Procese

Contents

Procese

Un concept cheie în orice sistem de operare este procesul. Un proces este un program în execuie. Procesele sunt unitatea primitivă prin care sistemul de operare alocă resurse utilizatorilor. Orice proces are un spaiu de adrese i unul sau mai multe fire de execuie. Putem avea mai multe procese ce execută acelai program, dar oricare două procese sunt complet independente.

Spaiile de adrese, regitrii generali, PC (contor program), SP (indicator stivă), tabelele de fiiere deschise, lista de semnale (blocate, ignorate sau care ateaptă să fie trimise procesului), handler-ele pentru semnale, informaiile referitoare la sistemele de fiiere (directorul rădăcină, directorul curent), toate acestea NU sunt partajate, ci aparin fiecărui proces în parte. Aceste informaii necesare pentru rularea programului sunt inute de sistemul de operare într-o structură numită `Process Control Block`, câte una pentru fiecare proces existent în sistem.

În momentul lansării în execuie 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 dispoziie apeluri de sistem pentru crearea unui proces, terminarea unui proces, ateptarea terminării unui proces precum i apeluri pentru duplicarea descriptorilor de resurse între procese ori închiderea acestor descriptori.

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.

Pe sisteme de 32 de bii fiecare proces are un spaiu de adrese de 4 GiB din care 2 (sau în anumite configuraii 3

[5]
) GiB sunt disponibili pentru alocare procesului, iar ceilali 2 (respectiv 1) GiB fiind rezervat sistemului de operare (codul kernelului i al driverelor, date, cache-uri, etc.). Aadar fiecare proces "vede" sistemul de operare în spaiul său de adrese însă nu poate accesa zona respectivă decât prin intermediul apelurilor de sistem (comutând procesorul în modul de lucru privilegiat). Pe sisteme de 64 de bii spaiul total de adrese este de 16 EiB, iar pe sisteme de 16 bii de doar 64 KiB (procesoarele x86 pe 16 bii puteau adresa 2^{20} = 1 MiB de memorie folosind la adresare doi regitri de 16 bii întrucât, dei era procesor pe 16 bii, avea 20 de linii de adresă).

Procese în Linux

Apelurile de sistem puse la dispoziie de Linux pentru gestionarea proceselor sunt: `fork` i `exec` pentru crearea unui proces i respectiv modificarea imaginii unui proces, `wait` i `waitpid` pentru ateptarea terminării unui proces i `exit` pentru terminarea unui proces. Pentru copierea descriptorilor de fiier Linux pune la dispoziie apelurile de sistem `dup` i `dup2`. Pentru citirea, modificarea ori tergerea unei variabile de mediu, biblioteca standard C pune la dispoziie apelurile `getenv`, `setenv`, `unsetenv` precum i un pointer la tabela de variabile de mediu `environ`.

Rularea unui program executabil

Modul cel mai simplu prin care se poate crea un nou proces este prin folosirea funciei de bibliotecă **`system`**:

```
int system(const char *command);
```

Apelul acestei funcii are ca efect execuia ca o comandă shell a comenzii reprezentate prin irul de caractere `command`. Să luăm ca exemplu următorul program C:

Exemplu 1. sys1.c

```
#include <stdlib.h>
int main(int argc, char **argv)
{
        system("ls -la $HOME");
}
```

care este echivalent cu

[1]
```
$ sh -c "ls -la $HOME"
```

Încă un exemplu:

Exemplu 2. sys2.c

```
#include <stdlib.h>
int main(int argc, char **argv)
{
        system("cd /etc/rc.d/rc$RUNLEVEL.d/; ls -la");
}
```

program C care este echivalent cu

[2]
```
$ sh -c "cd /etc/rc.d/rc$RUNLEVEL.d/; ls -la"
```

Implementarea **`system`**: se creează un nou proces cu `fork`; procesul copil execută prin intermediul `exec` programul `sh` cu argumentele `-c "comanda"`, timp în care părintele ateaptă terminarea procesului copil.

Crearea unui proces

În UNIX singura modalitate de creare a unui proces este prin 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. Copilul primete un nou `PID` de la sistemul de operare. Secvena clasică de creare a unui proces este prezentată în continuare:

Exemplu 3. ex_fork.c

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

...

switch (pid = fork()) {
```

```
        case -1: /* fork failed */
                printf("fork failed\n");
                exit(-1);
        case 0: /* child starts executing here */
                ...
        default: /* parent starts executing here */
                printf("created process with pid %d\n", pid);
                ...
}
```

După cum se observă din comentariile de mai sus, apelul de sistem `fork` întoarce `PID`-ul noului proces în procesul părinte i valoarea 0 în procesul copil. Pentru aflarea `PID`-ului procesului curent ori al procesului părinte se va apela una din funciile de mai jos.

Funcia **`getpid`** întoarce `PID`-ul procesului apelant:

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

Funcia **`getppid`** întoarce `PID`-ul procesului părinte al procesului apelant:

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

Înlocuirea imaginii unui proces

Familia de funcii **`exec`** va executa un nou program, înlocuind imaginea procesului curent, cu cea dintr-un fiier (executabil). Spaiul de adrese al procesului va fi înlocuit cu unul nou, creat special pentru execuia fiierului. De asemenea vor fi reiniializai regitrii IP (EIP/RIP - contorul program) i SP (ESP/RSP - indicatorul stivă) i regitrii generali. Mătile de semnale ignorate i blocate sunt setate la valorile implicite, ca i handler-ele semnalelor. `PID`-ul i descriptorii de fiiere care nu au setat flagul `CLOSE_ON_EXEC` rămân neschimbai (implicit flagul `CLOSE_ON_EXEC` nu este setat).

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

Presupunem ca vrem sa apelam comanda `ls -la`:

```
execl("ls", "ls", "-la", NULL);
```

Se observa ca primul argument este insusi numele programului, iar ultimul este `NULL`.

`execl` nu caută programul dat ca parametru în `PATH`, astfel că acesta trebuie însoit de calea completă. Versiunea **`execlp`** caută programul i în `PATH`.

Există, de asemenea, versiunile **`execv`**, care primesc argumentele programului de rulat ca vector, in loc de argumente variabile ale functiei.

