TEMĂ DE CASĂ SISTEME DE OPERARE
PROCESE ŞI FIRE DE EXECUŢIE

PROFESOR ÎNDRUMĂTOR: conf.dr.ing. Ştefan Stăncescu

CUPRINS:

A. INTRODUCERE GENERALĂ

Tema noastră poartă numele de Procese şi Fire de execuţie şi putem spune că este una dintre cele mai importante subiecte din cadrul unui sistem de operare. Vom încerca să acoperim cât mai bine şi mai structurat acest subiect punându-ne problema mai întâi a unor concepte fundamentale ale acestei teme. Astfel în capitolul întâi, numit concepte fundamentale, vom prezenta utilizarea, gestionarea proceselor şi comunicare între procese din punct de vedere teoretic şi fără a intra în detalii. În a doua parte a capitolului 1 apare conceptul de fir de execuţie pe care cu ajutorul unor definiţii pur teoretice şi a unor exemple îl vom studia în vederea continuării cu capitolul 2 care deja trece la gestiunea proceselor şi firelor de execuţie precum şi la o problemă foarte importantă a OS-urilor şi anume problema interacţiunii între procese. Ne propunem să prezentăm funcţionalitatea proceselor şi a firelor de execuţie pentru a caracteriza cât mai corect o gestiune de procese. Problemele comunicării interprocese o vom prezenta în forma excluderii reciproce cu aşteptare şi fără aşteptare trecând prin metode esenţiale pentru acest domeniu. În capitolele 3 şi 4 ne propunem să trecem de latura teoretică a temei şi să atingem o latură mai practică a problemei, adică implementarea proceselor şi firelor de execuţie şi apeluri de sistem de gestionare a proceselor comparând două sisteme de operare de succes: Linux şi Windows. Subiectul 5 atinge credem cea

mai importantă problemă a acestei teme şi anume planificarea de procese pentru că aceasta permite exploatarea simultană a resurselor sistemului de calcul ţinând cont şi de timpul de prelucrare. Astfel ne propunem să prezentăm principiile de proiectare a politicilor de planificare şi a planificării în Unix, Linux şi Windows.

În capitolul 6 ne propunem să prezentăm emularea programelor MS-DOS pe Windows NT şi Windows 9x prin acoperirea ambelor tipuri de emulări. În ultimul capitol pentru o comparaţie între Linux şi Windows vom prezenta ambele procese de pornire a sistemului de operare cu tot ce implică acesta

B. SUBIECTE

1.CONCEPTE FUNDAMENTALE (*Vasile Andreea, 443 A*)

1.1. PROCESE

Toate programele in executie la un moment dat ale sistemului de operare poarta numele de procese. Acestea contin seturi de instructiuni ce se desfasoara secvential (una dupa cealalta) care interactioneaza cu alte secvente pentru a folosi resursele comune ale sistemului de operare (procesorul, memoria sau hard-disk-ul). Procesele sunt formate dintr-un program (un sir de instructiuni care trebuie executat de catre calculator) in executie, o zona de date, o stiva si un PC (program counter).
La un moment dat, procesorul nu poate executa decat un singur program, asa ca sarcina de a rula mai multe programe revine sistemului de operare. Acesta introduce un model prin intermediul caruia executia programelor, privita din perspectiva utilizatorului, se desfasoara in paralel. Practic se formeaza un sistem de **pseudoparalelism**. **[TANENBAUM]**

In acest sistem de pseudoparalelism procesorul este pus la dispozitia programelor pe rand intr-o periodata de timp definita pentru fiecare program in parte. Practic ideea este urmatoarea:

Avem spre exemplu 5 procese care ruleaza. Initial avem un singur numarator al programelor (program counter – pc) care realizeaza interschimbarea proceselor. Acestea sunt accesate secvential: se intra in primul proces, se executa codul, se trece la urmatorul s.a.m.d.

Daca am avea 5 numaratoare de program, situatia ar fi alta: s-ar executa toate programele simultan (in paralel). Astfel s-a introdus conceptul de multiprogramare: sistemul de operare executa pe rand, pentru o durata de timp, fiecare dintre cele 5 procese. Totusi, multiprogramarea nu este un concept foarte bine definit in ceea ce priveste ordinea de executie intre doua procese, durata de executie a unui proces sau durata de executie a unei secvente de intructiuni a unui proces.
Concluzia pe care trebuie sa o tragem este ca orice proces este executat secvential, insa sistemul de operare face posibil ca mai multe procese sa fie rulate in paralel (intre ele), distribuind pe rand procesorul cate unui proces. Desi la un moment dat se executa un singur proces, pot fi executate portiuni din mai multe procese, iar un proces se poate gasi in mai multe stari. **[TANENBAUM], [YOLINUX]**

1.1.1. STĂRILE PROCESELOR

1. *Running* (In executie)
2. *Ready* (Pregatit pentru executie)
3. *Waiting/Blocked* (asteapta/este blocat)

- Un proces este in executie atunci cand instructiunile sale sunt executate de catre procesor.

- Procesul este pregatit pentru executie daca, cu toate ca instructiunile sale sunt gata ptr a fi executate, este lasat intr-o coada de asteptare din cauza ca un alt proces este in executie la momentul respectiv de timp.
- Un proces poate fi blocat deoarece in setul sau de instructiuni exista instructiunea de suspendare a executiei sau pentru ca efectueaza o operatie in afara procesorului (adresare memorie, etc) care este foarte consumatoare de timp.

Procesele care se gasesc in Starile Ready si Blocked sunt introduse in cozi de procese: procesele Ready sunt introduse in coada Ready, procesele Blocked sunt introduse in coada Blocked care sunt cozi de Intrare-Iesire.

Procesele isi pot schimba starea din Ready in Running, din Running in Blocked sau din Running in Ready, iar executia lor poate fi planificata.

Sunt practic 4 feluri in care un proces poate fi creat **[MSDN]** :

- prin initializarea sistemului
- prin executia unei secvente de instructiuni care creaza un proces din alt proces
- prin solicitarea unui user de a crea un anumit proces
- prin initializarea unui batch.

1.1.2. UTILIZAREA PROCESELOR

In sistemul de operare Unix orice proces trebuie creat de catre un alt proces. Procesul creator este numit proces parinte, iar procesul creat proces fiu, cu o singura exceptie: procesul *init* care este procesul creat la pornirea sistemului de operare.

Procesele parinte creaza procese fiu care sunt copii fidele ale lor. Procesul copil va avea propria lui zona de date, stiva, set de instructiuni, toate identice cu ale procesului parinte.

Trebuie musai facuta distinctia in cazul acesta intre procesul parinte si procesul copil.

In Linux procesele sunt organizate intr-o ierarhie, avand la baza procesul init si care lanseaza la pornirea sistemului de operare celelalte procese. **[WIKIPEDIA]**

1.1.3. UNIX

Fiecare proces are un identificator numeric, *identificatorul de proces (process identifier – PID)*. Acest identificator este folosit atunci cand se face referire la procesul respectiv.

Un proces este alcatuit din mai multe elemente, din care mentionam: un spatiu de adresa propriu (o zona de memorie in care se afla codul programului), datele si stiva; o lista a descriptorilor de fisiere deschise, o valoare care il identifica in mod unic. Acest lucru se poate face folosind functia *fork()*. Ea returneaza:

- eroare (-1)
- 0 – este proces fiu
- Identificatorul de proces (pid) in cadrul procesului parinte.

Referirea la functa *fork()* s-ar putea face astfel **[EVANJONES], [TANENBAUM], [YOLINUX]** :

```
if ((pid = fork()) <0)
{
        Console.writeline(”eroare”);
        return 1;
}
if (pid ==0)
{
        Console.writeline(”codul procesului copil”
        return 0;
}
If (pid ==1)
{
        Console.writeline(”codul procesului parinte”)
        return pid;
}
```

wait(status)

Prin intermediul functiei *fork()* de creare a proceselor ambele procese create (atat procesul tata, cat si procesul copil) vor efectua in continuare aceleasi instructiuni. In urma instructiunii *switch*, ele vor continua sa execute cod in paralel. Programul (aplicatia) se va termina in momentul in care ambele procese si-au terminat executia prin apelul functiei *exit()*.

