptorii de fișiere sunt moșteniți după un apel exec, deși programatorul a dorit ca acesta să nu poată fi accesat după exec, nu poate preveni apariția unui apel exec între creare și fcntl.

Pentru a rezolva această condiție de cursă, s-au introdus în Linux 2.6.27 (2008) versiuni noi ale unor apeluri de sistem:

int dup3(int oldfd, int newfd, int flags);

int pipe2(int pipefd[2], int flags);

int accept4(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen, int flags);

Aceste variante ale apelurilor de sistem adaugă câmpul flags, prin care se poate specifica O\_CLOEXEC, pentru a crea și activa CLOSE\_ON\_EXEC în mod *atomic*. Numărul din numele apelului de sistem, specifică numărul de parametri ai apelului.

Apelurile de sistem care creează descriptori de fișiere care primeau deja un parametru flags (e.g. open) au fost doar extinse să accepte și O\_CLOEXEC.

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](http://linux.die.net/man/3/getenv) întoarce valoarea variabilei de mediu denumite name, sau NULL dacă nu există o variabilă de mediu denumită astfel:

char\* [getenv](http://www.opengroup.org/onlinepubs/009695399/functions/getenv.html)(const char \*name);

[setenv](http://linux.die.net/man/3/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](http://linux.die.net/man/3/unsetenv) șterge din mediu variabila denumită name:

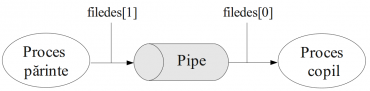
int unsetenv(const char \*name);

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](http://linux.die.net/man/2/pipe):

[](http://ocw.cs.pub.ro/courses/_detail/so/laboratoare/pipe.png?id=so%3Alaboratoare%3Alaborator-03)

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[1)](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#fn__1) 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 semnaluluiSIGPIPE. Dacă semnalul este captat și se revine din rutina de tratare, funcția de sistem write returnează eroare și variabila errno are valoarea EPIPE.

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 unuifork, nu uitați să închideți capetele pipe-ului în procesul părinte).

Alte funcții utile: [popen](http://linux.die.net/man/3/popen), [pclose](http://linux.die.net/man/3/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](http://linux.die.net/man/3/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:

Detalii despre flagul O\_NONBLOCK

* dacă O\_NONBLOCK nu este specificat (cazul normal), atunci un open pentru citire se va bloca până când un alt proces deschide același FIFO pentru scriere. Analog, dacă deschiderea este pentru scriere, se poate produce blocare până când un alt proces efectuează deschiderea pentru citire.
* dacă se specifică O\_NONBLOCK, atunci deschiderea pentru citire revine imediat, dar o deschidere pentru scriere poate returna eroare cu errno având valoarea ENXIO, dacă nu există un alt proces care a deschis același FIFO pentru citire.

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.

Depanarea unui proces

Informații suplimentare legate de depanarea unui proces se gǎsesc [aici](http://ocw.cs.pub.ro/courses/so/laboratoare/resurse/gdb#depanarea_unui_proces)

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](http://msdn.microsoft.com/en-us/library/ms682425.aspx).

|  |  |
| --- | --- |
| 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 dispoziție câteva funcții înrudite precum [CreateProcessAsUser](http://msdn.microsoft.com/en-us/library/ms682429%28VS.85%29.aspx),[CreateProcessWithLogonW](http://msdn.microsoft.com/en-us/library/ms682431%28VS.85%29.aspx) ori [CreateProcessWithTokenW](http://msdn.microsoft.com/en-us/library/ms682434%28VS.85%29.aspx), 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](http://msdn.microsoft.com/en-us/library/ms684320(VS.85).aspx):

HANDLE OpenProcess(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

DWORD dwProcessId

);

iar pentru a obține un handle al procesului curent se va apela [GetCurrentProcess](http://msdn.microsoft.com/en-us/library/ms683179(VS.85).aspx):

HANDLE GetCurrentProcess(void);

Pentru a obține PID-ul procesului curent se va apela [GetCurrentProcessId](http://msdn.microsoft.com/en-us/library/ms683180(VS.85).aspx):

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

Deoarece funcția [CreateProcess](http://msdn.microsoft.com/en-us/library/ms682425.aspx) se întoarce imediat, fără a aștepta ca procesul nou creat să-și termine inițializările, este nevoie de un mecanism prin care procesul părinte să se sincronizeze cu procesul copil înainte de a încerca să comunice cu acesta. Windows pune la dispoziție funcția de așteptare[WaitForInputIdle](http://msdn.microsoft.com/en-us/library/ms687022.aspx).