```
char * const argvec[] = {"ls", "-la", NULL};
execv("/bin/ls", argvec);
```

Folosirea oricărei funcii din familia `exec` necesită includerea header-ului `unistd.h`.

Ateptarea terminării unui proces

Familia de funcii **wait** suspendă execuia 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);
```

Valoarile uzuale ale argumentului pid sunt identificatorul unui proces copil (spre exemplu, returnat de fork) sau -1, în cazul în care se dorete ateptarea oricărui proces copil.

Parametrul options permite specificarea diferitelor opiuni.

Funcia va întoarce PID-ul procesului a cărui stare e raportată; infomaiile de stare sunt depuse ca int la adresa indicată prin argumentul status.

Starea procesului interogat se poate afla examinând status cu macrodefiniii precum **WEXITSTATUS**, care întoarce codul de eroare cu care s-a încheiat procesul copil, evaluând cei mai nesemnificativi 8 bii. Prin convenie, un cod de eroare egal cu 0 semnifică succes.

Există o variantă simplificată care ateaptă orice proces copil să se termine:

```
pid_t wait(int *status);
```

Este echivalentă cu:

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

Pentru a folosi wait sau waitpid trebuie incluse header-ele sys/types.h i sys/wait.h.

Terminarea unui proces

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

- apelul **_exit** (POSIX.1-2001):

```
#include <unistd.h>
void _exit (int status);
```

- apelul **_Exit** din biblioteca standard C (conform C99):

```
#include <stdlib.h>
void _Exit (int status);
```

- apelul **exit** din biblioteca standard C (conform C89, C99):

```
#include <stdlib.h>
void exit (int status);
```

_exit(2) i _Exit(2) sunt funcional echivalente (doar că sunt definite de standarde diferite):

- procesul apelant se va termina imediat.
- toi descriptorii de fiier ai procesului sunt închii, copiii procesului sunt "înfiai" de `init`, iar părintelui procesului îi va fi trimis un semnal `SIGCHLD`. Procesului părinte îi va fi întoarsă valoarea `status` ca rezultat al unei funcii de ateptare (`wait` sau `waitpid`).

În plus `exit`(3):

- va terge toate fiierele create cu `tmpfile()`
- va scrie bufferele streamurilor deschise i le va închide.

Notă: Conform ISO C, un program care se termină cu un `return x;` din `main()` va avea acelai comportament ca i 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` i semnale în laboratorul de semnale.

my_system

Un exemplu de folosire a primitivelor `exec`, `fork`, `exit` i `wait` (sau de rularea a unui program) îl reprezintă chiar reimplementarea apelului de bibliotecă `system`

Exemplu 4. my_system.c

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

int my_system(const char *command)
{
    int pid, status;

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

        case 0:
        {
            const char *argv[] = {"/bin/bash", "-c", command, NULL};
            execv("/bin/bash", (char *const *)argv);

            /* exec se poate întoarce doar cu cod de eroare (de ex. când
               nu se găsete fiierul de executat - în cazul nostru /bin/bash.
               În caz de eroare, terminăm procesul copil */
            exit(-1);
        }
    }

    //doar procesul părinte ajunge aici, i doar dacă fork() s-a terminat cu succes
    waitpid(pid, &status, 0);

    // obtinem codul de eroare cu care s-a terminat copilul
    if (WIFEXITED(status))
        printf("Child %d terminated normally, with code %d\n", pid, WEXITSTATUS(status));
```

```
    return status;
}

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

Copierea descriptorilor de fiier

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

```
#include <unistd.h>
int dup(int oldfd);
```

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

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Descriptorii de fiier sunt, de fapt, indeci în tabela de fiiere deschise. Tabela este populată cu pointeri către o structură cu informaiile despre fiiere. Duplicarea unui descriptor de fiier înseamnă duplicarea intrării din tabela de fiiere deschise (adică 2 pointeri de la poziii diferite din tabelă vor indica spre aceeai structură din sistem asociată fiierului). Din acest motiv, toate informaiile asociate unui fiier (lock-uri, cursor, flaguri) sunt partajate de cei doi file descriptori. Ceea ce înseamnă că operaiile ce modifică aceste informaii pe unul din file descriptori (de ex. `lseek`) sunt vizibile i pentru celălat file descriptor (duplicat). **Atentie!** Există i o excepie: flagul `CLOSE_ON_EXEC` nu este partajat (acest flag nu este inut în structura menionată mai sus).

Motenirea descriptorilor de fiier după operaii `fork`/`exec`

Descriptorii de fiier ai procesului părinte se motenesc în procesul copil în urma apelului `fork`. După un apel `exec` descriptorii de fiier sunt păstrai de asemenea, mai puin aceia dintre ei care au setat flagul `CLOSE_ON_EXEC`.

Pentru a seta flagul `CLOSE_ON_EXEC` se folosete funcia **fcntl** cu un apel de genul:

```
fcntl(file_descriptor, F_SETFD, FD_CLOEXEC);
```

Pentru a putea folosi funcia `fcntl` trebuie incluse header-ele `unistd.h` i `fcntl.h`.

Trei dintre descriptorii de fiier sunt mai importani:

`STDIN_FILENO`

: toate programele obinuite conin acest file descriptor care are valoarea 0; el reprezintă intrarea standard; tot ce utilizatorul scrie la terminal, programul va putea citi din acest file descriptor

`STDOUT_FILENO`

toate programele obinuite au acest file descriptor care are valoarea 1; el reprezintă ieirea standard; toate apelurile de genul `printf` vor genera output la terminal

`STDERR_FILENO`
: toate programele obinuite au acest file descriptor care are valoarea 2; el reprezintă ieirea standard de eroare

Variabile de mediu

```
extern char **environ;
```

Un vector de pointeri la iruri de caractere, ce conin variabilele de mediu i valorile lor. Vectorul e terminat cu `NULL`. irurile de caractere sunt de forma `"VARIABILA=VALOARE"`.

`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) i îi setează valoarea la `value`. Dacă variabila există i `replace` e 0, aciunea 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` terge variabila denumită `name` din mediu:

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

Depanarea unui proces

Pe majoritatea sistemelor de operare pe care a fost portat, `gdb` nu poate detecta când un proces realizează o operaie `fork()`. Atunci când programul este pornit, depanarea are loc exclusiv în procesul iniial, procesele copii nefiind ataate debugger-ului. În acest caz, singura soluie este introducerea unor întârzieri în executia procesului nou creat (de exemplu, prin apelul de sistem `sleep()`), care să ofere programatorului suficient timp pentru a ataa manual `gdb`-ul la respectivul proces, presupunând că i-a aflat PID-ul în prealabil.