In practica, apelul functiei *fork()* este insotit de apelul functiei *exec()*. Aceasta functie (*exec*) are ca efect inlocuirea imaginii procesului nou creat cu un nou program pe care dorim sa-l executam.

1.1.4. GESTIONAREA PROCESELOR

Sistemul de operare Unix are cateva comenzi foarte utile care se refera la procese **[YOLINUX]** :

- **ps** - afiseaza informatii despre procesele care ruleaza in mod curent pe sistem
- **kill -*semnal proces*** - trimite un semnal unui process.
- **killall -*semnal nume*** - trimite semnal catre toate procesele cu numele *nume*

1.1.5. COMUNICAREA ÎNTRE PROCESE

1.1.5.1. PIPE-URI

Comunicarea intre procese se poate face folosind pipe-uri (conducte). „Conducta" este o cale de legatura care poate fi stabilita intre doua procese inrudite. Ea, bineinteles, are doua capete: unul in care se pot scrie date si altul prin care datele pot fi citite.

Pipe-urile permit o comunicare unidirectionala. Sistemul de operare permite conectarea a unuia sau mai multor procese la fiecare din capetele unui pipe, deci este posibil sa existe mai multe procese care scriu, respectiv mai multe procese care citesc din pipe. Astfel se formeaza comunicarea intre procesele care scriu si procesele care citesc din pipe-uri.

1.1.5.2. SEMNALE

Comunicarea intre procese se poate realiza si folosind semnalele (o exprimare a evenimentelor care apar asincron in sistem). Un proces este capabil de a genera sau a primi semnale. In cazul in care un proces primeste un semnal, el poate alege sa reactioneze la semnalul receptionat intr-unul dintre urmatoarele moduri:

- sa capteze semnalul si sa-l trateze (*signal handler*)
- sa ignore semnalul respectiv
- sa execute actiunea implicita la primirea unui semnal.

Semnalele pot fi de mai multe tipuri, care corespund in general unor actiuni specifice. Fiecare semnal are asociat un numar, iar acestor numere le corespund unele constante simbolice definite in bibliotecile sistemului de operare. Standardul POSIX.1 defineste cateva semnale care trebuie sa existe in orice sistem UNIX. **[TANENBAUM]**

1.2. FIRE DE EXECUŢIE

Executia planificata a proceselor presupune ca la un moment dat procesorul sa fie alocat procesului care trebuie sa se execute, deci trebuie scos din procesul in care se afla initial la momentul respectiv de timp. Aceasta comutare intre procese (*process switching*) este o operatie consumatoare de timp deoarece trebuie salvati registrii care apartin proceselor.

Un concept des folosit in programare si in management-ul sistemelor de operare este f*irul de executie (thread-ul)* in interiorul unui proces. Firele de executie sunt numite *procese usoare* (lightweight processes), sugerandu-se asemanarea lor cu procesele.

Un fir de executie este un flux de instructiuni care se executa in interiorul unui proces. Un proces poate sa fie format din mai multe thread-uri care se executa in paralel, avand in comun toate caracteristicile procesului. In interiorul procesului thred-urile ruleaza in paralel, impart zona de date si executa portiuni distincte din acelasi cod. Variabilele procesului sunt practic globale, se vad in toate thread-urile, orice modificare a unei variabile a procesului dintr-un thread este practic o modificare a variabilei procesului, si se poate vedea si in celelalte thread-uri care ruleaza din interiorul procesului.

La nivelul sistemului de operare, executia in paralel a thread-urilor este realizata in mod asemanator cu cea a proceselor. Se obtine o comutare intre thread-uri, conform unui algoritm de prelucrare si sincronizare. Comutarea este mult mai rapida deoarece informatiile caracteristice fiecarui thread sunt mult mai putine decat in cazul proceselor, deoarece firele de executie nu au sfoarte putine resurse proprii.

Putem percepe un fir de executie ca un PC (program counter), o stiva si un set de registri. Toate celelalte resurse sunt resursele proprii ale proceselor care sunt puse la dispozitia thread-ului. Practic putem vedea firul de executie ca un program in executie fara spatiu de adresa propriu.

Daca spre exemplu avem un program in care trebuie sa citim informatiile dintr-o baza de date, sa le accesam si pentru fiecare informatie gasita in baza respectiva sa executam un cod de instructiuni, lucrul acesta s-ar putea realiza in 2 metode:

1. executand instructiunile secvential – pentru fiecare rand citim din baza de date executam instructiunile, trecem la urmatorul rand s.a.m.d
2. pornim cate un thread pentru fiecare rezultat in paralel (acelasi timp) si compunem un algoritm de planificare la sfarsit care sa realizeze returnarea valorilor date de instructiuni pe rand (una cate una) pentru fiecare rand returnat din baza de date. **[TANENBAUM] , [MSDN], [MSDN FORUMS]**

1.2.1. CREAREA FIRELOR DE EXECUŢIE

In cazul sistemului de operare Linux, biblioteca standardul de management si sincronizare a firelor de executie este libraria *pthreads.* In cazul sistemului de operare Windows - Visual Studio - libraria este *System.Threads.*

Pentru a crea un thread in Linux avem la dispozitie functia *pthread_create*, cu urmatorul apel:

*pthread_create (pthread_t * thread, pthread_attr_t* attr, void* (*start_routine) (void*), void *arg);*

unde:

- thread = pointer care contine informatii despre structura
- attr = pointer care specifica atributele firului nou creat
- start_routine = pointer catre functia ce va fi executata (metoda go() din windows)
- arg = pointer catre argumentele asteptate de start_routine;

Tot in cadrul sistemului Linux se pot implementa firele de executie prin intermediul functiei *clone()*. Aceasta este o interfata alternativa la functia *fork()* (cea care crea procesul copil folosind procesul tata). **[YOLINUX].** Terminarea unui thread se face in Linux apeland functia *pthread_exit()* ori prin iesirea din functia *start_routine()*.

1.2.2. TERMINAREA FIRELOR DE EXECUŢIE

Un thread in Linux se poate termina apeland functia *void pthread_exit(void *retval)*. Valoarea retval este valoarea pe care thread-ul o returneaza la terminare. Practic, cum in programare totul este posibil, valoarea returnata de un thread poate fi introdusa in functia de start a unui alt thread din cadrul aceluiasi proces.

Insa, nu pentru toate thread-urile poate fi preluata valoarea de iesire. Thread-urile se impart in 2 categorii:

- **joinable** – thread-uril ale caror stari/valori de iesire pot fi preluate de catre alte procese/alte fire de executie
- **detached** – thread-uri ale caror stari/valori de iesire nu pot fi preluate.

In cazul thread-urilor **joinable**, in momentul preluarii acestora, resursele lor nu sunt complet dezactivate, asteptandu-se un eventual join pentru ele. Thread-urile detached dezactiveaza complet resursele proprii in momentul in care se iese din ele. **[TANENBAUM], [YOLINUX]**

1.2.3. SINCRONIZAREA FIRELOR DE EXECUŢIE

Pentru a evita ca la iesirea din functia go() [in cazul windows] sau pthread_exit (linux) toate thread-urile sa-si termine executia setului de instructiuni in acelasi timp, trebuie implementat un algoritm de sincronizare si planificare.

Problema sincronizarii nu se pune numai la thread-uri, ci si la procese. In cazul in care mai multe procese/fire de executie folosesc resurse comune, rezultatul final al executiei lor poate sa nu fie foarte stabil deoarece conteaza ordinea in care procesele /firele de executie returneaza rezultatele executarii seturilor de instructiuni.

Definitie: Situatiile in care rezultatul executiei unui sistem format din mai multe procese (fire de executie) depinde de viteza lor relativa de executie se numesc conditii de cursa (in engleza: race conditions).