DWORD WaitForInputIdle(

HANDLE hProcess,

DWORD dwMilliseconds

);

Funcția va cauza blocarea firului de execuție apelant până în momentul în care procesul hProcess și-a terminat inițializarea și așteaptă date de intrare. Funcția poate fi folosită oricând pentru a aștepta ca procesul hProcesssă treacă în starea în care așteaptă date de intrare, nu doar la momentul creării sale. Funcției i se poate specifica o durată de așteptare prin intermediul parametrului dwMilliseconds.

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](http://msdn.microsoft.com/en-us/library/ms687032.aspx) ori [WaitForMultipleObjects](http://msdn.microsoft.com/en-us/library/ms687025.aspx).

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. Mai multe detalii pot fi găsite [aici](http://ocw.cs.pub.ro/courses/so/laboratoare-2013/laborator-05).

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](http://msdn.microsoft.com/en-us/library/ms683189.aspx):

BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);

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

* parametrul transmis uneia din funcțiile [ExitProcess](http://msdn.microsoft.com/en-us/library/ms682658(VS.85).aspx) sau [TerminateProcess](http://msdn.microsoft.com/en-us/library/ms686714(VS.85).aspx) (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 întoarsă 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](http://msdn.microsoft.com/en-us/library/ms682658(VS.85).aspx).

void ExitProcess(UINT uExitCode);

Consecinţele funcţiei ExitProcess sunt:

* 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.

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](http://msdn.microsoft.com/en-us/library/ms686714(VS.85).aspx).

BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);

Funcţia TerminateProcess se ocupă de:

* A iniția terminarea procesului hProcess și a tuturor firelor sale de execuție. Se vor revoca operațiile de intrare/ieșire neterminate după care funcția [TerminateProcess](http://msdn.microsoft.com/en-us/library/ms686714(VS.85).aspx) va întoarce imediat.
* A închide toți descriptorii de resurse (handle) ai procesului.

Funcția [TerminateProcess](http://msdn.microsoft.com/en-us/library/ms686714(VS.85).aspx) 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](http://ocw.cs.pub.ro/courses/_export/code/so/laboratoare/laborator-03?codeblock=34)

*#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](http://msdn.microsoft.com/en-us/library/ms682425%28VS.85%29.aspx), handle-urile din procesul părinte pot fi **moștenite** în procesul copil.

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

* membrul bInheritHandle, al structurii SECURITY\_ATTRIBUTES, transmise lui [CreateFile](http://msdn.microsoft.com/en-us/library/aa363858%28VS.85%29.aspx), trebuie să fie TRUE
* parametrul bInheritHandles, al lui [CreateProcess](http://msdn.microsoft.com/en-us/library/ms682425%28VS.85%29.aspx), trebuie să fie TRUE.
* atunci când se doreşte moştenierea handler-elor în procesul copil, trebuie să ne asigurăm că acestea sunt valide deorece în procesul copil nu se fac validări suplimentare. Transmiterea unor handlere invalide poate duce la un comportament nedefinit în procesul copil.

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

Cei 3 descriptori speciali de fișier pot fi obținuți apelând funcția [GetStdHandle](http://msdn.microsoft.com/en-us/library/ms683231(VS.85).aspx):

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](http://msdn.microsoft.com/en-us/library/ms686331(v=VS.85).aspx), transmise lui [CreateProcess](http://msdn.microsoft.com/en-us/library/ms682425%28VS.85%29.aspx). În acest caz, membrul dwFlags al aceleiași structuri trebuie setat la STARTF\_USESTDHANDLES. Dacă se dorește ca anumite handle-uri să rămână implicite, li se poate atribui handle-ul întors de [GetStdHandle](http://msdn.microsoft.com/en-us/library/ms683231(VS.85).aspx).

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;

Pentru a realiza redirectarea in mod corespunzător, câmpurile hStdInput, hStdOutput, hStdErrordin 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](http://msdn.microsoft.com/en-us/library/ms683179%28VS.85%29.aspx).

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 aceleași 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](http://msdn.microsoft.com/en-us/library/ms683188(VS.85).aspx):

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](http://msdn.microsoft.com/en-us/library/ms686206(VS.85).aspx):

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](http://msdn.microsoft.com/en-us/library/ms686206(VS.85).aspx) 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.