Pentru a ataa debugger-ul la un proces deja existent, se folosete comanda `attach`, în felul următor:

```
(gdb) attach PID
```

Această metodă este destul de incomodă i poate cauza chiar o funcionare anormală a aplicaiei depanate, în cazul în care necesităile de sincronizare între procese sunt stricte (de exemplu operaii cu time-out).

Din fericire, pe un număr limitat de sisteme, printre care i Linux, `gdb` permite depanarea comodă a programelor care creează mai multe procese prin `fork()` i `vfork()`. Pentru ca `gdb` să urmărească activitatea proceselor create ulterior, se poate folosi comanda `set follow-fork-mode`, în felul următor:

```
(gdb) set follow-fork-mode mode
```

unde *mode* poate lua valoarea `parent`, caz în care debugger-ul continuă depanarea procesului părinte, sau valoarea `child`, i atunci noul proces creat va fi depanat în continuare. Se poate observa că în această manieră debugger-ul este ataat la un moment dat doar la un singur proces, neputând urmări mai multe simultan.

Cu toate acestea, `gdb` poate *ine evidena* tuturor proceselor create de către programul depanat, dei în continuare numai un singur proces poate fi rulat prin debugger la un moment dat. Comanda `set detach-on-fork` realizează acest lucru:

```
(gdb) set detach-on-fork mode
```

unde *mode* poate fi `on`, atunci când `gdb` se va ataa unui singur proces la un moment dat (comportament implicit), sau `off`, caz în care `gdb` se ataează la toate procesele create în timpul execuiei, i le suspendă pe acelea care nu sunt urmărite, în funcie de valoarea setării `follow-fork-mode`.

Comanda `info forks` afiează informaii legate de toate procesele aflate sub controlul `gdb` la un moment dat:

```
(gdb) info forks
```

De asemenea, comanda `fork` poate fi utilizată pentru a seta unul din procesele din listă drept cel activ (care este urmărit de debugger).

```
(gdb) fork fork-id
```

unde *fork-id* este identificatorul asociat procesului, aa cum apare în lista afiată de comanda `info forks`.

Atunci când un anumit proces nu mai trebuie urmărit, el poate fi înlaturat din listă folosind comenzile `detach fork` i `delete fork`:

```
(gdb) detach fork fork-id
(gdb) delete fork fork-id
```

Diferena dintre cele două comenzi este că `detach fork` lasă procesul să ruleze independent, în continuare, în timp ce `delete fork` îl încheie.

Pentru a ilustra aceste comenzi într-un exemplu concret, să considerăm programul următor:

Exemplu 5. forktest.c

```
// forktest.c
 1      #include <stdio.h>
 2      #include <sys/types.h>
 3      #include <sys/wait.h>
 4      #include <unistd.h>
 5
 6
 7      int main(int argc, char **argv) {
 8              pid_t childPID = fork();
 9
10              if (childPID < 0) {
11                      // An error occured
12                      fprintf(stderr, "Could not fork!\n");
13                      return -1;
```

```
14              } else if (childPID == 0) {
15
16                      // We are in the child process
17                      printf("The child process is executing...\n");
18                      sleep(2);
19
20              } else {
21
22                      // We are in the parent process
23                      if (wait(NULL) < 0) {
24                              fprintf(stderr, "Could not wait for child!\n");
25                              return -1;
26                      }
27                      printf("Everything is done!\n");
28
29              }
30
31              return 0;
32      }
-
```

Dacă vom rula programul cu parametrii implicii de depanare, vom constata că `gdb` va urmări exclusiv execuia procesului părinte:

```
$ gcc -O0 -g3 -o forktest forktest.c
$ gdb ./forktest
[...]
(gdb) run
Starting program: /home/stefan/forktest
The child process is executing...
Everything is done!

Program exited normally.
```

Punem câte un breakpoint în codul asociat procesului părinte, respectiv procesului copil, pentru a evidenia mai bine acest comportament:

```
(gdb) break 17
Breakpoint 1 at 0x8048497: file forktest.c, line 17.
(gdb) break 27
Breakpoint 2 at 0x80484f0: file forktest.c, line 27.

(gdb) run
Starting program: /home/stefan/forktest
The child process is executing...

Breakpoint 2, main () at forktest.c:27
27                      printf("Everything is done!\n");
(gdb) continue
Continuing.
Everything is done!

Program exited normally.
```

Setăm debugger-ul să urmărească procesele copii, i observăm că de data aceasta celălalt breakpoint este atins:

```
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /home/stefan/forktest
[Switching to process 6217]
```

```
Breakpoint 1, main () at forktest.c:17
17                      printf("The child process is executing...\n");
(gdb) continue
Continuing.
The child process is executing...

Program exited normally.
Everything is done!
```

Observai că ultimele două mesaje au fost inversate, faă de cazul precedent: debugger-ul încheie procesul copil, apoi procesul părinte afiează mesajul de final (`Everything is done!`).

Procese în Windows

Apelurile `Win32 API` pe care Windows le pune la dispoziie pentru gestionarea proceselor sunt: `CreateProcess` i variaii ale acesteia pentru crearea unui proces, `WaitForSingleObject` i alte funcii de ateptare pentru ateptarea terminării unui proces, `ExitProcess` pentru terminarea procesului curent i `TerminateProcess` pentru terminarea unui alt proces din sistem. Pentru duplicarea descriptorilor de resurse între procese se va apela funcia `DuplicateHandle`. Pentru citirea ori modificarea unei variabile de mediu, avem la dispoziie apelurile `GetEnvironmentVariable` i `SetEnvironmentVariable` precum i `GetEnvironmentStrings` care întoarce un pointer la tabela de variabile de mediu.

Crearea unui proces

În Windows atât crearea unui nou proces cât i înlocuirea imaginii lui cu cea dintr-un program executabil se realizează prin apelul funciei **`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
);
```

Exemplul de mai jos lansează în execuie `notepad.exe`:

```
STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);         // size of structure in bytes
ZeroMemory( &pi, sizeof(pi) );

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 dispoziie câteva funcii înrudite precum `CreateProcessAsUser`, `CreateProcessWithLogonW` ori `CreateProcessWithTokenW` care permit crearea unui proces într-un context de securitate diferit de cel al utilizatorului curent. Mai multe informaii despre funcia **`CreateProcess`** găsii în documentaia de Platform SDK.