Conditiile de cursa apar atunci cand trebuiesc executate/modificate parti din program care sunt puse la comun (sunt folosite si de alte procese/thread-uri). Aceste portiuni care acceseaza parti din program puse la comun se numesc *zone (sectiuni critice – critical sections).* Daca ne asiguram ca thread-urile care ruleaza nu executa cod in acelasi timp in zonele critice, problema sincronizarii este rezolvata. Acest procedeu poarta denumirea de *excluziune mutuala.*

O alta metoda de sincronizare este ***metoda Semafoarelor***. Semaforul este un tip de date abstract ce poate avea o valoare intreaga nenegativa si asupra caruia se pot efectua operatiile: *init*, *up* si *down*. Init – initializeaza semaforul, down – realizeaza o operatie de decrementare, daca valoarea este pozitiva. Daca valoarea este nula, procesul se blocheaza. Operatia *up* incrementeaza valoarea semaforului, daca nu exista un process blocat pe acel semafor. Daca exista, valoarea semaforului ramane zero iar unul dintre procese care este blocat va fi deblocat.

Ca si metode de sincronizare mai avem ***mecanisme System sau VIPC***. Semafoarele si memoria partajata fac parte din mecanismele de comunicare si sincronizare a proceselor cunoscute ca System VIPC. Prin mecanismul de memorie partajata doua sau mai multe procese pot pune in comun o zona de memorie. Fiind o resursa pusa la comun, accesul la zona de memorie partajata trebuie sa fie sincronizat.

[WIKIPEDIA], [EVANJONES]

1.3. CONCLUZII

- Un proces este vazut ca fiind format dintr-o zona de cod, o zona de date, stiva si registri ai procesorului.
- Fiecare proces este o multime distincta, independent de celelalte procese aflate in executie la un moment dat.
- Procesorul poate rula la un moment dat un singur proces, deci procesele sunt executate pe rand, dupa un algoritm de planificare
- La nivelul aplicatiei acestea par ca sunt executate in paralel.
- Procesele trebuie planificate in executie, astfel incat ele sa ruleze aparent in paralel.
- un proces, imediat ce a fost creat, este format dintr-un singur fir de executie, numit fir de executie principal (initial).
- Toate firele de executie din cadrul unui proces se vor executa in paralel.
- Firele de executie folosesc toate variabilele globale ale proceselor.
- Cu toate ca procesele fi thread-urile sunt concepte diferite, in Linux ele sunt implementate la nivelul sistemului de operare prin apeluri catre aceeasi functie – clone().
- Thread-urile au aparut ca o solutie pentru imbunatatirea performantei.
- In cadrul aplicatiilor, procesele incurajeaza modularitatea codului, ceea ce duce la programe mai flexibile si mai robuste pe termen lung.
- Procesele imbunatatesc secutitatea aplicatiei deoarece diferite parti sunt isolate intre ele.
- Firele de executie pot fi o solutie mai buna atunci cand trebuie accesate frecvent zone de date comune.

In continuare vom da 2 exemple de thread-uri/procese implementate in Visual Studio 2005:

Sa presupunem ca avem o consola in care vrem sa rulam bat-ul test:

```
static void Main(string[] args)
      {
                  System.Diagnostics.Process proc; // declaram un nou proces
            proc = System.Diagnostics.Process.Start("C:\\test.bat"); // rulam test.bat
                  proc.WaitForExit();      //      Asteptam      ca      procesul      sa      se      termine.
      }
```

Mai jos avem o consola care creeaza procese:

```
static void Main(string[] args)
{
      Console.WriteLine("Hello, I'm creating a process");
      System.Diagnostics.Process proc; // Declaram noul proces
      System.Diagnostics.ProcessStartInfo procInfo = new
      System.Diagnostics.ProcessStartInfo(); // declaram informatia de start pentru procesul respectiv
      procInfo.UseShellExecute = true;  //daca conditia este falsa, putem rula numai executabile
      procInfo.WorkingDirectory = "C:"; //avem drive-ul C
      procInfo.FileName = "notepad.exe"; // programul sau comanda care trebuie executata
      procInfo.Arguments = "C:\\boot.ini"; //argumentele comenzii
      procInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Maximized; // Vom rula notepad-
      ul maximizat, totodata mai putem avea 2 versiuni: (ProcessWindowStyle.Hidden) sau
      (ProcessWindowStyle.Minimized)

            Console.WriteLine("Notepad Process Started at : {0}", DateTime.Now.ToString());
            proc = System.Diagnostics.Process.Start(procInfo); //  ca si cum am testa "notepad.exe
      C:\boot.ini" din windows Start->Run.
            proc.WaitForExit(); // asteapta ca procesul sa se termine (sa fie inchis de user in cazul
      nostrum)
            Console.WriteLine("Notepad Process Closed at : {0}", DateTime.Now.ToString());
            Console.WriteLine("\n\nPress Enter To Continue.");
            Console.ReadLine();

            if(!proc.HasExited) // pentru a ne asigura ca procesul a fost inchis
            {
               proc.Kill();}
            }
         }
      }
```

Un alt exemplu este pornirea thread-urilor. Codul de mai jos este o clasa care este folosita pentru a scoate informatiile de sistem ale unor servere. Din clasa respectiva vom evidentia numai ceea ce prezinta obiectul interesului nostru.

```
using System;
using System.Threading;

public class Threads
{
   private Integer finishedThreads;
   private static Mutex mut = new Mutex();

   public Threads()
   {
   }
   public void setFinishedThreads(Integer finish)
   {
      this.finishedThreads = finish;
   }
   public void go()
```

```
    {
        ManagementPath path1 = new ManagementPath(String.Format("\\\\{0}\\root\\cimv2", host));
        ManagementScope ms1 = new ManagementScope(path1, connectionTest);
        DataRow row1 = table1.NewRow();
        try
        {
            ms1.Connect();
            row1["Nume Server"] = getSystemInformation(host, connectionTest) + "!myCustomBR!" +
getIpAdress(host, connectionTest);
            string carry = "";
            ArrayList hardUsage = getHardUsage(host, connectionTest);
            hardUsage.Reverse();
            foreach (string entry in hardUsage)
            {
                carry = entry + "; " + "!myCustomBR!" + carry;
            }
            row1["Disk Usage[Mb]"] = carry;
            carry = "";
            foreach (string entry in getcpuUsage(host, connectionTest))
            {
                carry = entry + "; " + carry;
            }
            row1["CPU Usage[%]"] = carry;
            carry = "";
            foreach (string entry in getMemUsage(host, connectionTest))
            {
                carry = entry + ";" + "!myCustomBR!" + carry;
            }
            row1["Memory Usage[Kb]"] = carry;
            carry = "";
            foreach (string entry in getNetworkUtilization(host, connectionTest))
            {
                carry = entry;
            }
            row1["Network Usage[%]"] = carry;
            carry = "";
            row1["Data/Ora"] = System.DateTime.Now;
            mut.WaitOne();
            table1.Rows.Add(row1);
            finishedThreads.increment();
            mut.ReleaseMutex();
        }
        catch (Exception ex)
        {
        }
    }
    public void startThisThread()
    {
        Thread mythread = new Thread(go);
        mythread.Start();
    }
}
```

In pagina in care avem nevoie de clasa vom proceda in felul urmator:
Vom crea un obiect al clasei:

```
Threads myTh = null;
myTh = new Threads();
myTh.setTable(table1);
```

setam toate valorile de care avem nevoie si pornim thread-ul:

```
myTh.startThisThread();
while (startedThreads > finishedThreads.getIndex())
                {
                   Thread.Sleep(500);
                }
```

Conditia pentru a nu exista suprapuneri.

