Procese și fire de execuție

Tema 5 – Sisteme de Operare

Autori:
Popescu Laurențiu, Ghimie Liliana,
Larisa Mazilu, Balanişcu Dragoş

Cuprins

Autori: **Popescu Laurenţiu – Capitolele 1 şi 2**

Ghimie Liliana – Capitolele 3 şi 4

Mazilu Larisa – Capitolele 4 şi 5

Balanişcu Dragoş – Capitolele 6 şi 7

Concepte fundamentale

Chiar dacă un sistem de operare execută la un moment dat mai multe programe simultan, în realitate, un procesor poate rula un singur proces, acestea fiind executate pe rând după un anumit algoritm de planificare. La nivelul aplicațiilor programele apar ca și cum ar fi executate simultan.

Utilizatorul poate rula, spre exemplu, în acelasi timp un player de muzică, un browser de internet şi un wordpad. Toate aceste programe care sunt în execuție se numesc procese şi sunt secvente de instrucțiuni care rulează într-o maniera secvențiala, concurând pentru folosirea resurselor calculatorului, precum procesorul, memoria sau hard-diskul. Fiecare dintre aceste procese este o entitate distinctă şi independentă de celalalte procese care se mai află în execuție în acel moment.

Procesul reprezintă un program executat şi pus în actiune. Acesta este alcătuit din două elemente: informațiile necesare pentru planificare la rulare (scheduling information) şi spațiul de adresă propriu adică zona de memorie în care se află codul programului, datele şi stiva.

Un sistem de procese ierarhizat este necesar pentru a putea organiza într-un mod mult mai accesibil utilizatorilor procesele ce se deruleaza într-un sistem. Astfel, primul proces pornit

în Linux este procesul *init.* Acest proces este părintele tuturor proceselor ce rulează pe acel sistem. Toate procesele care au rezultat dintr-un părinte se numesc *procese fiu*. [1]

Sistemul de procese ierarhizate presupune că:

- Fiecare proces din sistem are un proces care l-a creat, numit proces părinte (sau tata) şi de la care "moşteneste" un anumit ansamblu de caracteristici (cum ar fi proprietarul, drepturile de acces, s.a.), şi poate crea, la randul lui, unul sau mai multe procese fii.
- Fiecare proces are asignat un PID (process identification), ce este un număr întreg pozitiv şi care este unic pe durata vieții acelui proces (în orice moment, nu există în sistem două procese cu acelaşi PID).

În sistemele de operare UNIX, *PID*-ul unui proces (*child proces*) fiu este aflat prin funcția *fork()* adresată procesului părinte (*paerent process*).

În programe, de regulă părinte, PID-ul poate fi folosit ca atribut de către funcții de control (*waitpid() sau kill()*) pentru a executa anumite acțiuni asupra procesului specificat. Dacă sistemul de operare are implementat suport pentru *procfs*, atunci în fişierul /proc/pid se găsesc informații despre procese.

În sistemele de operare Linux există două procese care se deosebesc de celelalte procese din sistem:

- procesul **idle** care are PID-ul zero (0) şi nu se termină niciodata
- procesul **init process** care are PID-ul 1 şi nu face nimic altceva decât să aştepte după procesele child ca să se termine (să moară). Acesta nu are proces tată şi din el se trag toate procesele din acel sistem.

Procesele *idle* şi *init* se regăsesc şi în sistemele de operare Windows, aici fiind numite *System Idle Process*, respectiv *System*. (Linux Kernel)

Un aspect foarte important al proceselor este că acestea au întotdeauna spații de adresă distincte, fiecare dintre ele rulând ca şi cum toată memoria ar fi a sa. Din aceasta rezultă că procesele sunt izolate între ele şi nu se pot accesa unul pe celălalt în mod direct. Totuşi există comunicare între ele prin mecanisme IPC (Inter-Process Communication) precum: memorie partajată, socketuri, coadă de mesaje sau semafoare.

Spre exemplu, socketurile sunt concepute să funcționeze precum socketurile de rețea. Ele se pot compara cu niste fluxuri de biți, precum conexiunile de retea, singura diferența fiind că acestea rămân în interiorul computerului.

Pentru a se folosii eficient un procesor, acesta trebuie să execute concomitent mai multe procese. Deoarece un procesor nu poate să execute la un moment dat decat un singur proces, fiecarui proces activ în acel moment îi este alocată o bucata de timp (*time slice*) prin intermediul unui planificator (scheduler). Dacă un proces, de exemplu, foloseste o operatie I/O pentru o perioadă îndelungată şi nu pune procesorul în mod *idle* într-un anumit interval, se porneşte un cronometru care, atunci cand expiră, intrerupe procesul consumator de resurse şi alocă resursele unui nou proces. Astfel este asigurat faptul că un proces nu poate să dețină monopol asupra procesorului. [2]

Din ce am zis mai sus rezultă două idei fundamentale despre procese:

- Rulează independent pentru că au zone de cod, stiva şi date distinct
- Trebuie să fie planificate la execuție astfel ca ele să ruleze în paralel la nivelul aplicației

Firele de execuție (cunoscute ca şi procese uşoare – lightweight processes) sunt nişte fluxuri de instrucțiuni care se execută în interiorul unui proces şi pot fi văzute ca un program ce nu are spațiu de adresă propriu deşi se afla în execuție. Ele rulează în cadrul unui proces şi îi partajează spațiu de adresă al acestuia. Fiecare proces are cel puțin un fir de execuție care rulează în interiorul său. Uneori există mai multe fire de execuție în interiorul aceluiaşi proces când apare nevoia de lucru în paralel asupra aceloraşi date. De exemplu în cazul unui editor de texte unde atât programul principal de editare cât şi rutina care salvează o copie de siguranță a documentului elaborat de primul, lucrează asupra aceloraşi date, rezultă că vor exista doua fire de execuție în cadrul aceleiaşi aplicații. [1]

Deşi firele de execuție partajează spațiul de adresă al aceluiaşi proces, ele au nevoie de resurse individuale cum ar fi stiva, regiştrii şi contorul program care permit mai multor fire de execuție din cadrul aceluiaşi proces să urmeze căi diferite de instrucțiuni.

