Laborator 3

Procese

Sisteme de Operare

10 Martie 2010

Outline

Ce este un proces?

- unitatea primitivă prin care sistemul de operare alocă resurse utilizatorilor
- program în execuție
- caracteristici
  - spațiu de adrese
  - unul sau mai multe fire de execuție
- informații asociate procesului (`Process Control Block`)
  - tabela de fișiere deschise
  - handler-ele pentru semnale
  - directorul curent
  - etc
- variabile de mediu

Operații cu procese

- creare
- așteptarea terminării
- terminare
- duplicarea descriptorilor de resurse

- organizare ierarhică
  - procesul `init` (`pid=1`) este strămoșul (indirect) al tuturor proceselor
- `fork`
  - **duplică** procesul curent
  - întoarce
    - `0`, în copil
    - `pid > 0`, în părinte
    - `-1`, în caz de eroare
- `exec`
  - **înlocuiește** imaginea procesului cu programul precizat
  - parametri: programul și parametrii săi
    - primul parametru este însuși numele programului!
  - întoarce doar în caz de eroare!
  - exemplu: `execlp("ls", "ls", "-la", NULL);`

Crearea unui proces (Windows)

- organizare neierarhică
- `CreateProcess`
  - **îmbină** cele două operaţii de pe Linux
  - întoarce `TRUE` în caz de succes
  - exemplu:

```
1  STARTUPINFO si;
2  PROCESS_INFORMATION pi;
3
4  ZeroMemory( &si, sizeof(si) );
5  si.cb = sizeof(si);          // size of structure in bytes
6  ZeroMemory( &pi, sizeof(pi) );
7
8  BOOL bRes = CreateProcess(
9          NULL,                // No module name
10         "notepad.exe"        // Command line
11         NULL,                // Process handle not inheritable
12         NULL,                // Thread handle not inheritable
13         FALSE,               // Set handle inheritance to false
14         0,                   // No creation flags
15         NULL,                // Use parent's environment block
16         NULL,                // Use parent's starting directory
17         &si,                 // Pointer to STARTUPINFO structure
18         &pi);                // Pointer to PROCESS_INFORMATION structure
```

Așteptarea terminării unui proces (Linux)

- `waitpid`
  - suspendă execuția procesului apelant până când procesul (procesele) specificat în argumente fie s-au terminat, fie au fost oprite (`SIGSTOP`)
  - parametri
    - `pid`, întors de `fork`
    - `status`, în care funcția depune informația despre terminarea procesului așteptat
    - `options`, diferite opțiuni
- `WIFEXITED, WEXITSTATUS ...`
  - obțin modul și codul de ieșire ale procesului, examinând `status`, întors de `waitpid`

Așteptarea terminării unui proces (Windows)

- `WaitForSingleObject, WaitForMultipleObjects`
  - suspendă execuția procesului curent până când unul sau mai multe alte procese se termină
  - parametri
    - `hHandle`, handle-ul procesului așteptat, obținut din `pi.hProcess`, structura populată de `CreateProcess`
    - `dwMilliseconds`, numărul maxim de milisecunde la care se poate prelungi așteptarea, de obicei `INFINITE`
  - întoarce `WAIT_FAILED` în caz de eroare
- `GetExitCodeProcess`
  - determină codul de eroare cu care s-a terminat un anumit proces
  - parametri
    - `hHandle`, vezi mai sus
    - `lpExitCode`, codul de ieșire, populat de funcție. `STILL_ACTIVE` dacă procesul nu s-a terminat încă.
  - întoarce `TRUE` în caz de succes

Exemple de așteptare

- Linux

```c
waitpid(pid, &status, 0);

if (WIFEXITED(status))
        printf("Child %d terminated normally, with code %d\n",
                        pid, WEXITSTATUS(status));
```

- Windows

```c
DWORD dwExitCode;

if (WaitForSingleObject(hProcess, INFINITE) != WAIT_FAILED) {
        if (GetExitCodeProcess(hProcess, &dwExitCode))
                printf("Process terminated normally, with code %d\n",
                                dwExitCode);
}
```

Terminarea unui proces (Linux)

- `exit`
  - încheie procesul curent
  - parametri
    - `status`, codul de ieșire al procesului
  - consecințe
    - procesul apelant se va termina imediat
    - toți descriptorii de fișier ai procesului sunt închiși
    - copiii procesului sunt "înfiați" de `init`
    - părintelui procesului îi va fi trimis un semnal `SIGCHLD`. Tot acestuia îi va fi întoarsă valoarea `status`, ca rezultat al unei funcții de așteptare.
    - va scrie bufferele streamurilor deschise și le va închide

Terminarea unui proces (Windows)

- `ExitProcess`
  - încheie procesul curent
  - parametri
    - `uExitCode`, codul de ieșire al procesului
- `TerminateProcess`
  - încheie alt proces. **Nu** este recomandată.
  - parametri
    - `hProcess`, handle-ul procesului
    - `uExitCode`, codul de ieșire al procesului

Duplicarea descriptorilor de resurse (Linux)

- dup, dup2
  - studiați laboratorul trecut
- descriptorii din părinte se moștenesc, implicit, în copil
- exemplu:

```
1 fd = open("file.txt", O_WRONLY);
2
3 // child
4 if (fork() == 0)
5         write(fd, "Childish thoughts", 18);
```

  - **Atenție!** Nu confundați vizibilitatea variabilei `fd`, în copil, cu semnificația care i se atribuie. Faptul că un copil vede valoarea 7 pentru `fd`, nu înseamnă, neapărat, că descriptorul 7 este recunoscut acolo.
- aplicație: redirectarea stream-urilor standard în copil