2. GESTIUNEA PROCESELOR ŞI FIRELOR DE EXECUŢIE. COMUNICAREA INTERPROCESE (ENOIU EDUARD PAUL, 443A)

2.1.GESTIUNEA PROCESELOR ŞI APELURILE DE SISTEM DE GESTIUNE A PROCESELOR

Procesul este o entitate independentă cu propriul său numărător de program şi astfel are o anumită stare internă dar există nevoia interacţionării între procese. Folosind acest model de proces putem să ne punem problema ce se întâmplă în interiorul sistemului. Dacă de exemplu apare o întrerupere legată de disc, sistemul de operare ia decizia de întrerupere a procesului curent şi va executa procesul dedicat gestiunii discului care aşteaptă această întrerupere. Astfel în ansamblu în loc să gândim totul în întreruperi putem să ne imaginăm procese diferite care se blochează atunci când aşteaptă să se întâmple ceva şi se deblochează pentru a fi gata de execuţie. Acest punct de vedere din **[TANENBAUM]** conturează un nivel inferior sistemului de operare şi reprezentat de blocul planificator (acest modul este discutat în paragraful 2.2.2. FUNCŢIONALITATEA FIRELOR DE EXECUŢIE şi în capitolul Planificarea proceselor) care este structurat pe procese şi tratează întreruperile şi planificarea în timp. Acest nivel inferior al unui sistem de operare poate avea forma din Figura 2.1. Pentru a implementa modelul de proces, sistemul de operare

foloseşte un aşa numit BLOC DE CONTROL AL PROCESULUI pe care îl vom discuta în paragraful următor.

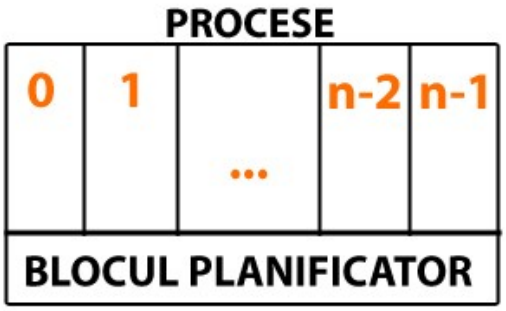


Figura 2.1. Blocul planificator şi procesele secvenţiale.

Mai multe detalii despre planificarea proceselor găsiţi în capitolul de Planificare a proceselor.

2.1.1. BLOCUL DE CONTROL AL PROCESULUI

Această noţiune de bloc de control al procesului este numită aici **[TANENBAUM]** şi aici **[TANENBAUM87]** tabelă de procese sau *process table*. O să păstrăm notaţia de bloc de control al procesului pentru că defineşte mai bine gestiunea şi implementarea modelului de proces. Blocul de control al procesului are o intrare pentru fiecare proces. Această intrare conţine informaţii despre starea procesului, numărătorul program, indicatorul de stivă, alocarea memoriei, starea fişierelor deschise, detalii de planificare şi orice alte informaţii utile despre un proces care trebuiesc salvate atunci când procesul comută între stările *activ*, *blocat*, *gata* şi *terminat,* astfel încât să poată să fie repornit din aceeaşi parametrii rămaşi.

Întrările din blocul de control al procesului pot avea forme diferite de la un sistem de operare la altul dar pentru generalitate considerăm următoarea formă a unor câmpuri ale unei intrări obişnuite din blocul de control al procesului din Tabelul 2.1.1:

GESTIUNEA PROCESULUI	GESTIUNEA MEMORIEI	GESTIUNEA FIŞIERELOR
Registre Numărătorul program Cuvântul de stare al programului Indicatorul de stivă Starea procesului Prioritate Parametrii de planificare Identificatorul procesului Proces primar Grupul procesului Semne Momentul începerii procesului Durata de utilizare UCP	Indicator spre segmentul de text Indicatorul spre segmentul de date Indicatorul spre segmentul de stivă	Directorul rădăcină Directorul de lucru Descriptorii de fişiere Identificatorul utilizatorului Identificatorul grupului

Tabelul 2.1.1 Câmpurile unei intrări obişnuite din tabela de procese.

În tabelul 2.1.1 putem observa câmpurile din prima coloană care se referă la gestiunea procesului. Celelalte două coloane se referă la gestiunea memoriei şi respectiv a fişierelor.

2.1.2 APELURILE DE SISTEM DE GESTIUNE A PROCESELOR ÎN LINUX ŞI WINDOWS

Ne propunem să tratăm problema gestiunii de procese din perspectiva implementării lor în practică în sistemele de operare Windows şi Linux. Astfel vom trata pe rând cele două implementări astfel încât să putem compara sistemul de gestiune a proceselor din punct de vedere al metodelor şi apelurilor de sistem folosite de fiecare sistem în parte.

Dacă un program se execută în sistemul de operare pentru alocarea resurselor necesare rulării programului se va crea un proces. Fiecare sistem de operare pune la dispoziţie apeluri de sistem pentru crearea unui proces, terminarea unui proces, aşteptarea terminării unui proces dar şi apeluri pentru duplicarea descriptorilor de resurse între procese ori închiderea acestor descriptori. Un proces are

posibilitatea să verifice sau să seteze variabilele mediului în care acesta rulează prin mai multe apeluri de bibliotecă.

2.1.2.1 APELURILE DE SISTEM PENTRU PROCESE ÎN LINUX

Apelurile de sistem puse la dispoziţie de Linux pentru gestionarea proceselor sunt: fork şi exec pentru crearea unui proces şi respectiv modificarea imaginii unui proces, wait şi waitpid pentru aşteptarea terminării unui proces şi exit pentru terminarea unui proces. Din **[YOLINUX]** ştim că pentru copierea descriptorilor de fişier Linux pune la dispoziţie apelurile de sistem dup şi dup2. Pentru citirea, modificarea ori ştergerea unei variabile de mediu, biblioteca standard de C pune la dispoziţie apelurile getenv, setenv, unsetenv precum şi un pointer la tabela de variabile de mediu environ.

Fiecare proces are un spaţiu de adrese de 4 GB din care 3 GB sunt disponibili pentru alocare procesului, iar în celălalt 1 GB rămas pot fi adresate structurile şi simbolurile sistemului de operare, bineînţeles în mod protejat. Aşadar fiecare proces “vede” sistemul de operare în spaţiul său de adrese însă nu poate accesa zona respectivă decât prin intermediul apelurilor de sistem **[BOVET]** (comutând procesorul în modul de lucru privilegiat).

În LINUX singura modalitate de creare a unui proces este dată de apelul de sistem **fork**. Efectul acestui apel de sistem este crearea unui nou proces, copie a procesului care a apelat fork. Diferă doar PID-ul proceselor, noul proces primind un nou PID de la sistemul de operare. Secvenţa clasică de creare a unui proces este prezentată în continuare:

```
#include <sys/types.h>
#include <unistd.h>
…
switch (pid = fork()) {
      case -1:
            printf(“fork eşuat\n”);
            exit(-1);
      case 0:
            …
      default:
            printf(“se creează un process cu pid %d\n”, pid);
            …
}
```

Mai multe detalii şi exemple găsiţi capitolul dedicat acestei probleme numit Apelurile de sistem de gestiune a proceselor in Linux.