Pentru a se obine un descriptor (handle) al unui proces cunoscându-se `PID`-ul procesului respectiv, se va apela funcia **`OpenProcess`**:

```
HANDLE OpenProcess(
        DWORD dwDesiredAccess,
        BOOL bInheritHandle,
        DWORD dwProcessId
);
```

iar pentru a obine un descriptor al procesului curent se va apela **`GetCurrentProcess`**:

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

Pentru a obine `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 deine descriptori (handle) ai proceselor create de el, însă descriptorii pot fi duplicai între procese ceea ce duce la situaia în care un proces deine descriptori ai unor procese care nu sunt create de el, deci ierarhia implicită dispare.

Ateptarea iniializării procesului creat

Deoarece funcia `CreateProcess` întoarce imediat, fără a atepta ca procesul nou creat să-i termine iniializările, 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 dispoziie funcia de ateptare **`WaitForInputIdle`**.

```
DWORD WaitForInputIdle(
        HANDLE hProcess,
        DWORD dwMilliseconds
);
```

Funcia va cauza blocarea firului de execuie apelant până în momentul în care procesul `hProcess` i-a terminat iniializarea i ateaptă date de intrare. Funcia poate fi folosită oricând pentru a atepta ca procesul `hProcess` să treacă în starea în care ateaptă date de intrare, nu doar la momentul creării sale. Funciei i se poate specifica o durată de ateptare prin intermediul parametrului `dwMilliseconds`.

Ateptarea terminării unui process

Pentru a suspenda execuia procesului curent până când unul sau mai multe alte procese se termină, se va folosi una din funciile de ateptare **`WaitForSingleObject`** ori **`WaitForMultipleObjects`**.

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

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

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

Funciilor li se pot specifica intervale de timeout prin parametrul `dwMilliseconds`. `WaitForMultipleObjects` permite ateptarea terminării mai multor procese.

Funciile de ateptare sunt folosite în cadrul mai general al mecanismelor de sincronizare între procese i vor fi prezentate în detaliu în laboratorul de sincronizare între procese.

Aflarea codului de terminare al procesului ateptat

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

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

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

- parametrul pasat uneia din funciile `ExitProcess` sau `TerminateProcess` (`exit` din libc)
- valoarea returnată de funcia `main` sau `WinMain` a procesului
- codul de eroare al unei excepii netratate care a cauzat terminarea procesului.

Terminarea unui proces

Pentru terminarea procesului curent, Windows API pune la dispoziie funcia **`ExitProcess`**.

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

Procesul apelant i toate firele sale de execuie se vor termina imediat. Toate DLL-urile de care era ataat procesul sunt notificate i se apelează metode de distrugere a resurselor alocate de acestea în spaiul de adresă al procesului. Toi descriptorii de resurse (handle) ai procesului sunt închii. Codul de terminare al procesului este setat la `uExitCode`.

Pentru terminarea unui alt proces din sistem se va apela funcia **`TerminateProcess`**.

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

Se va iniia terminarea procesului `hProcess` i a tuturor firelor sale de execuie i se vor revoca operaiile de intrare/ieire neterminate după care funcia `TerminateProcess` va întoarce imediat. Toi descriptorii de resurse (handle) ai procesului sunt închii. Funcia `TerminateProcess` este periculoasă i se recomandă folosirea ei doar în cazuri extreme, deoarece ea nu notifică DLL-urile de care este ataat procesul `hProcess` de detaarea acestuia, lăsând astfel alocate eventualele date rezervate de DLL în spaiul de adresă al procesului.

Aadar, metoda recomandată de terminare a unui proces este `ExitProcess`. ea asigurând terminarea graioasă a procesului cu eliberarea tuturor resurselor alocate de proces direct sau prin intermediul unor DLL-uri.

Terminarea unui proces NU implică terminarea proceselor create de acesta.

Exec

Exemplificăm în continuare noiunile prezentate până acum.

Exemplu 6. Exec.cpp

```
#include <windows.h>
#include <cstdio>
#include <cstdlib>
#include <cstring>
using namespace std;

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

int Exec(const char *cmdLine)
{
    STARTUPINFO si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    PROCESS_INFORMATION pi;
    char *cmd = _strdup(cmdLine);// lpCommandLine not const
    BOOL bRes = CreateProcess(NULL, cmd, NULL, NULL, FALSE, NORMAL_PRIORITY_CLASS, NULL,
        NULL, &si, &pi);
    if (!bRes) {
        free(cmd);// proper
        return -1;
    }
    DWORD dwRes = WaitForSingleObject(pi.hProcess, INFINITE);// signaled = finished
    if (WAIT_FAILED == dwRes) {
        free(cmd);// proper
        CloseProcess(&pi);
        return -1;
    }
```

```
    bRes = GetExitCodeProcess(pi.hProcess, &dwRes);
    free(cmd);
    CloseProcess(&pi);
    if (!bRes) return -1;
    return static_cast<int>(dwRes);
}

int main()
{
    Exec("cmd /c start /max mspaint");
    printf("done\n");
    return 0;
}
```

Duplicarea descriptorilor de resurse

Pentru duplicarea unui descriptor de resursă (handle) se va apela funcia **DuplicateHandle**. Descriptorul iniial i cel duplicat vor referi acelai obiect i rezultatele operaiilor efectuate asupra oricăruia dintre cei doi descriptori vor fi vizibile i din perspectiva celuilalt. Duplicarea unui descriptor de resursă poate fi de dorit în cazul în care vrem să obinem un descriptor de resursă cu drepturi de acces diferite faă de cel iniial, sau dacă dorim obinerea unui descriptor care nu va fi motenit în procesele copil dintr-unul care va fi motenit.

```
BOOL DuplicateHandle(
        HANDLE hSourceProcessHandle,
        HANDLE hSourceHandle,
        HANDLE hTargetProcessHandle,
        LPHANDLE lpTargetHandle,
        DWORD dwDesiredAccess,
        BOOL bInheritHandle,
        DWORD dwOptions
);
```

Funcia va duplica descriptorul `hSourceHandle` din procesul `hSourceProcessHandle` prin crearea descriptorului `lpTargetHandle` în procesul `hTargetProcessHandle`. Noului descriptor îi vor fi ataate, pe cât posibil, drepturile de acces specificate prin `dwDesiredAccess`. În unele cazuri descriptorul duplicat poate să aibă drepturi mai puin restrictive decât originalul. Prin `bInheritHandle` se specifică dacă descriptorul duplicat va fi motenit de procesele ce urmează să fie create de procesul `hTargetProcessHandle`. Prin `dwOptions` se pot specifica opiuni precum închiderea descriptorului original în momentul duplicării sau preluarea exactă a drepturilor de acces i ignorarea parametrului `dwDesiredAccess`.