Duplicarea descriptorilor de resurse (Windows)

- `DuplicateHandle`
  - permite obținerea unui duplicat al unui descriptor, într-un/dintr-un **alt** proces
  - util când se dorește obținerea unui descriptor
    - cu drepturi de acces diferite față de cel inițial
    - care nu va fi moștenit în procesele copil dintr-unul care va fi moștenit

Redirectarea stream-urilor procesului creat, în Windows

- descriptorii ce indica fisierele către care se face redirectarea trebuie să poată fi moșteniți în procesul creat
  - membrul `bInheritHandle`, al structurii `SECURITY_ATTRIBUTES`, pasate lui `CreateFile`, trebuie sa fie `TRUE`
- la crearea procesului, trebuie solicitată moștenirea descriptorilor
  - parametrul `bInheritHandles`, al lui `CreateProcess`, trebuie sa fie `TRUE`
- la crearea procesului, trebuie populată structura `STARTUPINFO`
  - membrii `hStdInput`, `hStdOutput`, `hStdError` trebuie setați la descriptorii corespunzători
  - membrul `dwFlags` trebuie setat la `STARTF_USESTDHANDLES`

Variabile de mediu (Linux)

- `int main(int argc, char **argv, char **environ)`
  - Parametrul `environ`: vector de siruri de caractere, de forma `VARIABILA = VALOARE`
- `getenv`
  - obține valoarea unei variabile de mediu
  - parametri
    - `name`
  - întoarce `NULL` dacă nu există
- `setenv`
  - stabilește valoarea unei variabile de mediu
  - parametri
    - `name`
    - `value`
    - `replace`, 1 dacă se dorește suprascrierea unei variabile existente
- `unsetenv`
  - înlătură o variabilă de mediu
  - parametri
    - `name`

Variabile de mediu (Windows)

- `GetEnvironmentStrings`
  - returnează `environment block`-ul utilizatorului (vezi `environ` din Linux)
- `FreeEnvironmentStrings`
  - eliberează spațiul întors de `GetEnvironmentStrings`
  - parametri
    - `lpszEnvironmentBlock`
- `GetEnvironmentVariable`
  - obține valoarea unei variabile de mediu
  - parametri
    - `lpName`, numele variabilei
    - `lpBuffer`, zona de memorie, unde va fi depusa valoarea
    - `nSize`, dimensiunea zonei de mai sus
- `SetEnvironmentVariable`
  - stabilește valoarea unei variabile de mediu
  - permite înlăturarea unei variabile, punând o valoare `NULL`
  - parametri
    - `lpName`
    - `lpValue`

Outline

Ce este un pipe?

- mecanisme de comunicare între procese, ce oferă acces de tip FIFO: datele se scriu la un capăt și sunt citite de la celălalt capăt
- sistemul de operare garantează sincronizarea între operațiile de citire și scriere la cele două capete
- două tipuri
  - **anonime**
    - pot fi folosite doar între procese înrudite
    - există doar în prezența proceselor care dețin descriptori către ele
  - **cu nume**

Pipe-uri anonime (Linux)

- `pipe`
  - creează un pipe
  - parametri
    - `filedes`, vector cu cei 2 descriptori, desemnând cele 2 capete (0 - citire, 1 - scriere)
- `read`, `write`
  - folosite pentru lucrul cu pipe-uri
- `close`
  - la închiderea tuturor descriptorilor de scriere în pipe, se va primi `EOF` (`read` va întoarce `0`) pe toate capetele de citire
- **Atentie!** Când se utilizează `fork`, descriptorii din părinte vor fi **duplicați** în copil, astfel că numărul necesar de închideri de descriptori se va dubla. Nesocotirea acestui aspect, și închiderea parțială a descriptorilor, conduce la blocaje în `read`.

Pipe-uri anonime (Windows)

- `CreatePipe`
  - creează un pipe
  - parametri
    - `hReadPipe`, inițializat de funcție cu descriptorul capătului de citire
    - `hWritePipe`, inițializat de funcție cu descriptorul capătului de scriere
    - `lpPipeAttributes`, al cărei parametru, `bInheritHandle`, stabilește dacă descriptorii vor putea fi moșteniți sau nu.
- `ReadFile`, `WriteFile`
  - folosite pentru lucrul cu pipe-uri
- `CloseHandle`
  - la închiderea tuturor descriptorilor de scriere în pipe, se va primi `EOF` (`read` va întoarce `0`) pe toate capetele de citire
- **Atentie!** Spre deosebire de Linux, în Windows, valorile descriptorilor `nu` sunt direct vizibile în procesul copil, și trebuie făcute cunoscute printr-o metoda alternativă (de exemplu, ca parametri în linia de comandă). O altă variantă este `redirectarea` stream-urilor standard către capetele pipe-ului.

Pipe-uri cu nume (Linux)

- `mkfifo`
  - creează un pipe cu nume, ca fișier ce poate exista **independent** de procesele care-l utilizează
  - parametri
    - `pathname`, calea din sistemul de fișiere
    - `mode`, permisiunile
- `open`, `read`, `write`, `close`
  - folosite pentru lucrul cu pipe-uri
- moduri de deschidere
  - blocant
  - neblocant

Pipe-uri cu nume (Windows)

- arhitectură client-server
- moduri de comunicare
  - flux de octeți
  - mesaj
- `CreateNamedPipe`
  - creează un pipe cu nume
  - apelată de server
- `ConnectNamedPipe`
  - apelată de server, pentru a aștepta cereri de conectare de la clienți
- `CreateFile`
  - apelată de client
- `ReadFile`, `WriteFile`
  - folosite pentru lucrul cu pipe-uri