2.1.2.2 APELURILE DE SISTEM PENTRU PROCESE ÎN WINDOWS

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

Conform **[CRK] [MSDN]** fiecare proces are un spaţiu de adrese de 4 GB din care 2 GB sunt disponibili pentru alocare aplicaţiei iar în ceilalţi 2 GB pot fi adresate structurile şi simbolii sistemului de operare, bineînţeles în mod protejat. Opţional se pot aloca 3GB proceselor utilizator şi doar 1 GB pentru

sistemul de operare. Aşadar fiecare proces “vede” sistemul de operare în spaţiul său de adrese însă nu poate accesa zona respectivă decât prin intermediul apelurilor de sistem. Procesele noi sunt create folosind funcţia Win32 API CreateProcess. Faţă de apelul de sistem fork care nu conţine nici un parametru şi exec care are numai trei: referinţe către numele fişierului care trebuie executat, vectorul cu parametrii din linia de comandă (analizat sintactic) şi şirurile de mediu, această funcţie are 10 parametrii. Astfel putem trasa o comparaţie între cele două sisteme de operare spunând că funcţia CreateProcess este mult mai complicată din punct de vedere constructiv. Vorbind în mare, cei 10 parametrii din CreateProcess conform **[TANENBAUM]** sunt:

1. O referinţă către numele fişierului executabil.
2. Linia de comandă în sine (neanalizată sintactic).
3. O referinţă către un descriptor de securitate pentru proces.
4. O referinţă către un descriptor de securitate pentru firul de execuţie iniţial.
5. Un bit ce specifică dacă noul proces moşteneşte referinţele creatorului.
6. Diferiţi indicatori (ex. Modul de eroare).
7. O referinţă către şirurile de mediu.
8. O referinţă către numele catalogului de lucru curent al noului proces.
9. O referinţă către o structură ce descrie fereastra iniţială pe ecran.
10. O referinţă către o structură care întoarce 18 valori către apelant.

Din cauza acestor parametrii sistemul de operare Windows NT nu impune nici un fel de legătură copil-părinte ca la Linux şi astfel toate procesele sunt create egale. Numărul de apeluri Win32 API care lucrează cu procese, fire de execuţie şi fibre este de aproape 100.

2.2. GESTIUNEA FIRELOR DE EXECUŢIE

Fiecare proces are un spaţiu de adrese şi un singur fir de control în sistemele de operare tradiţionale dar conform **[TANENBAUM]** multe sunt situaţiile în care este de dorit existenţa mai multor fire de control în acelaşi spaţiu de adrese rulând intr-un mod pseudo-paralel. Firul de execuţie adaugă procesului posibilitatea execuţiilor independente şi multiple în cadrul mediului aceluiaşi proces. Ştiind că existenţa mai multor fire de execuţie rulând în paralel seamănă cu procesele rulând în paralel pe acelaşi calculator putem spune că firele de execuţie partajează spaţiul de adrese, fişierele deschise iar procesele partajează memoria fizică, discurile şi alte resurse. Putem exemplifica aceste concepte prin figura 2.2. în care observăm că fiecare proces are propriul spaţiu de adrese şi un singur fir de control.

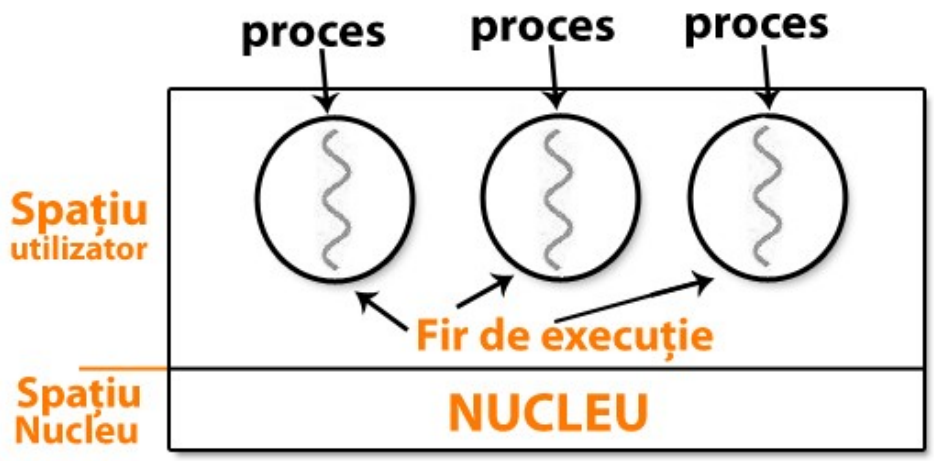


Figura 2.2. Exemplu de trei procese cu un singur fir de execuţie.

Există şi cazul în care există un singur proces cu mai multe fire de execuţie acestea lucrând în acelaşi spaţiu de adrese. Pentru mai multe detalii despre conceptele generale ale firelor de execuţie puteţi consulta capitolul Concepte fundamentale.

2.2.1. UTILIZAREA FIRELOR DE EXECUŢIE

Principalul motiv pentru folosirea firelor este că în multe aplicaţii se desfăşoară mai multe activităţi în acelaşi timp şi astfel modelul de programare devine mai simplu, acest motiv rămânând la fel ca la procese.

Faţă de exemplele din **[TANENBAUM87]** şi **[RUSSINOVICH]** pentru utilitatea şi utilizarea firelor de execuţie putem considera exemplul unui server WEB pentru un site Word Wide Web. În acest exemplu se aşteaptă cereri pentru pagini şi apoi pagina cerută este transmisă înapoi clientului. O metodă de organizare a serverului web bazat pe fire de execuţie multiple este prezentată în **[TANENBAUM]** şi arată ca în Figura 2.2.1. În acest caz, un fir de execuţie numit dispecer, citeşte cererile de lucru de la reţea. După ce această cerere de lucru este examinată, dispecerul alege un fir de execuţie lucrător liber şi îi transmite cererea: Dispecerul trezeşte apoi lucrătorul inactiv, mutându-l din starea blocat în starea gata de execuţie. Când lucrătorul se activează, verifică cererea şi spune dacă poate fi satisfăcută din cache-ul paginilor intens folosite la care are au acces toate firele de execuţie. În caz contrar, porneşte o nouă operaţie de citire pentru a obţine pagina de pe disc şi se blochează până când se termină această operaţie. Când acest fir s-a blocat se alege un alt fir gata de execuţie.

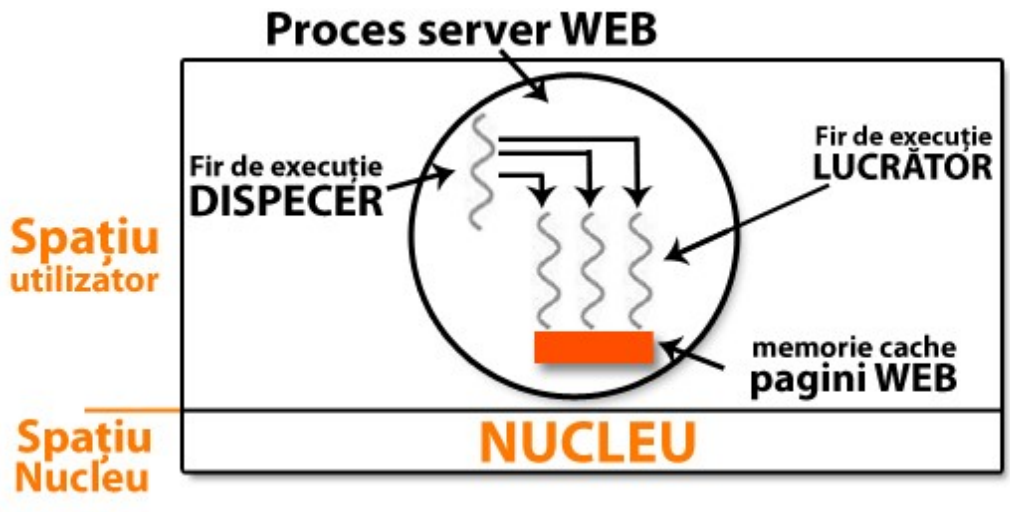


Figura 2.2.1. Exemplu de server web construit cu fire de execuţie multiple.

Acest implementare face ca serverul să fie scris ca o colecţie de fire de execuţie secvenţiale. Dacă am gândi serverul fără fire de execuţie am avea două posibilităţi: o metodă de operare a unui server cu un singur fir ca în **[TANENBAUM]** şi rezulatul evident este că nu se pot procesa mai multe cereri simultan; a doua metodă foloseşte o arhitectură de automat finit ca în **[TANENBAUM]**. Ambele metode nu sunt utile din cauză că metoda cu fire de execuţie multiple este singura care asigură secvenţialitatea proceselor ajungându-se la parallelism. Această metodă ne dă o imagine a utilizării funcţionale a firelor de execuţie. Pentru mai multe detalii despre utilizarea firelor de execuţie consultaţi capitolul Concepte fundamentale.