Un **proces** poate fi format din mai multe fire care sunt executate în paralel având în comun toate caracteristicile şi resursele principale procesului. Drept urmare, în interiorul unui proces **firele de execuție** sunt entități care rulează în paralel şi care impart între ele zona de date şi execută porțiuni distincte din acelasi cod. Zona de date fiind comună atrage după sine şi faptul că firele de execuție văd la fel toate variabilele procesului, orice modificare efectuată de

un fir fiind imediat văzută şi de celelalte. în general un proces propriu zis este de fapt un proces cu un singur fir de execuție.

Făcând o comparație între procese şi fire de execuție observăm ca procesele încurajează modularitatea codului ceea de duce la programe mai flexibile şi mai robuste în timp, îmbunătățesc securitatea aplicației, deoarece există module diferite izolate între ele (aflându-se în spații de adresă diferite nu îți pot accesa direct datele), iar o breşa de securitate poate fi mai greu exploatată pentru a duce la compromiterea întregii aplicații, pe când firele de execuție având bucăți de cod şi de memorie comune, o eroare se poate propaga în mult mai multe direcții ducând la o mai mare vulnerabilitate a sistemului. Deoarece mecanismele de comunicare între procese (IPC) pot îngreuna programarea aplicației, firele de execuție par a fi o soluție mai bună atunci când trebuie accesate frecvent zone de date comune. Un avantaj net superior al firelor de execuție în fața proceselor este paralelismul. Acest avantaj constă în posibilitatea de folosire mult mai eficientă a resurselor sistemului pentru o prelucrare mult mai rapidă a datelor. [4]

Execuția planificată a proceselor presupune că, la momente de timp determinate de algoritmul folosit, procesorul să fie "luat" de la procesul care tocmai se executa şi să fie "dat" unui alt proces. Această comutare între procese (process switching) este o operație consumatoare de timp, deoarece trebuie "comutate" toate resursele care aparțin proceselor: trebuie salvați şi restaurați toti regiştrii procesor, trebuie (re)mapate zonele de memorie care aparțin de noul proces etc.

La nivelul sistemului de operare, execuția în paralel a firelor de execuție este obtinută în mod asemănător cu cea a proceselor, realizându-se o comutare între fire, conform unui algoritm de planificare. Spre deosebire de cazul proceselor, însa, aici comutarea poate fi făcută mult mai rapid, deoarece informațiile memorate de către sistem pentru fiecare fir de execuție sunt mult mai puține decât în cazul proceselor, datorită faptului că firele de execuție au foarte puține resurse proprii. Practic, un fir de execuție poate fi văzut ca un numărător de program, o stivă şi un set de regiştri, toate celelalte resurse (zona de date, identificatori de fişier etc) aparținând procesului în care rulează şi fiind exploatate în comun. [3]

Kernelul reprezintă componenta centrală a sistemelor de operare. Este reponsabil pentru administrarea resurselor sistemului şi pentru administrarea accesului programelor la aceste resurse. Exista mai multe tipuri de kernel: monolitice, hibride, microkerneluri, nanokerneluri şi exokerneluri.

Kernelul stocheaza lista de procese într-o listă circulară dublu înlănțuită numită **task list** (lista de procese). Fiecare element din task list este un descriptor al fiecărui proces şi conține

toate informațiile despre respectivul proces. Descriptorul de proces este de tipul struct task_struck care este definit în linux/sched.h. [1]

Descriptorul de procese conține datele care descriu programul în execuție: fişierele deschise, spațiul de adresă al procesului, semnalele procesului, starea procesului şi multe alte informații utile.

Un **descriptor de fişiere** este un index pentru intrările dintr-o structură de date ce este rezidentă în kernel şi conține detalii despre toate fişierele deschise de către sistem. Acest descriptor este necesar pentru că în acelaşi moment două aplicații să nu acceseze acelaşi fişier şi astfel să se ajungă la coruperi de fişiere sau la pierderi de date. [3]

Există trei valori pentru descriptorul de fişiere:

- #0 – Standard input (stdin);
- #1 – Standard output (stdout);
- #2 – Standard error (stderr).

Apelurile de sistem de gestiune a proceselor în Linux

În Linux procesele sunt organizate ca o ierahie având la bază procesul obligatoriu şi unic, **init**. Procesul **init** este creat la pornirea sistemului de operare şi lansează în execuție alte procese ca, de exemplu, procesele consolelor de logare ale utilizatorilor (de exemplu **bash**) sau diverse programe care rulează în fundal cunoscute şi sub numele de daemoni (server web, server de mail, sshd etc). şi acestea au la randul lor posibilitatea să lanseze în execuție alte aplicații în funcție de necesitate sau la intervenția utilizatorilor care rulează diverse programe. Rezultă de aici ca mecanismul de gestiune al proceselor este bazat pe un model care are la baza relația între procese de tip **părinte-copil.**

Linux foloseşte pentru implementarea acesti model funcția *fork()* pentru a creea o copie exactă a programului aflat în execuție rezultând astfel un nou proces – procesul copil.

Apelurile API pentru gestionarea job-urilor şi ale proceselor

Apelurile de sistem puse la dispoziție de sistemul de operare LINUX pentru gestionarea proceselor sunt: **fork** şi **exec** pentru creearea şi respectiv modificarea imaginii unui proces, **wait** şi **waitpid** pentru aşteptarea terminării unui proces şi **exit** pentru terminarea unui proces; pentru copierea descriptorilor de fișier există în LINUX apelurile de sistem **dup** şi **dup2**; pentru lucrul cu variabilele de mediu biblioteca standard C pune la dispoziție apelurile **getenv**, **setenv** şi **unsetenv** precum şi un pointer la tabela de variabile de mediu **environ**. Fiecare proces are un spațiu de adrese de 4GB care este împărțit astfel: 3 GB pentru alocarea procesului, iar în celălălt GB pot fi adresate structurile şi simbolurile sistemului de operare în mod protejat. Astfel fiecare proces poate "vede" sistemul de operare în spațiul său de adrese însă acea zonă nu poate fi accesată decât prin intermediul apelurilor de sistem (procesorul este comutat în modul privilegiat). [4]