Funcia `DuplicateHandle` poate fi invocată fie de procesul sursă, fie de procesul destinaie (inclusiv atunci când procesul sursă este acelai cu procesul destinaie). Să analizăm situaia în care procesele sursă i destinaie diferă. Dacă procesul destinaie apelează `DuplicateHandle`, el nu are de unde să tie `hSourceHandle`, care va trebui să-i fie comunicat de procesul sursă printr-un mecanism de comunicare între procese (parametru în linia de comandă, variabilă de mediu, fiier extern, socket; pipe, memorie partajată, coadă de mesaje). Dacă procesul sursă apelează `DuplicateHandle`, acesta va trebui să-i transmită procesului destinaie descriptorul obinut, printr-un mecanism de comunicare între procese.

Pentru închiderea unui descriptor se va apela funcia **CloseHandle**:

```
BOOL CloseHandle(
        HANDLE hObject
);
```

Apelul funciei invalidează descriptorul `hObject` i decrementează contorul de descriptori al obiectului asociat. Abia când ultimul descriptor spre obiect este închis (contorul de descriptori ajunge 0), obiectul este distrus i eliminat din sistem. Chiar dacă un proces se termină, obiectul asociat lui nu este eliminat din sistem până când nu este închis descriptorul procesului (procesul părinte apelează `CloseHandle(ProcesCopil)`). Dacă descriptorul unui proces este închis înainte de terminarea procesului, obiectul asociat procesului nu este distrus până când procesul respectiv nu se termină (altfel-spus, nu este necesară ateptarea copilului în părinte pentru a apela `CloseHandle`).

Motenirea descriptorilor de resurse la `CreateProcess`

După un apel `CreateProcess` descriptorii de resurse din procesul părinte pot fi motenii în procesul copil. Pentru ca un descriptor de fiier sa poată fi motenit in procesul creat, trebuie îndeplinite 2 condiii:

- membrul `bInheritHandle` al structurii `SECURITY_ATTRIBUTES` pasate lui `CreateFile` trebuie sa fie `TRUE`
- parametrul `bInheritHandles` al lui `CreateProcess` trebuie sa fie `TRUE`.

Descriptorii motenii sunt valizi doar în contextul procesului copil.

Cei 3 descriptori speciali de fiier pot fi obinui apelând funcia **`GetStdHandle`**:

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

cu unul din parametrii:

- `STD_INPUT_HANDLE`
- `STD_OUTPUT_HANDLE`
- `STD_ERROR_HANDLE`

Pentru redirectarea descriptorilor standard în procesul copil puteti folosi membrii `hStdInput`, `hStdOutput`, `hStdError` ai structurii `STARTUPINFO` pasate lui `CreateProcess`. În acest caz, membrul `dwFlags` al aceleiaşi structuri trebuie setat la `STARTF_USESTDHANDLES`. Dacă se doreşte ca anumiți descriptori să rămână cei impliciți, li se poate atribui handle-ul întors de `GetStdHandle`.

Alte proprietăi ale procesului părinte care pot fi motenite sunt variabilele de mediu i directorul curent. Nu vor fi motenii descriptori ai unor zone de memorie alocate de procesul părinte i nici pseudo-descriptori precum cei întori de funcia `GetCurrentProcess`.

Descriptorul din procesul părinte i cel motenit în procesul copil vor referi acelai obiect exact ca în cazul duplicării. De asemenea, descriptorul motenit în procesul copil are aceeai valoare i aceleai drepturi de acces ca i descriptorul din procesul părinte. Pentru a folosi descriptorul motenit, procesul copil va trebui să-i cunoască valoarea i ce obiect referă. Aceste informaii trebuie sa fie pasate de părinte printr-un mecanism extern (IPC, etc).

Variabile de mediu

În Windows există un set de variabile de mediu globale, valabile pentru toi utilizatorii; în plus, fiecare utilizator în parte are asociat un set propriu de variabile de mediu. Împreună, cele două seturi formează `EnvironmentBlock`-ul utilizatorului respectiv. Acest `EnvironmentBlock` este similar cu variabila

`environ` din Linux.

Un utilizator are acces la propriul `EnvironmentBlock` prin apelul funciei **`GetEnvironmentStrings`**:

```
LPTCH GetEnvironmentStrings(void);
```

care îi va întoarce un pointer spre acesta, pe care îl poate elibera cu **`FreeEnvironmentStrings`**:

```
BOOL FreeEnvironmentStrings(
        LPTSTR lpszEnvironmentBlock
);
```

Pentru a afla valoarea unei variabile de mediu se va apela funcia **`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`. Funcia se va folosi i pentru tergerea unei variabile de mediu prin pasarea unui parametru `lpValue=NULL`. `SetEnvironmentVariable` are efect doar asupra variabilelor de mediu ale utilizatorului i nu poate modifica variabile de mediu globale.

Un proces copil va moteni `EnvironmentBlock`-ul părintelui dacă acesta apelează `CreateProcess` cu parametrul `lpEnvironment=NULL`.

Se poate obine `EnvironmentBlock`-ul unui alt utilizator prin intermediul funciei **`CreateEnvironmentBlock`**:

```
BOOL CreateEnvironmentBlock(
        LPVOID* lpEnvironment,
        HANDLE hToken,
        BOOL bInherit
);
```

Trebuie să pasăm `hToken`, tokenul asociat utilizatorului al cărui bloc vrem să-l aflăm, pe care putem să-l obinem prin apelarea funciei **`LogonUser`**:

```
BOOL LogonUser(LPTSTR lpszUsername,
        LPTSTR lpszDomain,
        LPTSTR lpszPassword,
        DWORD dwLogonType,
        DWORD dwLogonProvider,
        PHANDLE phToken
);
```

i, bineînțeles, trebuie cunoscută parola utilizatorului respectiv.