2.2.2. FUNCŢIONALITATEA FIRELOR DE EXECUŢIE

Funcţionalitatea firelor de execuţie este dată în primul rând de implementarea firelor de execuţie. În acest paragraf vom analiza funcţionalitatea firelor de execuţie din perspectiva metodelor generale de implementare. O analiză a implementării firelor de execuţie pentru sistemele de operare Windows şi Linux găsiţi în capitolul Implementarea proceselor si firelor de executie in Linux si in Windows. Pentru o înţelegere cât mai bună am ales împărţirea implementărilor ca în **[TANENBAUM]** în 3 metode: implementarea funcţionalităţii firelor de execuţie în spaţiul utilizator, în nucleu precum şi implementarea hibridă.

IMPLEMENTAREA FIRELOR DE EXECUŢIE ÎN SPAŢIUL UTILIZATOR

Această metodă pune întregul pachet al firelor de execuţie în totalitate în spaţiul utilizator astfel nucleul nu ştie nimic despre ele şi astfel acesta crede că se ocupă de gestiunea unor procese pur şi simplu normale, fiecare proces având un singur fire de execuţie. Asta ne duce cu gândul că singura implementare acestei metode poate fi într-un sistem de operare care nu suportă fire. Trebuie spus că toate sistemele de

operare au făcut odată parte din această categorie. Indiferent de sistemul de operare, gestiunea firelor de execuţie în spaţiul utilizator are următoarea schemă din Figura 2.2.2(a). Firele de execuţie rulează deasupra unei colecţii de proceduri pentru gestiunea firelor numit în **[TANENBAUM]** şi în alte lucrări de specialitate executive. În cazul în care firele de execuţie sunt gestionate în spaţiul utilizator, fiecare proces are nevoie de un bloc de control al firelor de execuţie pentru a se putea urmări firele procesului respectiv. Acest bloc de control al firelor de execuţie este numit în **[TANENBAUM]** tabelă de fire de execuţie şi este similară blocului de control al proceselor conţinând propietăţile firelor (numărătorul program, indicatorul de stivă, registre, etc.). Noi vom folosi denumirea de bloc de control al firelor de execuţie în analogie cu denumirea blocului de control al proceselor.

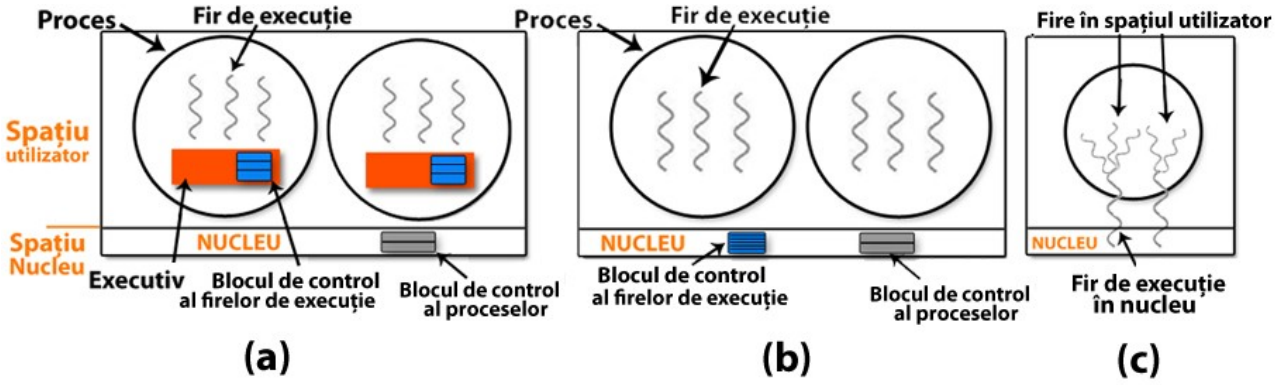


Figura 2.2.2. (a) Fire de execuţie în spaţiul utilizator. (b) Fire de execuţie în nucleu. (c) Implementarea hibridă a firelor de execuţie

Din **[RUSSINOVICH]** şi **[TANENBAUM]** ştim că există o suită de avantaje pentru folosirea funcţionaliţăţii firelor de execuţie în spaţiul utilizator printre care eficacitatea comutării între firele de execuţie, permit fiecărui process să aibă propriul său algoritm de planificare dar prezintă câteva dezavantaje majore**:**

1. implementarea defectuoasă a apelurilor blocante de sistem care dă posibilitatea în acest caz să existe un fir blocat care să afecteze celelalte fire (o alternativă pentru rezolvarea acestei probleme am găsit-o în **[BOVET]** şi **[RUSLING]** anume că există un apel de sistem *select* care permite apelantului să îţi dea seama de o anumită blocare a firelor).
2. problema defectelor de pagină apare atunci când un fir de execuţie cauzează un defect de pagină(dacă un program apelează sau efectuează un salt la o instrucţiune care nu este în memorie apare defectul de pagină) şi nucleul normal nu are de unde să ştie de existenţa firelor şi blochează întregul proces până când pagina căutată şi găsită pe disc, cu toate că s-ar putea executa alte fire.

IMPLEMENTAREA FIRELOR DE EXECUŢIE ÎN NUCLEU

Această metodă ia în calcul că nucleul ştie de existenţa firelor de execuţie şi astfel nucleul le gestionează neexistând un executiv pentru fiecare proces şi nici un bloc de control al firelor de execuţie pentru fiecare proces. Cum le gestionează nucleul? Prin un bloc de control al firelor de execuţie care menţine evidenţa tuturor firelor din sistem **[TANENBAUM]**. Când un fir doreşte crearea unui fir nou sau distrugerea altui fir, efectuează un apel către nucleu care se ocupă apoi de creare sau distrugere prin modificarea blocului de control al firelor de execuţie. Implementarea firelor de execuţie în nucleu poate fi observată în Figura 2.2.2(b).

Problema care exista în cazul firelor de execuţie implementate în spaţiul utilizator cu apelurile blocante aici numai apare pentru că acestea sunt implementate ca apeluri de sistem. Dacă un fir de execuţie cauzează un defect de pagină, nucleul poate verifica uşor dacă procesul are alte fire de execuţie şi în acest caz poate executa unul dintre ele fără să aştepte ca pagina respectivă să fie adusă de pe disc.
Dezavantajul principal al acestei implementări este că un apel de sistem are o durată mai mare astfel că în cazul în care există o aglomerare de operaţii cu fire va apărea o supraîncărcare.

IMPLEMENTĂRI HIBRIDE

În acest paragraf ne vom ocupa de iniţiativele de combinare a firelor din spaţiul utilizator şi firele din nucleu. Această abordare este prezentată în amănunt în **[TANENBAUM].** Au existat destule proiecte de cercetare cau au combinat avantajele firelor implementate în spaţiul utilizator cu cele implementate la nivel de nucleu. Astfel există posibilitatea folosirii unor fire la nivel de nucleu care să fie folosite de unul sau mai multe fire din spaţiul utilizator astfel încât nucleul este conştient de firele din nucleu nu şi de cele din spaţiul utilizator. Legătura dintre firele din spaţiul nucleu şi cele din spaţiul utilizator este o legătură numită în **[TANENBAUM]** legătură multiplexată. Pur şi simplu fiecare fir implementat în nucleu este utilizat pe rand de mai multe fire din spaţiul utilizator. Această metodă poate fi observată structural în Figura 2.2.2(c).

2.3. COMUNICAREA INTERPROCESE

Problema interacţiunii între procese este una din cele mai importante probleme din sistemele de operare moderne. Trebuie astfel să evităm situaţiile în care procesele se afectează în mod nedorit. Pe caz general trebuie ca un ”program executat să se creadă singur” şi astfel să nu existe condiţii conflictuale. Condiţiile conflictuale sunt situaţiile în care procesele folosesc o resursă comună în care este posibilă influenţarea între procese.