Apelurile de sistem puse la dispoziție de Windows (WIN32 API) pentru gestionarea proceselor sunt: **CreateProcess** şi variații ale acestuia pentru crearea unui proces; **WaitForSingleObject** şi alte funcții de aşteptare pentru aşteptarea terminării unui proces; **ExitProcess** pentru terminarea procesului curent; **TerminateProcess** pentru terminarea unui alt proces din sistem. Pentru duplicarea descriptorilor de resurse între procese se apelează funcția **DuplicateHandle**. Pentru citirea sau modificarea unei variabile de mediu putem folosi **GetEnvironmentVariable** şi **SetEnvironmentVariable** precum şi **GetEnvironmentStrings** care întoarce un pointer la tabela de variabile de mediu. Fiecare proces are un spațiu de adrese de 4GB care este împărțit astfel: 2GB pentru alocarea aplicației şi în ceilalți 2GB pot fi adresate structurile şi simbolii sistemului de operare în mod protejat. Opțional se pot aloca 3GB proceselor utilizator şi doar 1GB pentru sistemul de operare. Astfel fiecare proces poate "vede" sistemul de operare în spațiul său de adrese însă acea zonă nu poate fi accesată decât prin intermediul apelurilor de sistem. [3]

Comparație între implementarea proceselor şi firelor de execuție la Windows şi Linux

Procese în Linux

Rularea unui program executabil

Cel mai simplu mod prin care se poate rula un proces este prin folosirea funcției de bibliotecă **system**

int system(const char*command);

Apelarea acestei funcții are ca efect execuția sub forma unei comenzi shell comanda reprezentată prin stringul de caractere *command.* Implementarea **system**: este creat un nou process cu fork; procesul copil rezultat execută prin intermediul exec programul sh cu argumentele –c “command”, timp în care părintele aşteaptă terminarea procesului copil. [6]

Creearea unui proces

În Linux singurul mod prin care se poate creea un nou proces este prin apelul de sistem **fork**. Efectul acestui apel este creearea unui nou proces, copie a procesului care a apelat fork. Difera doar PID-ul proceselor, noul proces primind un alt PID de la sistemul de operare. [6]

Înlocuirea imaginii unui proces

Familia de funcții **exec** execută un nou program în care inlocuiesc imaginea procesului curent cu cea dintr-un fişier (executabil). Spațiul de adrese este înlocuit de unul nou special creat pentru execuția fişierului. De asemenea, vor fi reinitializați regiştrii hardware ai procesorului IP (EIP/RIP) şi SP (ESP/RSP) şi regiştrii generali. Vor fi setate la valorile implicite măstile de semnale ignorate şi blocate, ca şi handler-ele semnalelor. PID-ul şi descriptorii fişierelor ce nu au setat flagul CLOSE_ON_EXEC vor rămâne neschimbați (deoarece implicit flagul CLOSE_ON_EXEC nu este setat). [7]

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

EXECV execută programul descris de pointerul către şirul de caractere filename; vectorul argv e format din pointeri către şiruri de caractere ce descriu argumentele cu care programul reprezentat de filename va fi executat; ultimul element al vectorului trebuie să fie setat pe NULL. Calea catre program trebuie să fie completă pentru că nu se caută în PATH. Primul argument este numele programului.

int execl(const char *filename, const char *arg0, ...);

La EXECL argumentele vor fi date ca parametri ai funcției. Ultimul parametru trebuie să fie NULL, iar primul argument trebuie să fie numele programului.

int execve(const char *filename, char *const argv[], char *const env[]);

EXECVE este la fel ca execv, dar primeşte un vector de pointeri la şiruri de caractere care descriu variabile de mediu. Ultimul element al vectorului env trebuie să fie setat pe NULL, iar şirurile de caractere trebuie să fie de forma “VARIABILA = VALOARE”.

int execle(const char *filename, const char *arg0, ..., char *const env[]);

EXECLE este la fel ca execl, dar primeşte un vector de pointeri la şiruri de caractere care descriu variabile de mediu. Ultimul element al vectorului env trebuie să fie setat pe NULL, iar şirurile de caractere trebuie să fie de forma “VARIABILA = VALOARE”.

int execvp(const char *filename, char *const argv[]);

EXECVP este asemănătoare cu execv, dar calea programului nu mai trebuie să fie absolută pentru că va fi căutat în toate directoarele specificate de variabila de mediu PATH.

int execlp(const char *filename, const char *arg0, ...);

EXECLP este asemănătoare cu execl, dar calea programului nu mai trebuie să fie absolută pentru că va fi căutat în toate directoarele specificate de variabila de mediu PATH.

Daca se foloseşte orice funcție din familia **exec**, atunci este necesară includerea headerului **unistd.h**. [6]

Aşteptarea terminării unui proces

Familia de funcții **wait** suspendă execuția procesului apelant până când procesul (procesele) specificate în argumentele acesteia fie s-au terminat, fie au fost oprite (SIGSTOP).

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

Procesul apelant va fi blocat până când procesul cu PID-ul pid se va termina sau va fi oprit şi el la rândul lui.
Valoarea argumentului pid poate fi:

- >0 atunci cand reprezintă PID-ul unui proces;
- 0 (WAIT_MYGRP) şi atunci identifică orice proces copil din grupul de procese din care face parte procesul apelant;
- -1 (WAIT_ANY) orice proces copil.

Parametrul options poate fi setat cu unul sau mai multe din următoarele flaguri (combinate prin SAU pe biți):

- WNOHANG – dacă nici unul din procesele specificate nu este terminat, se va ieşi imediat din apelul waitpid (procesul care apelase waitpid nu se blochează);
- WUNTRACED – waitpid se întoarce şi dacă unul dintre procesele specificate s-a oprit, iar informațiile despre această oprire nu au fost încă raportate; dacă acest flag este setat, waitpid se întoarce şi pentru procesele copil care s-au oprit, dar care nu se află între

procesele “urmărite” de procesul apelant; informațiile despre copii “urmăriți” sunt furnizate chiar dacă acest flag nu este precizat.[4]

Funcția va întoarce PID-ul procesului a cărui stare este raportată; informațiile de stare sunt depuse în locația către care indică status. Funcția se va bloca dacă nici unul din procesele specificate nu este încă terminat (sau dacă nici unul din ele nu este oprit şi s-a setat WUNTRACED), mai puțin atunci când este setat WNOHANG, caz în care funcția întoarce 0. Atunci când este eroare se va întoarce -1, iar errno este setat pe:

- EINTR apelul a fost întrerupt de un semnal
- ECHILD nu s-a specificat un PID de proces copil valid
- EINVAL este specificat un flag invalid pentru options