`EnvironmentBlock`-ul obinut prin `CreateEnvironmentBlock` poate fi pasat ca parametru funciei `CreateProcessAsUser` de exemplu, i se va distruge prin apelul funciei **`DestroyEnvironmentBlock`**:

```
BOOL DestroyEnvironmentBlock(
        LPVOID lpEnvironment
);
```

Pipe-uri

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 şi sunt citite de la celălalt capăt). Sistemul de operare garantează sincronizarea între operațiile de citire şi scriere la cele două capete.

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 este accesibil doar prin moştenire. Aceste pipe-uri nu mai există după ce procesele şi-au terminat execuția.
- **pipe-uri cu nume** - există ca fişiere cu drepturi de acces. Aceasta înseamnă că ele vor exista în continuare independent de procesul care le creează şi pot fi folosite de procese neînrudite.

Pipe-uri în Linux

Apelul pipe

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 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 nu pot fi create de utilizatori diferiți ai sistemului.

Apelul de sistem pentru creare este **`pipe`**:

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

Parametrul `filedes` returnează 2 descriptori de fişier: `filedes[0]` deschis pentru citire şi `filedes[1]` deschis pentru scriere. Se consideră că ieşirea lui `filedes[1]` este intrare pentru `filedes[0]`.

Apelul returnează 0 în caz de succes şi -1 în caz de eroare.

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:

1. 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.
2. 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, funcția de sistem `write` returnează eroare şi variabila `errno` are valoarea `EPIPE`.

Cea mai frecventă greşeală relativ la lucrul cu pipe-urile constă în faptul 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`**.

FIFO (Pipe-uri cu nume)

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

Apelul de sistem pentru crearea FIFO este:

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

Parametrii sunt:

- `pathname` reprezintă numele de cale al fişierului FIFO.
- `mode` reprezintă un întreg ce indică drepturile de acces ale fişierului FIFO.

Apelul returnează 0 în caz de succes si -1 în caz de eroare.

Observații: după ce FIFO a fost creat, acestuia i se pot aplica toate funcțiile pentru operații cu fişiere: `open`, `close`, `read`, `write` la fel ca şi altor fişiere.

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

1. Dacă `O_NONBLOCK` nu este specificat (cazul normal), atunci un `open` pentru citire se va bloca până când un alt proces deschide acelaşi FIFO pentru scriere. Analog, dacă deschiderea este pentru scriere, se poate produce blocare până când un alt proces efectuează deschiderea pentru citire.
2. 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 acelaşi FIFO pentru citire.

Atunci când ultimul proces care scrie într-un FIFO îl închide, se va genera un "sfârşit de fişier" pentru procesul care citeşte din FIFO.

Pipe-uri in Windows

Pipe-uri anonime

Sunt pipe-uri unidirecționale create cu ajutorul funcției **`CreatePipe`**. Funcția returnează două handlere: unul pentru citirea din pipe şi unul pentru scrierea în pipe.

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

`hReadPipe` `hWritePipe` sunt inițializați cu capetele de citire, respectiv de scriere ale pipe-ului. În caz de succes se returnează o valoare diferită de zero şi zero în caz de eroare. Pentru informații detaliate asupra erorii se foloseşte `GetLastError`. Iată un exemplu de apel:

```
HANDLE hReadPipe, hWritePipe;
SECURITY_ATTRIBUTES sa;       // set bInheritHandle to TRUE for allowing child to inherit handles

CreatePipe(
        &hReadPipe,
        &hWritePipe,
        &sa,
        0             // implicit buffer size
);
```

Observații: `CreatePipe` creează pipe-ul, stabilind pentru bufferul de stocare dimensiunea specificată. `CreatePipe` creează de asemenea handlere pe care procesul le foloseşte pentru a citi/scrie din/în pipe cu ajutorul funcțiilor `ReadFile` şi `WriteFile`.

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

Când un proces foloseşte `WriteFile` pentru a scrie într-un pipe anonim, operația de scriere se termină atunci când toți octeții au fost scrişi. Dacă bufferul pipe-ului este plin înainte ca toți octeții să fie scrişi, `WriteFile` rămâne blocat pâna ce alt proces sau thread foloseşte `ReadFile` pentru a face loc.

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

Pipe-uri cu nume

Un pipe cu nume este un pipe unidirecțional sau bidirecțional ce realizează comunicația între un server pipe şi unul sau mai mulți clienți 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 mulți clienți prin acelaşi pipe se folosesc *instanțe* ale pipe-ului. O instanță a unui pipe foloseşte acelaşi nume, dar are propriile handlere şi buffere.

Cel mai simplu server pipe ar fi acela care creează o singură instanță a unui pipe şi comunică cu un singur client.

Moduri de comunicare

Comunicația prin pipe-urile cu nume poate fi de tip *flux de octeți* sau de tip *mesaj*.

Un pipe de tip mesaj va trimite şi va citi date sub formă de mesaje. Deci este nevoie să se ştie lungimea mesajului şi se vor citi respectiv scrie doar mesaje complete.

Pentru un pipe de tip flux de octeți nu există nicio garanție asupra numărului de bytes care sunt citiți / scrişi în orice moment. Astfel 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).

Pipe-urile cu nume în Windows au un nume unic care le distinge în lista de obiecte cu nume din sistem. Un server specifică un nume pentru pipe la crearea lui prin apelarea funcției `CreateNamedPipe`. Clienții specifică numele pipe-ului când apelează `CreateFile` pentru conectarea la o instanță a unui pipe.

Funcția **`CreateNamedPipe`** creează o instanță a unui pipe cu nume şi returnează un handle pentru operații cu pipe-ul. Un proces server utilizează această funcție fie pentru a crea prima instanță a unui pipe cu nume şi a-i stabili atributele de bază fie pentru a crea o nouă instanță a unui pipe cu nume existent.

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

`lpName` este un pointer către un şir care identifică în mod unic pipe-ul de forma următoare: `\\.\pipe\nume_pipe`. `nume_pipe` poate include orice caracter în afară de backslash, inclusiv numere şi caractere speciale. Lungimea maximă a numelui este de 256 caractere. Numele nu este case sensitive. În caz de succes se returnează un handle către capătul serverului al unei instanțe a pipe-ului. În caz de eroare se returnează `INVALID_HANDLE_VALUE`. Pentru mai multe detalii se foloseşte `GetLastError`. Iată un exemplu de apel:

```
HANDLE hPipe;
hPipe = CreateNamedPipe(
"\\\\.\\pipe\\mypipe",      // name
        PIPE_ACCESS_INBOUND,// data goes from client to server only
```

```
        PIPE_TYPE_BYTE | PIPE_WAIT,  // byte stream
        PIPE_UNLIMITED_INSTANCES,    // any number of instances can be created
        BUFSIZE,            // out buffer size
        BUFSIZE,            // in buffer size
0,                          // default time out
NULL                        // security attributes
);
```

Cu ajutorul funcției **ConnectNamedPipe** serverul pipe aşteaptă conectarea unui proces client la o instanță a unui pipe.