Detalii despre Environment Block

Un utilizator are acces la propriul Environment Block prin apelul funcției [GetEnvironmentStrings](http://msdn.microsoft.com/en-us/library/ms683187(VS.85).aspx):

LPTCH GetEnvironmentStrings(void);

care îi va întoarce un pointer spre acesta, pe care îl poate elibera cu [FreeEnvironmentStrings](http://msdn.microsoft.com/en-us/library/ms683151(VS.85).aspx):

BOOL FreeEnvironmentStrings(

LPTSTR lpszEnvironmentBlock

);

Un proces copil va moșteni Environment Block-ul părintelui dacă acesta apelează CreateProcess, cu parametrullpEnvironment = NULL.

Se poate obține Environment Block-ul unui alt utilizator prin intermediul funcției [CreateEnvironmentBlock](http://msdn.microsoft.com/en-us/library/bb762270(VS.85).aspx):

BOOL CreateEnvironmentBlock(

LPVOID\* lpEnvironment,

HANDLE hToken,

BOOL bInherit

);

Trebuie să pasăm hToken, token-ul asociat utilizatorului al cărui bloc vrem să-l aflăm, pe care putem să-l obținem prin apelarea funcției [LogonUser](http://msdn.microsoft.com/en-us/library/aa378184.aspx):

BOOL LogonUser(

LPTSTR lpszUsername,

LPTSTR lpszDomain,

LPTSTR lpszPassword,

DWORD dwLogonType,

DWORD dwLogonProvider,

PHANDLE phToken

);

Și, bineînțeles, trebuie cunoscută parola utilizatorului respectiv.

Environment Block-ul, obținut prin CreateEnvironmentBlock, poate fi transmis ca parametru funcției[CreateProcessAsUser](http://msdn.microsoft.com/en-us/library/ms682429(VS.85).aspx), și se va distruge prin apelul funcției [DestroyEnvironmentBlock](http://msdn.microsoft.com/en-us/library/bb762274(VS.85).aspx):

BOOL DestroyEnvironmentBlock(

LPVOID lpEnvironment

);

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](http://msdn.microsoft.com/en-us/library/aa365152(VS.85).aspx):

|  |  |
| --- | --- |
| BOOL CreatePipe(  PHANDLE hReadPipe,  PHANDLE hWritePipe,  LPSECURITY\_ATTRIBUTES lpPipeAttributes,  DWORD nSize  ); | CreatePipe(  &hReadPipe,  &hWritePipe,  &sa, *//pentru moștenire sa.bInheritHandle=TRUE*  0 *//dimensiunea default pentru pipe*  ); |

Pentru a **moșteni** un pipe anonim, este nevoie ca parametrul bInheritHandle din structura[LPSECURITY\_ATTRIBUTES](http://msdn.microsoft.com/en-us/library/aa379560%28v=VS.85%29.aspx) 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](http://msdn.microsoft.com/en-us/library/aa914377.aspx) și [WriteFile](http://msdn.microsoft.com/en-us/library/aa910675.aspx).

[ReadFile](http://msdn.microsoft.com/en-us/library/aa914377.aspx) 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
* a apărut o eroare.

[WriteFile](http://msdn.microsoft.com/en-us/library/aa910675.aspx) se termină atunci când toți octeții au fost scriși. Dacă bufferul pipe-ului este plin înainte ca toți octeții să fie scriși, [WriteFile](http://msdn.microsoft.com/en-us/library/aa910675.aspx) rămâne blocat până când alt proces sau thread folosește [ReadFile](http://msdn.microsoft.com/en-us/library/aa914377.aspx) 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](http://msdn.microsoft.com/en-us/library/aa365150%28VS.85%29.aspx).

|  |  |
| --- | --- |
| 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* |

Funcția returnează un handle către capătul serverului la pipe. Acest handle poate fi transmis funcției[ConnectNamedPipe](http://msdn.microsoft.com/en-us/library/aa365146(VS.85).aspx) 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](http://msdn.microsoft.com/en-us/library/aa363858%28VS.85%29.aspx) sau[CallNamedPipe](http://msdn.microsoft.com/en-us/library/aa365144(VS.85).aspx) - ultima funcţie este mai utilă pentru transmiterea de mesaje.

Un exemplu funcțional folosind pipe-uri cu nume se află [aici](http://msdn.microsoft.com/en-us/library/aa365588%28v=VS.85%29.aspx)

Mai multe detalii despre moștenirea pipe-urilor se pot găsi [aici](http://msdn.microsoft.com/en-us/library/windows/desktop/aa365782(v=vs.85).aspx).