Starea procesului interogat se poate afla prin examinarea statusului cu următoarele macrodefiniții:

WIFEXITED(status) întoarce o valoare diferită de 0 dacă procesul interogat s-a terminat prin exit

WEXITSTATUS(status), dacă WIFEXITED(status) e diferit de 0, arată codul de terminare a procesului

WIFSIGNALED(status) întoarce o valoare diferită de 0 dacă procesul interogat s-a terminat datorită unui semnal netratat

WTERMSIG(status), dacă WIFSIGNALED(status) e diferit de 0, arată numărul semnalului datorită căruia procesul s-a terminat

WCOREDUMP(status) întoarce o valoare diferită de 0 dacă procesul interogat a generat un core

WIFSTOPPED(status) întoarce o valoare diferită de 0 dacă procesul interogat este oprit

WSTOPSIG(status), dacă WIFSTOPPED(status) e diferit de 0, întoarce numărul semnalului care a oprit procesul

Observație

Aceste macrodefiniții primesc ca argument buffer-ul în care se află informația de stare (adică un int), şi nu un pointer către acest buffer.

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

Varianta simplificată care aşteaptă orice proces copil să fie terminat. Este echivalentă cu:

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

Pentru a folosi wait sau waitpid trebuie incluse header-ele sys/types.h şi sys/wait.h.[7]

Terminarea unui proces

Pentru terminarea procesului curent LINUX pune la dispoziție apelul de sistem **exit** (al cărui antet este descris în stdlib.h).

void exit(int status);

Procesul apelant va fi terminat imediat. Toți descriptorii de fişier ai procesului sunt închişi, copii procesului sunt "infiați" de procesul **init**, iar părintelui procesului îi va fi trimis un semnal SIGCHLD. Procesului părinte îi va fi întoarsă variabila status ca rezultat al unei funcții de aşteptare (wait sau waitpid).

Pentru terminarea unui alt proces din sistem, se va trimite un semnal către procesul respectiv prin intermediul apelului de sistem **kill**.[4]

Copierea descriptorilor de fişier

int dup(int oldfd);

Duplică descriptorul de fişier oldfd şi întoarce noul descriptor de fişier sau -1 în caz de eroare.

int dup2(int oldfd, int newfd);

Duplică descriptorul de fişier oldfd în descriptorul de fişier newfd; dacă newfd există, mai întâi va fi închis fişierul asociat. Întoarce noul descriptor de fişier sau -1 în caz de eroare.

Descriptorii de fişier sunt de fapt un index în tabela de fişiere deschise, ce conține pointeri către o structură ce conține informații despre fişier. Duplicarea unui descriptor de fişier înseamnă duplicarea intrării din tabela de fişiere deschise (adica un pointer). Din acest motiv, toate informațiile asociate unui fişier (lock-uri, poziția în fişier, flagurile) sunt partajate de cei doi file descriptori. Ceea ce înseamnă că operațiile ce modifică aceste informații pe unul din file descriptori (de ex. lseek) sunt vizibile şi pentru celălat file descriptor (duplicat) şi invers.

Observatie: Există şi o excepție: flagul CLOSE_ON_EXEC nu este partajat (acest flag nu este ținut în structura menționată mai sus).

Pentru a putea folosi dup şi dup2 trebuie inclus header-ul unistd.h. [8]

Moştenirea descriptorilor de fişier după operații fork/exec

Descriptorii de fişier ai procesului părinte se moştenesc în procesul copil în urma apelului fork. După un apel exec descriptorii de fişier sunt păstrați de asemenea, mai puțin aceia dintre ei care au setat flagul CLOSE_ON_EXEC.

Pentru a seta flagul CLOSE_ON_EXEC se foloseşte funcția fcntl cu un apel de genul: fcntl(file_descriptor, F_SETFD, FD_CLOEXEC);

Pentru a putea folosi funcția fcntl trebuie incluse header-ele unistd.h şi fcntl.h.

Trei dintre descriptorii de fişier sunt mai importanți:

STDIN_FILENO
toate programele obişnuite conțin 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 obişnuite au acest file descriptor care are valoarea 1; el reprezintă ieşirea standard; toate apelurile de genul printf vor genera output la terminal
STDERR_FILENO
toate programele obişnuite au acest file descriptor care are valoarea 2; el reprezintă ieşirea standard de eroare.[7]

Variabile de mediu

char **environ;

Un vector de pointeri la şiruri de caractere, ce conțin variabilele de mediu şi valorile lor. Vectorul e terminat cu NULL. Şirurile de caractere sunt de forma “VARIABILA=VALOARE”.

char* getenv(const char *name);

Întoarce valoarea variabilei de mediu denumite name, sau NULL dacă nu există o variabilă de mediu denumită astfel.

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

Adaugă în mediu variabila cu numele name (dacă nu există deja) şi îi setează valoarea la value. Dacă variabila există şi replace e 0, acțiunea de setare a valorii variabilei e ignorată; dacă replace e diferit de 0, valoarea variabilei devine value.

int unsetenv(const char *name);

Şterge variabila denumită name din mediu. [4]

Depanarea unui proces

Pe majoritatea sistemelor de operare pe care a fost portat, gdb nu poate detecta când un proces realizează o operație fork(). Atunci când programul este pornit, depanarea are loc exclusiv în procesul inițial, procesele copii nefiind ataşate debugger-ului. În acest caz, singura soluție este introducerea unor întârzieri în executia procesului nou creat (de exemplu, prin

apelul de sistem sleep()), care să ofere programatorului suficient timp pentru a ataşa manual gdb-ul la respectivul proces, presupunând că i-a aflat PID-ul în prealabil.

Pentru a ataşa debugger-ul la un proces deja existent, se foloseşte comanda attach, în felul următor:

(gdb) **attach *PID***

Această metodă este destul de incomodă şi poate cauza chiar o funcționare anormală a aplicației depanate, în cazul în care necesitățile de sincronizare între procese sunt stricte (de exemplu operații cu time-out).

Din fericire, pe un număr limitat de sisteme, printre care şi Linux, gdb permite depanarea comodă a programelor care creează mai multe procese prin fork() şi 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, şi atunci noul proces creat va fi depanat în continuare. Se poate observa că în această manieră debugger-ul este ataşat la un moment dat doar la un singur proces, neputând urmări mai multe simultan.