Comunicarea interprocese poate aduce unele probleme din punct de vedere al zonei de memorie partajate de mai multe procese ce concurează în acelaşi timp pentru aceeaşi coadă de executare. Exemplele cele mai concludente pentru această problemă se găsesc în **[TANENBAUM]** şi în **[STĂNCESCU]** în legătură cu gestionarul de imprimantă la care două procese vor să acceseze memoria partajată în acelaşi timp. Situaţia din acest exemplu în care două sau mai multe procese citesc sau scriu date partajate şi rezultatul final depinde într-o oarecare măsură de cine se execută şi când, sunt numite aici **[TANENBAUM]** condiţii de cursă. Altfel spus există programe care folosesc aceleaşi resurse doar în acele părţi de program care pot eventual produce conflicte în procese şi care pot conduce la condiţii de cursă. Aceste zone sunt numite zone critice. Pentru remedierea acestor potenţiale necazuri trebuie să existe comunicare între procese şi astfel să existe nişte condiţii de excludere reciprocă între procese. Unele concepte de comunicare interprocese se regăsesc şi în capitolul 1 Concepte fundamentale.

2.3.1. PROBLEMA EXCLUDERII RECIPROCE

Problema excluderii reciproce este o metodă prin care procesele se anunţă reciproc asupra zonelor critice şi cumva se reglementează accesul la resursele critice. Pentru a evita condiţiile de cursă din **[TANENBAUM]** ştim că există patru condiţii de cooperare:

1. Să nu existe două procese simultan în propriile zone critice privind aceeaşi resursă;
2. Procesele să fie determinate (adică să aibă rezultate identice indiferent de starea maşinii);
3. Nici un proces care rulează în afara zonei critice să nu blocheze alte procese (condiţie de eficienţă);
4. Nici un proces să nu aştepte arbitrar de mult până la intrarea în zona critică proprie.

Conform **[STĂNCESCU]** există două categorii de metode de excludere între procese:

1. EXCLUDEREA RECIPROCĂ CU AŞTEPTARE.
2. EXCLUDERE RECIPROCĂ FĂRĂ AŞTEPTARE.

2.3.2. EXCLUDEREA RECIPROCĂ PRIN AŞTEPTARE

Excluderea reciprocă prin aşteptare este o metodă prin care un proces ocupat în propria regiune critică nu va fi deranjat în această regiune de nici un alt proces. O primă metodă numită în **[STĂNCESCU]** Metoda universală este o soluţie de folosire a întreruperilor prin blocarea şi deblocarea lor pentru a jongla într-un mod convenabil cu accesul la memoria partajată. Astfel blocarea sistemului de întreruperi duce la posibilitatea examinării sau actualizării memoriei partajate de către un proces fără grija altui proces care ar putea interveni. Nu este o metodă convenabilă în acest caz al excluderii reciproce din cauză că ar fi periculos ca un proces utilizator să aibă puterea să dezactiveze întreruperile şi să nu le mai reactiveze niciodată.

2.3.2.1. SOLUŢIA CU VARIANTĂ CU POARTĂ

Această soluţie se bazează pe o variabilă partajată cu rol de poartă cu valoare iniţială 0 adică nici un proces nu se află în regiunea critică. Când un proces vrea să intre în regiunea critică verifică poarta. Dacă poarta are valoarea 0, procesul îi setează porţii valoarea pe 1 (valoarea 1 înseamnă că în regiunea

critică se află un proces) şi intră în regiunea critică. Dacă poarta este 1, procesul este nevoit conform acestei metode să aştepte ca poarta să devină 0. Dar apare o problemă atunci când al doilea proces modifică valoarea porţii imediat după ce primul proces a terminat a doua verificare astfel încât ambele procese se vor afla în regiunile critice în acelaşi timp.

2.3.2.2. SOLUŢIA DE ALTERNARE FORŢATĂ CU O VARIABILĂ COMUTATOR COMUNĂ

Presupunem două procese A şi B. Această soluţie este o soluţie care impune celor două procese A şi B să alterneze accesul în regiunile lor critice, de exemplu în înregistrarea fişierelor pentru tipărire. **[TANENBAUM]** Nici unul dintre procese nu va avea voie să zicem să înregistreze consecutiv două fişiere. Astfel spus un proces care nu se află în zona critică poate să blocheze un alt proces. Ştiind de cele patru condiţii de cooperare ne dăm seama că această soluţie încalcă condiţia de eficienţă care spune că nici un proces care rulează în afara zonei critice să nu blocheze alte procese. Deşi acest algoritm evită cursele nu este eficient.

2.3.2.3. VARIANTA PETERSON

Soluţia lui Peterson de excludere reciprocă este o metodă ce nu impune alternarea strictă şi care oferă un mod foarte simplu de obţinere a excluderii dorite. Algoritmul lui Peterson este scris în limbaj C şi are următoarea formă:

```
#Define FALSE 0
#Define TRUE 1
#Define N 2
int randul_procesului;
int proces_interesat[N];

void intrare_regiune(int proces);
{
        int celalalt_proces;
        celalalt_proces=1-proces;
        proces_interesat[proces]=TRUE;
        randul_procesului=proces;
        while(randul_procesului==proces&&proces_interesat[celalalt]==TRUE
}

void plecare_regiune(int proces)
{
proces_interesat[proces]=FALSE;
}
```

Funcţionarea algoritmului este următoarea: Înainte de a intra în regiunea critică, fiecare proces apelează intrare_regiune cu un număr 0 sau 1 drept parametru de ordine. Astfel procesul aşteaptă până când e în regulă să intre. Presupunem că a intrat în regiunea critică, procesul apelează plecare_regiune pentru a indica faptul că a terminat şi să permită altui proces să intre dacă doreşte.

FUNCŢIONAREA SOLUŢIEI

Iniţial nici un proces nu se află în regiunea lui critică. În acest moment procesul 0 apelează intrare_regiune. Îşi arată interesul prin setarea elementului din vector corespunzător lui şi se setează randul_procesului pe 0. Acum există două cazuri, primul caz când procesul 1 nu este interesat şi intrare_regiune se termină imediat iar al doilea caz când procesul 1 apelează intrare_regiune şi face o buclă până când proces_interesat[0] devine FALSE adică când procesul 0 apelează plecare_regiune pentru a părăsi regiunea critică.
Dacă presupunem că ambele procese apelează aproape simultan intrare_regiune. Asta înseamnă că ambele vor salva în rândul_procesului numărul său propriu de ordine. Dacă presupunem că procesul 1 salvează ultimul şi rândul_procesului este 1. Când ambele procese ajung la instrucţiunea while, procesul 0 nu o execută niciodată şi intră în regiunea critică proprie iar procesul 1 intră în bucla până când procesul 0 nu iese din zona critică.

2.3.3. EXCLUDEREA RECIPROCĂ FĂRĂ AŞTEPTARE

Până acum am discutat despre soluţiile de excludere reciprocă cu aşteptare. Această aşteptare iroseşte cum era de la sine înţeles timp pe procesor. Din **[STĂNCESCU]** ştim că soluţia este blocarea resurselor fără procese şi trezirea lor la apariţia condiţiilor astfel în loc să risipim timpul procesorului, blocăm apelantul şi eficientizăm excluderea reciprocă a proceselor.

Pentru următoarele metode vom folosi o problemă clasică, problema Producător-Consumator, aici folosind varianta cu un singur producător şi consumator pentru a simplifica soluţiile.

2.3.3.1.METODA CU APELURI SLEEP/WAKE-UP

Metoda cu apeluri sleep/wake-up este o metodă simplu de implementat unde folosim apelul de sistem *sleep* care are ca efect blocarea apelantului, adică suspendarea execuţiei acestuia şi apelul de sistem wake/up pentru a trezi un proces.