Detalii extra despre pipe-urile cu nume

**Moduri de comunicare**

Comunicația prin pipe-urile cu nume poate fi de tip:

* **mesaj**
  + se scriu/citesc date sub formă de mesaje;
  + este necesară cunoașterea lungimii mesajului;
  + se scriu/citesc doar *mesaje complete*;
  + creat cu PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE.
* **flux de octeți**
  + nu există nicio garanție asupra numărului de octeți care sunt citiți/scriși în orice moment;
  + se pot transmite date fără să se țină seama de conținut, pe când, prin pipe-urile de tip mesaj, comunicația are loc în unități discrete (mesaje);
  + creat cu PIPE\_TYPE\_BYTE | PIPE\_READMODE\_BYTE (implicit).

**Numire**

Crearea unui pipe cu nume se poate face numai pe mașina locală (reprezentată prin primul .) cu un string de forma:

\\.\pipe\[path]pipename

Accesarea unui pipe cu nume se poate face folosind ca parametru un string de forma:

\\servername\pipe\[path]pipename

**Funcții utile**

[GetNamedPipeHandleState](http://msdn.microsoft.com/en-us/library/aa365443%28VS.85%29.aspx) - întoarce informații despre anumite atribute cum ar fi: tipul de comunicare (mesaj sau byte-stream), numărul de instanțe, dacă a fost deschisă în mod blocant sau neblocant.

[SetNamedPipeHandleState](http://msdn.microsoft.com/en-us/library/aa365787%28VS.85%29.aspx) - permite modificarea atributelor unui pipe

Exerciții

În rezolvarea laboratorului folosiți arhiva de sarcini [lab03-tasks.zip](http://elf.cs.pub.ro/so/res/laboratoare/lab03-tasks.zip)

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

Exercițiul -1 - GSOC (0p)

Google Summer of Code este un program de vară în care studenții (indiferent de anul de studiu) sunt implicați în proiecte Open Source pentru a își dezvolta skill-urile de programare, fiind răsplătiți cu o bursă în valoare totală de 5500$ [gsoc](https://developers.google.com/open-source/gsoc).

UPB se află în top ca număr de studenți acceptați; în fiecare an fiind undeva la aprox. 30-40 de studenți acceptați. Vă încurajăm să aplicați! Există și un grup de fb cu foști participanți unde puteti să îi contactați pentru sfaturi [facebook page](https://www.facebook.com/groups/240794072931431/)

Exercițiul 0 - Joc interactiv (2p)

* Detalii desfășurare [joc](http://ocw.cs.pub.ro/courses/so/meta/notare#joc_interactiv).

Linux (5p)

Exercițiul 1 - system (1.5p)

Intrați în directorul 1-system. Programul my\_system.c execută o comandă transmisă ca parametru, folosind funcția de bibliotecă [system](http://linux.die.net/man/3/system). Modul de funcționare al [system](http://linux.die.net/man/3/system) este următorul:

* se creează un nou proces cu [fork](http://linux.die.net/man/2/fork)
* procesul copil execută, folosind [execve](http://linux.die.net/man/2/execve), programul **sh** cu argumentele -c “comanda”, timp în care procesul părinte așteaptă terminarea procesului copil.

Compilați (folosind make) și rulați programul dând ca parametru o comandă.

* Exemplu:

./my\_system pwd

Cum procedați pentru a trimite mai mulți parametri unei comenzi? (ex: ls -la)

Pentru a vedea câte apeluri de sistem [execve](http://linux.die.net/man/2/execve) se realizează, rulați:

**strace** -e execve,clone -ff -o output ./my\_system **ls**

* **atenție!** nu este spațiu după virgulă în argumentul execve,clone
* argumentul -ff însoțit de -o output generează câte un fișier de output pentru fiecare proces.
  + cițiți pagina de manual [strace](http://linux.die.net/man/1/strace)

Revedeți secțiunea [Înlocuirea imaginii unui proces](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#inlocuirea_imaginii_unui_proces) și pagina de manual pentru [execve](http://linux.die.net/man/2/execve).

Exercițiul 2 - orphan (1p)

Intrați în directorul 2-orphan și inspectați sursa orphan.c.

Compilați programul (make) și apoi rulați-l folosind comanda:

​ ./​orphan

Deschideți alt terminal și rulați comanda:

**watch** **ps** -al

Observați că pentru procesul indicat de executabilul ​orphan​ (coloana ​CMD​),​ pid-ul procesului părinte (coloana ​PPID​) devine 1, întrucât procesul este adoptat de ​init​ după terminarea procesului său părinte.