Cu toate acestea, gdb poate *ține evidența* tuturor proceselor create de către programul depanat, deşi 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 ataşa unui singur proces la un moment dat (comportament implicit), sau off, caz în care gdb se ataşează la toate procesele create în timpul execuției, şi le suspendă pe acelea care nu sunt urmărite, în funcție de valoarea setării follow-fork-mode.

Comanda info forks afişează informații 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, aşa cum apare în lista afişată de comanda info forks.

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

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

Diferența dintre cele două comenzi este că detach fork lasă procesul să ruleze independent, în continuare, în timp ce delete fork îl încheie. [1][6]

Procese în Windows

Introducere

Un proces este definit ca fiind un program aflat în execuție ale cărui instrucțiuni sunt parcurse una câte una dar la momente de timp diferite. În momentul creării, proceselor le sunt alocate anumite resurse de care au nevoie pentru a-şi îndeplini sarcinile. Starea unui proces este definită ca activitate curentă a procesului.În timpul execuției procesul işi poate schimba starea. Un proces este reprezentat în cadrul sistemului de operare prin blocul de control al procesului (BCP).Acest bloc conține informații referitoare la: starea procesului, valoarea contorului program şi a registrelor UC, informații despre gestionarea memoriei, informații de stare referitoare la operațiile de I/O.[8]

Crearea unui proces

Sistemul de operare poate crea procese sau poate desființa procese.În timpul execuției, un proces poate crea unul sau mai multe procese noi. Procesele create sunt numite ”copii” iar procesul care le creaza este numit ”părinte”.

Procesul nou creat are şi el nevoie de resurse, pe care le ia direct de la sistemul de operare sau de la procesul ”părinte”.

Procesul ”părinte” poate aloca spre utilizare numai o parte a resurselor, cum ar fi memoria sau fişierele, prevenind astfel supraîncărcarea sistemului din cauza creării prea multor procese ”copii”. [8]

Funcția CreateProcess

Creează un proces nou.Noul proces rulează în contextul de securitate oferit de procesul apelat. Daca procesul apelat ține loc de un alt user, procesul nou utilizează semnul procesului apelat, nu semnul celui care ține loc userului. Pentru a rula noul process în contextul de

securitate al userului reprezentat de semnul înlocuitor, se foloseşte funcția **CreateProcessAsUser** sau funcția **CreateProcessWithLogonW**.

Sintaxa:

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

Parametrii:

lpApplicationName – Numele modulului care va fi executat.Acest modul poate fi o aplicație proprie Windows.Poate fi şi alt tip de modul(de exemplu, MS-Dos sau OS/2), dacă subsistemul potrivit este disponibil pe calculator. Parametrul *lpApplicationName* poate fi NULL. În acest caz numele modulului trebuie să fie primul semn delimitat de un spațiu, la sfârşitul şi la începutului argumentului.

lpCommandLine -Linia de comandă poate fi executată. Lungimea maximă a acestui şir este 32 mii caractere. Daca parametrul " *lpCommandLine* " este NULL , atunci avem o limitare de caractere la "MAX_PATH". Versiunea Unicod a acestei funcții, "**CreateProcessWithLogonW** " poate modifica conținutul acestui şir.Drept urmare aceşti parametrii nu pot fi pointeri la memoria "read-only" pentru că s-ar realiza o violare a accesului.

lpProcessAttributes-Un pointer la structura **SECURITY_ATTRIBUTES** care determină daca ceea ce returnează procesul poate fi moştenit la noul proces provenit din acesta. Dacă acest paramentru este NULL atunci rezultatul returnat de process nu poate fi moştenit.

lpStartupInfo – un pointer la structura " Startup Info".

" lpProcessInformation "- un pointer la structura "ProcessInformation " care primeşte informații identificatoare despre noul process.

" lpCurrentDirectory " Calea către directorul curent pentru proces. Dacă parametrul este NULL atunci noul proces va avea acelaşi director şi drive ca şi procesul apelat.

“ *lpEnvironment”* – un pointer către mediul alocat pentru procesul nou format. Daca parametrul este NULL, noul proces foloseşte mediul vechiului proces.

Daca funcția compilează atunci răspunsul va fi o valoare diferită de 0, dacă are eroare atunci întoarce 0. [10]

Funcția CloseHandle

Închide un descriptor deschis.

Sintaxa:
```
BOOL WINAPI CloseHandle(
  __in  HANDLE hObject
);
```

Funcția nu are parametrii. Dacă funcția compliează, valoarea întoarsă este diferită de 0, dacă nu, est 0. Funcția CloseHandle închide descriptor către urmatoarele obiecte:

Access token;
Communications device;
Console input;
Console screen buffer;
Event;
File;
File mapping;
I/O completion port;
Job;
Mailslot;
Memory resource notification, etc.[10]

Aşteptarea terminării unui proces

Funcția WaitForInputIdle

Aşteaptă până când procesul specificat aşteaptă inițializarea userului sau până când intervalul alocat s-a scurs.

Sintaxa:
```
DWORD WINAPI WaitForInputIdle(
  __in  HANDLE hProcess,
  __in  DWORD dwMilliseconds
```

);
Parametrii:
hProcess un descriptor către proces. Dacă procesul este o aplicație de consolă sau nu are un mesaj prestabilit atunci funcția raspunde imediat.

Return code/value Description

0	wait s-a terminat cu succes
WAIT_TIMEOUT	wait s-a terminat deoarece time-slotul s-a terminat
WAIT_FAILED	o eroare a avut loc. [10]

Terminarea unui proces

Terminarea unui proces poate fi:

- Terminare normala cand un proces se termină după executarea ultimei sale instrucțiuni;
- Terminare forțată când procesul se încheie înainte de parcurgerea tuturor instrucțiunilor sale.

Când ”copilul” a depăşit limita de folosire a resurselor care i-au fost alocate sau nu mai este util din punct de vedere al funcției pe care o realiza, procesul ”părinte” poate cere încheierea forțată a execuției sale.

Un proces independent este un proces a cărui execuție nu poate afecta sau nu poate fi afectată de execuția altor procese din sistem.[8]

Funcția TerminateProcess

Închide procesul specificat precum şi toate procesele deschise de acesta.