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

`hNamedPipe` este handle-ul către capătul serverului al unei instanțe a pipe-ului. Acest handle este returnat de funcția `CreateNamedPipe`. În caz de succes funcția returnează o valoare diferită de zero, altfel returnează zero. Pentru informații detaliate se apelează `GetLastError`.

Dacă un client se conectează înainte de apelul `ConnectNamedPipe`, funcția returnează zero şi `GetLastError` returnează `ERROR_PIPE_CONNECTED`. Acest lucru se poate întâmpla dacă un client se conectează în intervalul dintre apelul funcției `CreateNamedPipe` şi apelul `ConnectNamedPipe`. În această situație, conexiunea dintre client şi server este bună, chiar dacă funcția returnează zero.

Observație: Comportarea funcției depinde de modul în care a fost creat handle-ul pipe-ului: blocant sau neblocant.

Dacă handle-ul pipe-ului este în *modul blocant*, `ConnectNamedPipe` nu se termină până când un client s-a conectat.

Dacă handle-ul pipe-ului este în *modul neblocant*, `ConnectNamedPipe` se termină imediat. În acest mod funcția returnează o valoare diferită de zero prima dată când este apelată pentru o instanță a pipe-ului care este deconectată de la un client anterior. Aceasta indică faptul că pipe-ul este acum disponibil pentru conexiunea cu un nou client. În toate celelalte cazuri pentru modul neblocant, valoarea întoarsă este zero. În aceste situații `GetLastError` returnează `ERROR_PIPE_LISTENING` dacă niciun client nu s-a conectat, `ERROR_PIPE_CONNECTED` dacă un client s-a conectat şi `ERROR_NO_DATA` dacă un client şi-a închis handle-ul de pipe dar serverul nu s-a deconectat. O conexiune validă între client şi server va exista numai după ce s-a primit "eroarea" `ERROR_PIPE_CONNECTED`.

Mod de lucru

Prima dată un server pipe apelează `CreateNamedPipe`, specificând numărul maxim de instanțe ale pipe-ului care pot exista simultan. Serverul poate apela `CreateNamedPipe` în mod repetat pentru a crea noi instanțe ale pipe-ului atât timp cât nu se depăşeşte numărul maxim. Dacă funcția se termină cu succes fiecare apel returnează un handle către capătul serverului al unei instanțe a pipe-ului cu nume.

Imediat ce un server pipe creează o instanță a unui pipe, un client pipe se poate conecta la ea apelând `CreateFile` sau `CallNamedPipe`. Dacă o instanță a pipe-ului este disponibilă, `CreateFile` returnează un handle către capătul clientului al instanței pipe-ului. Dacă nu există instanțe disponibile un client pipe poate folosi funcția `WaitNamedPipe` pentru a aştepta până când un pipe devine disponibil. Un server poate determina când un client se conectează la o instanță a pipe-ului apelând `ConnectNamedPipe`. Dacă

handle-ul pipe-ului este în modul blocant, `ConnectNamedPipe` nu se termină până când un client s-a conectat.

Când un client şi serverul termină de folosit o instanță a pipe-ului, serverul trebuie să apeleze mai întâi `FlushFileBuffers`, pentru a se asigura că toți octeții sau toate mesajele scrise în pipe sunt citite de client. `FlushFileBuffers` nu se termină până când clientul nu a citit toate datele din pipe. Serverul apelează apoi `DisconnectNamedPipe` pentru a închide conexiunea cu un client pipe. Această funcție invalidează handle-ul clientului în cazul în care nu a fost deja închis. Orice dată necitită din pipe este distrusă. După deconectarea clientului, serverul apelează funcția `CloseHandle` pentru a închide handle-ul său către instanța pipe-ului. Ca o alternativă, serverul poate folosi `ConnectNamedPipe` pentru a permite unui nou client să se conecteze la această instanță a pipe-ului.

Câteva funcții unice sistemului Windows sunt disponibile pentru lucrul cu pipe-uri. Funcția `PeekNamedPipe` poate fi folosită pentru citirea dintr-un pipe fără a scoate datele din el. Funcția `TransactNamedPipe` scrie un mesaj de cerere şi citeşte un mesaj de răspuns într-o singură operație, îmbunătățind performanța rețelei. Poate fi folosită cu pipe-urile bidirecționale dacă handle-ul pipe-ului deținut de procesul apelant este setat în mod de citire. Un proces poate obține informații despre un pipe cu nume apelând `GetNamedPipeInfo`, care returnează tipul pipe-ului, dimensiunile bufferelor de intrare şi ieşire şi numărul maxim de instanțe ce pot fi create pentru pipe-ul respectiv. `GetNamedPipeHandleState` dă informații despre modurile de citire şi aşteptare ale handle-ului pipe-ului, numărul curent de instanțe ale pipe-ului, şi alte informații pentru pipe-urile care comunică printr-o rețea. `SetNamedPipeHandleState` setează modurile de citire şi aşteptare ale unui handle de pipe. Pentru clienții pipe ce comunică cu un server la distanță, funcția controlează de asemenea şi numărul maxim de bytes ce pot fi adunați sau timpul maxim de aşteptare până la transmiterea unui mesaj (presupunând că handle-ul clientului nu a fost deschis cu modul write-through activat).

Quiz

Pentru auto-evaluare răspundei la întrebările din acest quiz.

Exerciții

Exerciții pre-laborator

Linux

- utilizând programul `top` sau `htop` (`apt-get install htop` pentru sistemele Debian) urmării procesele aflate în execuie

Windows

- cu `Windows Task Manager` (în Windows, combinaia de taste Ctrl+Shift+Esc) sau instalând `Process Explorer` de la Sysinternals urmării procesele aflate în execuie

Exerciții de laborator

Precizări

- Exercițiile sunt independente de platformă.
- Puteți întrebuința macro-ul `CHECK`, definit în `utils.h`, pentru testarea valorilor întoarse de funcții.
- Exercițiile 1-4 decurg unul din altul. De exemplu, după ce ați rezolvat task-ul 1, copiați fișierul `1.c` din directorul `1` în directorul `2` şi redenumiți-l `2.c`. La exercițiile 2-4 nu uitați să **aşteptați** în părinte terminarea copilului.