Exercițiul 3 - Tiny-Shell (2.5p)

Intrați în directorul 3-tiny.

Următoarele subpuncte 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.

3a. Execuția unei comenzi simple (0.5p)

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.

Citiți exemplul [my\_system](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#exemplu_my_system) și urmăriți în cod comentariile cu TODO-1. Pentru testare puteți folosi comenzile:

./tiny

> pwd

> **ls** -al

> exit

3b. Adăugare suport pentru setarea și expandarea variabilelor de mediu (1p)

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 aceaste funcții.

* Urmăriți în cod comentariile cu TODO-2.
* Citiți secțiunea [Variabile de mediu in Linux](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#variabile_de_mediu_in_linux).
* Pentru testare puteți folosi comenzile:
* ./tiny
* > echo $HOME
* > name=Makefile

> **cat** $name

3c. Redirectarea ieșirii standard (1p)

Completați funcția do\_redirect astfel încât tiny-shell trebuie să suporte redirectarea output-ului unei comenzi (stdout) într-un fișier.

Dacă fișierul indicat de filename nu există, va fi creat. Dacă există, trebuie trunchiat.

Citiți secțiunea [Copierea descriptorilor de fișier](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#copierea_descriptorilor_de_fisier) și urmăriți în cod comentariile cu TODO-3. Pentru testare puteți folosi comenzile:

./tiny

> **ls** -al > out

Windows (4p)

Pentru exerciţiul **Tiny-Shell on Windows** 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 (atenție la folosirea apostrofurilor):

$ cd 'C:\Users\Student\Desktop'

Exercițiul 1 - Bomb (0.5p)

Deschideți proiectul (fișierul .sln) și compilați primul subproiect: 1-bomb.

Inspectați sursa 1-bomb.c. Ce credeți că face? [Fork Bomb](https://en.wikipedia.org/wiki/Fork_bomb)

NU executați 1-bomb.exe

Exercițiul 2 - Tiny-Shell on Windows (3.5p)

Ne propunem să continuăm implementarea de **Tiny-Shell**.

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 (atenție la folosirea apostrofurilor):

$ cd 'C:\Users\Student\Desktop'

2a. Execuția unei comenzi simple (0.5p)

Partea de execuție 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

2b. Redirectare (1.5p)

Realizați redirectarea tuturor HANDLE-relor.

Completați funcția RedirectHandle.

* **Atenție!** Trebuie inițializate toate handle-rele structurii STARTUPINFO.
* Urmăriți în cod comentariile cu TODO 1.
* Revedeți secțiunea [Moștenirea handle-urilor](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#mostenirea_handle-urilor_la_createprocess).
* Atenție la metoda de moștenire a handle-relor

Pentru testare puteți folosi comenzile:

./tiny

> **ls** -al > out

> exit

2c. Implementarea unei comenzi cu pipe-uri (1.5p)

Shell-ul vostru trebuie să ofere suport pentru o comandă de forma ' comanda\_simpla | comanda\_simpla '.

Urmăriți în cod comentariile cu TODO 2.

* Completați funcția PipeCommands.
* zeroizaţi structura SECURITY\_ATTRIBUTES sa, respectiv structurile PROCESS\_INFO pi1, pi2;
* **Atenție!**În procesul părinte, trebuie închise capetele pipe-urilor.
* Pentru redirectari, folosiți-vă de funcția RedirectHandle.
* Revedeți secțiunea despre [Pipe-uri anonime în Windows](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#pipe-uri_anonime_in_windows).

Pentru testare puteți folosi comenzile:

./tiny

> **cat** Makefile | **grep** tiny

> exit

BONUS

1 so karma - Pipe-uri cu nume (Linux/Windows)

Realizați două programe, denumite server și client, care interacționează printr-un **pipe cu nume**.

FIFO−ul 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.

Linux:

* Citiți secțiunea [Pipe-uri cu nume în Linux](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#pipe-uri_cu_nume_in_linux).

Windows:

* Citiți secțiunea [Pipe-uri cu nume în Windows](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#pipe-uri_cu_nume_in_windows).
* Puteți porni de la [exemplul](http://msdn.microsoft.com/en-us/library/aa365588.aspx) din documentația CreateNamedPipe.
* **Atenție**: Dacă ReadFile întoarce FALSE, iar mesajul de eroare (ce întoarce GetLastError()) esteERROR\_BROKEN\_PIPE, înseamnă că au fost închise toate capetele de scriere.

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.