Sintaxa:
BOOL WINAPI TerminateProcess(
  __in HANDLE *hProcess*,
  __in UINT *uExitCode*
);

Parametrii :

hProcess un descripor către procesul ce trebuie terminat. Acesta trebuie să aibă accesul corespunzător.

uExitCode codul de ieşire ce va fi utilizat de către procesul care va fi închis ca rezultat al acestei comenzi. Foloseşte funcția **GetExitCodeProcess** pentru a găsii această valoare.

Valoarea returnată este diferită de 0 dacă funcția compilează, dacă nu ea este 0. [10]

Funcția GetExitCodeProcess

Întoarce valoarea de sfarşit a procesului terminat.
Sintaxa:

```
BOOL WINAPI GetExitCodeProcess(
  __in   HANDLE hProcess,
  __out  LPDWORD lpExitCode
);
```

Parametrii:

hProcess- un descriptor pentru proces. Acesta trebuie să aibă accesul corespunzator.

lpExitCode- Un pointer către o variabilă prin care primeşte valoarea de sfârşit a procesului terminat. Dacă procesul specificat încă nu a fost terminat şi functia compilează, atunci valoare returnată este STILL_ACTIVE.

În rest, dacă funcția compilează valoarea este diferită de 0, daca nu, este 0.[11]

Variabile de mediu

Funcția GetEnvironmentStrings

Întoarce variabilele mediului corespunzător pentru procesul curent.
Sintaxa:

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

Daca funcția compilează atunci răspunsul este un pointer către environment block al procesului curet, dacă nu atunci răspunsul este NULL.[10]

Algoritmul de planificare în Linux şi Windows şi diferența fața de modelul folosit în Unix

Introducere

Planificarea thread-urilor sau proceselor constă în strategia folosită de către sistemul de operare pentru a decide la un moment dat care thread trebuie executat şi pentru cât timp. Componenta sistemului de operare care realizează planificarea se numeşte *planificatorul* de thread-uri sau procese.

Sistemul de operare Linux foloseşte un *mecanism de planificare* bazat pe *priorități*. Thread-urilor le este asociată câte o prioritate şi este ales întotdeauna pentru execuție thread-ul cu prioritatea cea mai mare. Se realizează o listă cu thread-uri care candidează pentru obținerea procesorului.Decizia de planificare este luată în urma implementării a trei strategii diferite de inserare a thread-urilor în listele de priorități. Strategiile se mai numesc şi politici de planificare.[7]

Prioritatea şi politica de planificare

Există două atribute ale unui thread care fac ca respectivul thread să fie tratat într-un mod special de către planificatorul de thread-uri. Aceste atribute sunt **prioritatea** şi **politica** de planificare.

Planificarea thread-urilor pentru execuție se face pe baza priorității. Politica de planificare reprezintă strategia ce defineşte modul în care thread-urile care au aceeaşi prioritate vor fi executate pe procesoarele disponibile. În Linux sunt trei politici de planificare, una pentru thread-urile aplicațiilor obişnuite şi celelalte două pentru aplicațiile de timp real. Politica de planificare determină pentru fiecare thread modul în care el este inserat şi avansează în lista corespunzătoare priorității statice pe care o are asociată. [6]

Politica de planificare în Linux

SCHED_FIFO- First In-First Out -este o politică ce poate fi folosită doar la threaduri cu prioritate statică mai mare ca 0. Principiul de funcționare al acestei politici este: primul venit, primul servit.

SCHED_FIFO este un simplu algoritm de planificare fără time slicing.

Regula care se respectă la această politică este ca: Un proces SCHED_FIFO care a fost întrerupt de un alt proces cu prioritate mai mare ca a sa va sta în capătul listei de priorități şi-şi va termina execuția de îndată ce toate procesele cu priorităţii mai mari sunt blocate din nou. Când acest proces devine rulabil el va fi pus la sfârşitul listei pentru prioritatea sa.[1]

SCHED_RR: tot ceea ce a fost descris mai sus pentru SCHED_FIFO se aplică şi pentru această politică, cu excepția faptului că fiecărui proces îi este îngăduit să ruleze pentru un interval

maxim de timp. Dacă a depăsit această perioadă de rulare, procesul va fi mutat la coada listei de prioritate pentru execuție.Lungimea perioadei de rulare se poate afla folosind **sched_rr_get_interval.**[1]

SCHED_OTHER: poate fi folosită doar la prioritatea statica de grad 0.Procesul care va rula va fi ales din cadrul acestei liste printr-un procedeu dinamic care ține de **setpriority** system call.Aceasta asigură un proces funcționabil corect pentru toate procesele *SCHED_OTHER.*[1]

Politica de planificare şi algoritmi în Windows

Planificarea este o operație foarte importanta pentru funcționarea eficientă a calculatorului.Aceasta operație nu numai că permite mai multor programe să funcționeze în acelasi timp dar scade şi gradul de utilizare al procesorului.

Algoritmi de planificare a UC

În funcție de criteriile:

- gradul de utilizare al UC ;
- numărul de procese executate într-un interval de timp precizat ;
- durata totală a execuției unui proces ;
- durata de aşteptare;
- durata de răspuns, se poate aplica unul din urmatorii algoritmi de planificare,al căror scop este de a optimiza criteriul luat în considerare.

Algoritmul FCFS-este un algoritm de tip FIFO : adică primul proces care cere alocarea UC acela este servit.Celălalte procese sunt servite succesiv dar nu înainte ca primul proces sa-şi termine execuția.

Acest algoritm este foarte simplu de implementat însa are dezavantajul că durata medie de aşteptare a proceselor nu este, în general, minimală ci poate varia în limite foarte largi.[1]

Algoritmul Round-Robin- este un algoritm de tip preemptiv.Un astfel de algoritm permite întreruperea execuție unui proces dacă în coada de procese a apărut un proces cu prioritate mai mare de execuție.

Alte caracteristici ale algoritmului sunt folosirea unei cuante de timp şi tratarea şirului ready ca fiind un şir circular. Algoritmul asigură un timp de aşteptare aproape egal pentru toate procesele din sistem.