Linux / Windows

Utilizați arhiva de sarcini a laboratorului.

1. (**0.5p**) `system`
   - Realizați un program care să execute o comandă transmisă ca parametru folosind funcția de bibliotecă system.
   - Presupunând că programul vostru gestionează doar primul argument din linia de comandă, cum procedați pentru a trimite parametri comenzii respective? (ex: `ls -la`)?
   - **Hints**:
     - Citiți secțiunea Rularea unui program executabil
     - Putei folosi label-ul `error` i instruciunea `goto` pentru a iei cu eroare.
2. (**1.5p**) `fork`/`exec`, `CreateProcess`
   - Realizați un program care execută linia de comandă cu `fork`/`exec`, respectiv `CreateProcess`.
   - Primul argument al programului va fi numele unui program de lansat, următoarele fiind transmise mai departe ca argumente programului respectiv.
   - **Hints**:
     - Linux: Citiți secțiunea my_system
     - Windows: Citiți secțiunea Exec
     - Recomandăm folosirea `execvp` i copierea argumentelor într-un nou vector.
     - Nu uitai să ateptai în părinte procesul copil.
3. (**1.5p**) Redirectare
   - Modificai programul de la exerciiul 2 astfel încât să execute comenzi cu ieirea redirectată [4]
     în fiierul `3.out`.
   - De exemplu secvena: `./program "ls"` ar trebui să pună toată ieirea comenzii **ls** într-un fiier.
   - Dacă fiierul nu există va trebui creat. Dacă există trebuie trunchiat.
   - **Hints**:
     - Linux
       - imediat după `fork` dar înainte de `exec`, obinei un file descriptor la fiier i apoi facei un `dup2`.
       - Citiți secțiunea Copierea descriptorilor de fiier.
     - Windows
       - `CreateProcess` primete, prin intermediul argumentului de tipul `LPSTARTUPINFO`, handle-urile `hStdInput`, `hStdOutput` i `hStdError`.

· `LPSTARTUPINFO` este pointer la o structură de tipul `STARTUPINFO`.
· Citiți secțiunea Motenirea descriptorilor de resurse la CreateProcess.
· Nu uitai să activai flag-ul de motenire a descriptorului la crearea fiierului folosind `CreateFile` (structura `SECURITY_ATTRIBUTES`).

4. (**2p**) Pipe anonim, variabile de mediu.
   - Modificați programul de la exercițiul 3 astfel încât să utilizeze un pipe anonim, al cărui capăt de citire să corespundă intrării standard a copilului.
   - In procesul copil rulați, în locul parametrului primit la exercițiul 3, programul `child` din arhivă.
   - Părintele va scrie în pipe un şir de caractere pe care `child` îl va afişa la `stdout`.
   - Observați că procesul copil afişează valoarea variabilei de mediu `MYVAR`.
   - Definiți-o în părinte cu ce valoare doriți, dar înaintea creării copilului.
   - **Hints**:
     ◊ Linux
     · Citiți secțiunea Apelul pipe.
     · În procesul părinte închidei capătul de citire al pipe-ului, iar în procesul copil capătul de scriere.
     · Citiți secțiunea Variabile de mediu.
     ◊ Windows
     · Este necesar ca handle-ul de scriere sa nu poată fi moştenit in procesul creat (altfel, copilul, neavând acces la el, nu-l va putea închide si nu se va mai trimite EOF prin pipe).
     · Folosiți funcția `SetHandleInformation` pentru împiedicarea moştenirii. Citiți mai multe aici.
     · Citiți secțiunea Pipe-uri anonime.
     · Citiți secțiunea Variabile de mediu.

Pentru acasă

1. Pipe cu nume
   - Realizați două programe denumite `client` si `server` care interacționează printrun pipe cu nume.
   - FIFOul se numeşte `myfifo`. Dacă nu există, este creat de server.
   - Serverul trebuie rulat înaintea oricărui client.
   - Clientul primeşte ca argument la rulare un număr n. Va interoga, prin intermediul FIFOului, serverul, pentru a afla daca *n* este prim sau nu.
   - Serverul va afişa rezultatul la ieşirea standard.
   - **Hints**:
     ◊ Linux: Citiți secțiunea FIFO (Pipe-uri cu nume).
     ◊ Windows
     · Puteți porni de la exemplul din documentația `CreateNamedPipe`.
     · Citiți secțiunea Pipe-uri cu nume.

Soluții

Soluții exerciții laborator 3

Resurse utile

1. Wikipedia - Fork
2. Opengroup - Fork
3. About Fork and Exec
4. YoLinux Tutorial - Fork, Exec and Process Control
5. Wikipedia - Windows Handles and Data Types
6. MSDN: Processes and Threads
7. C++ CreateProcess example
8. MSDN: Managing Heap Memory
9. Windows XP and 2003 Server Boot.ini options

Note

1. ^ Variabila de mediu **$HOME** este expandată de interpretorul de comenzi la directorul utilizatorului; presupunând că lucrăm cu utilizatorul *tavi*, atunci o posibilă expandare a variabilei **$HOME** ar putea fi **/home/tavi**
2. ^ În directorul **/etc/rc.d/** se găsesc scripturi de iniializare a sistemului; variabila de mediu **$RUNLEVEL** este setată la bootare la una din valorile: 1 (single user), 2 (multi-user cu reea), 3 (multi user), 5 (multi-user grafic - X11); pentru mai multe informaii: **man inittab**
3. ^ Apelul de sistem `ptrace` permite unui proces să observe i să controleze execuia altui proces; procesul care "urmărete" poate să inspecteze i să modifice imaginea executată de procesul "urmărit"; pentru mai multe informaii: **man ptrace**
4. ^ În general, pentru a redirecta output-ul unei comenzi într-un fiier shell-ul folosete următoarea sintaxă: `comanda [argumente] > nume_fisier`. Mai multe informaii în laboratorul următor.
5. ^ Verificai documentaia Microsoft în legătură cu opiunile disponibile pentru Boot.ini.