Performanțele algoritmului depind de mărimea cuantei folosite.Dacă cuanta este foarte mare algoritmul se comportă asemanător cu FCFS.Daca cuanta este forte mică,fiecare din cele n procese se vor executa cu o viteza egala cu 1/n din viteza procesorului.[1]

Emularea programelor MS-DOS

DOS , abreviere de la **D**isk **O**perating **S**ystem. este un sistem de operare care ruleaza o singura aplicatie in acelasi timp, adica un sistem single-tasking, single-user (un singur utilizator existent)ce este bazat pe linia de comanda, care nu asigura executarea concurenta si nici impartirea resurselor intre mai multe procese. Sub controlul unui sistem de operare monotasking, in sistemul de calcul se poate executa un singur program; in acelasi timp;el va ramane activ din momentul lansarii executiei sale pana in momentul finalizarii; atat timp cat este in executie, programul are access la toate resursele sistemului de calcul.[11]

Prima versiune de DOS, numita PC-DOS, a fost creata de Microsoft in 1981 pentru IBM. Microsoft s-a folosit de acest sistem si apoi l-a vandut fiind compatibil pentru calculatoarele IBM-PC care nu erau fabricate de IBM. Aceasta a fost cea mai populara versiune de DOS. Deoarece DOS este un program simplu si original din sistemul de operare CP/M, exista si alte variante de DOS, mai mult sau mai putin compatibile cu MS-DOS/PC-DOS, ca de exemplu DR-DOS si FreeDOS..DOS`ul a fost cel mai popular sistem de operare pentru arhitectura x86 pana la aparitia sistemului Windows 95 [6]

Liniile Windows 9x si NT-Windows 9x s-a dezvoltat pe baza sistemului Windows 3.0 si a hibrizilor sai :Windows 3.1 si 3.11 Windows 9x si NT - Windows 9x, sunt sisteme pe 32 de biti in totalitate multi-tasking, dar in continuare partial hibrid iar Windows NT considerat un sistem complet nou. De aceea MS-DOS nu a disparut complet decat dupa ce Windows ME a fost inlocuit de Windows XP, sistem bazat pe Windows 2000 si Windows NT. DOS era folosit ca un Command Prompt, interpretor standard de comenzi , ca mediu initial de instalare pentru noile

sisteme de operare.. Ultimele versiuni de DOS au fost cea integrata in Windows Me , MS-DOS 8.00 in 2000, si cea disponibila separat de Windows, MS-DOS 6.22 din 1994.

Astazi, DOS este folosit ca sistem embedded in putine aplicatii, fiind incorporat in aplicatii industriale si in sisteme care inca nu au fost inlocuite din diferite motive, numite sisteme legacy. Totusi acest sitem de operare(DOS) este indepartat din aplicatiile industriale ,pierzand teren in fata sistemelor moderne precum Linux sau QNX, care vin cu mai multe avantaje -multi-tasking, POSIX si auun impact minim; in sistemele legacy este inlocuit odata cu uzura fizica sau morala a sistemelor sau functiile putine ce le ofera DOS(un exemplu de sistem legacy inlocuit recent sunt aplicatiile oferite de Ministerul Finantelor contribuabililor pentru calculul diverselor taxe si impozite, care au fost portate recent de pe MS-DOS pe Windows sau pe web).

DOS nu mai exista in versiunile de Windows contemporane derivate din Windows NT (2000, XP, 2003 si Vista), totusi, interfata cu linia de comanda are comenzi cu sintaxa similara dar este completata cu un numar de functii suplimentare. Programele de DOS ruleaza cu un grad limitat de compatibilitate pe Windows 2000 si XP pe 32 de biti, pe cand compatibilitate cu programele pe sistemele pe 64 de biti a fost total eliminata.

Dos porneste in general din prima partitie fizica a sistemului. La initializarea sistemului, se citeste ora si data curenta din BIOS ;daca acestea nu exista sau nu au fost setate, utilizatorul este invitat sa le seteze.; urmeaza apoi procesarea unui fiesier prin care se puteau initializa driverii de memorie, de CD-ROM,de placa de baza, de placa video, etc, numit CONFIG., abia apoi se proceseaza AUTOEXEC.BAT care este un shell script (batch) pre-definit. Dupa acestea, se prezintautilizatorului linia de comanda Command Prompt-ul - sub forma:

[Partitie curenta][:][Director curent (ce e de obicei root-ul disk-ului, \][simbolul promptului, adica >] – [9]

Sistemele MS-DOS mai noi de 3.0 nu au o procedura predefinita de inchidere. Pentru inchiderea sistemului, se apasa pur si simplu butonul de power de pe carcasa. Sistemele mai vechi de versiunea 3.0 ofera comanda park, pentru ampozitiona capul de citire a hard-discului intr-o pozitie safe, dupa care se actiona butonul de inchidere de pe carcasa. Dupa versiunea 3.0,noua tehnologie a hard-disk`urilor a facut aceasta operatiune necesara.

Un calculator fara partea software, doar cu partea hardware este o masa inerta de piese. Cand porneste, calculator trebuie sa execute un program special numit sistem de operare. Rolul sistemului de operare este sa ajute alte programe sa mearga avand grija de detaliile dezordonate de a controla partea hardware a calculatorului.[10]

Procesul de pornire a sistemului de operare: comparatie Windows şi Linux

Sistemul de operare este un ansamblu de programe ce asigură utilizarea optimă a resurselor logice şi fizice ale unui sistem de calcul. Scopul sau este sa gestioneze componentele hardware ale sistemului de calcul, de a coordona şi controla executarea programelor şi de a permite comunicarea i cu sistemul de calcul al utilizatorului.Fara un sistem de operare, un calculator nu are sens.De aceea,sistemul de operare este o componenta software importanta ce coordoneaza si supravegheaza activitatea sistemului de calcul si care se intereseaza de comunicarea dintre utilizator si sistemul de calcul. Sistemul de operare este impartit pe 2 nivele avand in vedere legaturile sale cu componentele hardware dar si organizarea programelor softwa:e – nivelul fizic si nivelul logic.

A. nivelul fizic include componenta firmware a sistemului de calcul; nivelul fizic oferă servicii privind lucrul cu componentele hardware ale sistemului de calcul şi cuprinde acele elemente care depind de structura hardware a sistemului;la acest nivel mai sunt incluse programe a căror execuție este indispensabilă, asa cum e programul care lansează încărcarea automată a sistemului de operare, la pornirea calculatorului ; la acest nivel, comunicarea cu sistemul de calcul se realizează prin intermediul sistemului de întreruperi, prin care se semnalează anumite evenimente apărute în sistem; la apariția unei întreruperi, controlul este dat unor rutine de pe nivelul următor al sistemului de operare. Programele care se execută la pornirea sistemului de calcul sunt:

* programul POST (Power-On Self-Test), care verifică starea de funcționare a sistemului de calcul şi programele de inițializare a activității sistemului

* drivere fizice care sunt programe ce fac posibila functionarea componentelor fizice ale sistemului de calcul; ele oferă posibilitatea de lucru cu configurația hardware de baza a sistemului de calcul : consola, tastatura, imprimanta, perifericele standard şi ceasul sistemului. Avantajul acestei soluții este că asigură independența software-ului de pe nivelul logic față de caracteristicile constructive ale componentelor hardware de bază, ele fiind tratate unitar, prin intermediul driverelor.

B. nivelul logic se ocupa de partea de programe a sistemului de operare şi oferă utilizatorului mijloacele prin care poate utiliza sistemul de calcul; comunicarea utilizatorului cu sistemul de calcul se realizează prin adresarea comenzilor sistemului de operare sau prin intermediul instrucțiunilor programelor pe care le execută;

comunicarea de la calculator la utilizator se realizează prin intermediul mesajelor pe care sistemul le transmite.Mesajele pot fi de erori la nivelul aplicatiilor, la nivelul sistemului, informatii legate de sistem,interfata grafica etc. [1]

Procesul de pornire a sistemului de operare este numit bootare. Calculatorul stie sa booteze deoarece intructiunile de bootare sunt construite in unul din cipurile sale , si anume in cipul BIOS.

Cipul PROM EPROM BIOS ii spune sistemului de bootare sa se uite intr-un loc fixat pe hard discul care are numarul cel mai mic, adica discul de boot pentru a gasi un program special numit incarcator de boot boot loader(in windows) [sub Linux incarcatorul de boot este numit LILO LInux LOader]. Incarcatorul de boot este apoi adus in memorie si pornit sau executat. .Rolul incarcatorului de boot este deci sa porneasca sistemul de operare ce va gestiona programele software utilizate, facand legatura intre utilizator si masina.

Incarcatorul de boot va porni sistemul de operare cautand in nucleu numit kernel,softul necesar ,prelucrcandu-l in memorie sau in core; core-urile sunt componente de baza ale unui sistem hardware performant; ele reprezinta nucleul procesorului., In cele din urma, se executa comenzile din kernel.Spunem ca se incarca kernelul atunci cand Linux`ul booteaza si se afiseaza "LILO" pe ecran ,urmat de o linie de puncte. Fiecare dintre puncte inseamna ca a incarcat alt bloc de disc din codul nucleului sau a kernelului.Important este faptul ca Linux nu foloseste deloc aceste blocuri de disc decat dupa ce a bootat. Linux`ul a fost initial conceput pentru PC`uri mai vechi cum ar fi intel x86 pe 8 octeti cu discuri mici, si de fapt nu poate accesa destul din disc pentru a incarca kernelul direct. Pasul cu incarcatorul de boot va lasa, de asemenea sa porneasca unul din cateva sisteme de operare din locuri diferite de pe disc .Windows`ul procedeaza identic..[10]

Dupa ce kernelul a pornit,sistemul de operare trebuie sa gaseasca restul componentelor hardware,pentru ca apoi sa se pregateasca sa ruleze programe. El face acest lucru uitandu-se in spatiul de I/O la porturi I/O (Intrare/Iesire), si nu la locatiile de memorie.Porturile i/o sunt adrese speciale pe magistrala care sunt probabile sa aiba placi controlere care le asculta pentru comenzi. Kernelul nu cauta oriunde, si datorita softului bine pus la punct pe care il poseda,, stie unde sa caute pentru a gasi daca apar componente hardware noi, prin raspunsul asteptat de la controlere., daca acestea exista..Acest proces se numeste autoprobare. Odata ce s-a incarcat kernelul, procesul de bootare continua cu init..Incarcarea kernelului este doar prima etapa (cateodata numita run level 1 (nivelul de rulare 1)). Init'ul porneste cateva procese care sa se ocupe de functionarea buna a sistemului de operare Linux..

Rolul procesului init este de obicei acela de a verifica ca harddisk`urile sunt in regula. Sistemele de fisiere pentru discuri sunt foarte fragile, astfel ca daca au fost deteriorate de o problema sau o neasteptata pana de curent, sunt motive intemeiate sa se ia masuri de recuperare inainte ca Unix/Linux/Windows sa inceapa complet. Urmatorul pas al lui init este de a porni cativa daemoni.. Un daemon este un program care asteapa sa faca diverse lucruri pe care le cere utilizatorul, aceasta asteptare consumand putine resurse. Aceste programe speciale mai au rolul de a coordona cateva cereri ce ar putea genera un conflict. Daemonii sunt programe special create care ruleaza constant si stiu despre toate cererile utilizatorului catre calculator si este mai usor astfel decat daca ar fi sa se incerce sa se convinga ca o turma de programe nu se inteleg unul pe altul.

Starea in care toti daemonii sunt porniti este numita de nivel 2, run level 2. Acum trebuie sa se pregateasca de lucrul cu utilizatorul.In aceasta faza, va porni promptul de login, prin executarea de catre init a unei copii a programului getty care urmareste consola.Totodata, Init porneste mai multe copii ale programului getty pentru a monitoriza porturile de Dial-in. Dupa ce s-a initializat si acest program, trecem la run level 3, si utilizatorul poate rula programele. [11]

Bibliografie

[1] – Wikipedia free on-line encyclopedia – *www.wikipedia.com*

[2] – Linux Kernel Website - *www.kernel.org*

[3] – InformIT Website - *http://www.informit.com/articles/article.aspx?p=370047&rl=1*

[4] – S. Sinchack – *Hacking Windows XP* – e-book

[5] – Linux Magazin on-line – *www.linux-magazin.ro*

[6] – E. Raymond – *The Art of Unix Programming* – Addison-Wesley, 2003

[7] – Vicki Stanfield, R. Smith – *Administrarea Sistemului Linux* – Teora 2004

[8] – A. Tanenbaum – *Modern Operating Systems* – e-book

[9] – R. Marsanu – *Cursuri în format digital* – e-book

[10] – Microsoft Developer Network – *www.msdn.com*

[11] – T. Ionescu, Florentina Mocrienco – Structura si Functionarea Sistemelor de Calcul – e-book