Funcţionarea acestei metode este următoarea **[STĂNCESCU]**: Fiecare dintre procese va verifica dacă este cazul să îl trezească pe celălalt şi dacă da, îl va trezi. Condiţia de cursă apare pentru că accesul la condiţia Magazia plină? este nerestricţionat. Poate apărea următoarea situaţie- zona tampon este goală şi consumatorul tocmai a verificat condiţia Magazia goală? pentru a verifica dacă este DA. În acest moment planificatorul de procese decide să suspende execuţia consumatorului şi începe să execute producătorul. Producătorul introduce un element în magazie şi observă că condiţia Magazia plină? este acum DA. Deoarece condiţia tocmai a fost NU şi deci consumatorul este suspendat, producătorul apelează wakeup pentru a trezi consumatorul. Consumatorul nu este şi logic suspendat şi semnalul de trezire se va pierde şi astfel data următoare când consumatorul se va executa acesta va verifica condiţia Magazia goală? citită anterior, va găsi DA şi îşi va suspenda execuţia După câtva timp producătorul îşi va suspenda şi el execuţia. Ambele procese vor fi suspendate.

Problema acestei metode este pierderea suspendării consumatorului. Soluţie : memorarea bitului de trezire, dacă se trimite unui proces treaz. Ulterior, dacă acesta vrea să se autoblocheze, nu o va face, dar va şterge bitul de memorare a trezirii. Fiecare proces care poate trezi un proces încă treaz va trebui să îşi aibă bitul memorat, împreuna cu adresa procesului la care se referă.

2.3.3.2. METODA CU SEMAFOARE

Metoda cu semafoare are la bază conceptul de semafor ca variabilă care are valoarea 0 dacă nu a existat o trezire prealabilă şi N dacă au existat n treziri prealabile (o trezire prealabilă înseamnă că semaforul se numeşte Mutex). Semaforul este înzestrat cu două operaţii, down şi up. Operaţia de down efectuată asupra unui semafor verifică dacă valoarea acestuia este mai mare decât 0. Dacă da, decrementează valoarea şi continuă. Dacă valoarea este 0 procesul este suspendat. Verificarea, modificarea şi eventuala suspendare a procesului sunt efectuate ca o singură acţiune atomică (termenul apare în **[TANENBAUM]**, autorul dorind să definească astfel operaţia ca fiind indivizibilă). Această atomicitate este esenţială pentru rezolvarea problemelor de sincronizare şi pentru evitarea condiţiilor de cursă. Operaţia de up incrementează valoarea semaforului.

REZOLVAREA PROBLEMEI PRODUCĂTOR-CONSUMATOR

Semafoarele rezolvă problema pierderii semnalelor de trezire. Această soluţie foloseşte trei semafoare: unul numit full, pentru numărarea locurilor ocupate, altul numit empty pentru numărarea locurilor libere şi al treilea numit mutex care asigură accesul producătorului şi consumatorului la zonă.

2.3.3.3. METODA CU MONITOARE

Metoda cu monitoare impune o anumită protejare a zonei în care se face comunicaţia interprocese prevenind interblocările care apar în anumite situaţii. Monitorul este o colecţie de proceduri, variabile şi structuri de date formând un modul destinat implementării comunicaţiei interprocese asigurând excluderea reciprocă prin condiţia ca un singur proces activ să fie în el astfel încât orice proces care intră într-un monitor se blochează **[STĂNCESCU]**. Procedurile monitorului sunt semafoarele un fel de anticameră a zonei critice.

Soluţia rezolvării problemei producător-consumator conform **[TANENBAUM]** se face prin introducerea variabilelor de condiţii: WAIT() şi SIGNAL(). Când o procedură de monitor descoperă că numai

poate continua atunci efectuează o operaţie de wait asupra unei variabile de condiţie, de exemplu full. Această acţiune determină blocarea procesului apelant. De asemenea, permite altui proces, căruia i se interzisese accesul în monitor, să intre. Excluderea mutuală automată a monitoarelor garantează că dacă de exemplu producătorul aflat în interiorul monitorului observă că magazia e plină, acesta va putea termina operaşia de wait fără a se îngrijora că se poate comuta la consumator.
Rezolvarea cu monitor a problemei producător-consumator:

```
monitor producator_consumator
        procedură in
                if (count == N) then
                        WAIT(FULL);
                intra_produs;
                count++;
                if (count == 1) then
                        SIGNAL(EMPTY);
        end in

        procedură out
                if (count == 0) then
                        WAIT(EMPTY);
                scoate_produs;
                count--;
                if (count == N-1) then
                        SIGNAL (FULL);
        end scoate
end monitor

procedură producator
        while(true) do
                produce;
                producator_consumator.in;
        end while;
end producător

procedură consumator
        while (true) do
                producator_consumator.out;
        end while;
end consumator;
```

Putem face o comparaţie între excluderea cu semafoare şi cu monitor ca în **[STĂNCESCU]**:
În cazul monitoarelor funcţiile încapsulate WAIT şi SIGNAL înlocuiesc funcţiile individuale SLEEP si WAKEUP; excluderea reciprocă se automatizează în cazul monitoarelor şi astfel la semafoare ea este proiectată de către utilizator.
Tratarea cu monitoare ridică următoarele două probleme:
1. Este dificil de incorporat î compilatoare secvenţa de cod care să implementeze conceptul de monitor;
2. În cazul unui sistem cu mai multe procesoare care partajează o memorie comună rezolvarea este mai simplă cu semafoare.

3. IMPLEMENTAREA PROCESELOR ŞI FIRELOR DE EXECUŢIE ÎN LINUX ŞI ÎN WINDOWS

(*Petre Tiberiu, 443A)*

3.1.DEFINIŢII ŞI GENERALITĂŢI

Proces : o instanta a unui program aflat in executie.

Fir de executie(thread) : o parte a unui proces care se executa independent de alte parti ale acestuia.

Procesele au urmatoarele proprietati :

3. au alocat un spatiu virtual de memorie in care este stocat codul programului;
4. au drepturi de acces protejat la fisiere,la resurse de intrare/iesire,la memoria alocata altor procese;

Firele de executie au urmatoarele proprietati :

- au o stare de executie(in executie,oprit,in asteptare etc.);
- isi salveaza contextul atunci cand nu sunt in executie;
- au acces numai la spatiul de memorie alocat procesului din care au fost lansate.

Din **[TANENBAUM]** stim ca daca sunt implementate in sistemul de operare,firele de executie au urmatoarele avantaje fata de procese:

- un fir de executie este creat mai repede decat un proces deoarece utilizeaza spatiul de memorie al pocesului din care a fost lansat;
- un fir de executie poate fi oprit mai repede decat un proces;
- se poate trece usor de la un fir de executie la altul deoarece este utilizat acelasi spatiu de memorie;
- comunicarea intre firele de executie ale unui proces este foarte usoara deoarece ele utilizeaza acelasi spatiul memorie iar datele generate de un fir de executie sunt imediat disponibile celorlalte fire de executie.

Proprietatea unui sistem de operare de a suporta fire de executie se numeste *multithreading*.

3.2. IMPLEMENTAREA ÎN LINUX

In Linux nu exista o diferenta precisa intre procese si fire de executie.Un proces “parinte” poate crea mai multe procese “copil” care acceseaza acelasi spatiu de memorie alocat procesului parinte.Atunci cand unul dintre copii incearca sa modifice(sa scrie) o zona de memorie a procesului parinte,este creata o copie a acelei zone de memorie pe care va opera in continuare procesul copil,nefiind astfel nevoie sa fie creata o copie a intreg spatiului de memorie al procesului parinte pentru fiecare copil(se foloseste mecanismul copy-on-write).Procesele copil pot fi asemanate cu firele de executie. **[RUSLING]**

Pentru gestionarea proceselor in Linux,kernelul aloca fiecarui proces o structura de date numita *task_struct*(definita in fisierul linux/include/linux/sched.h din codul sursa al kernelului).Rezulta o colectie de structuri de date *task_struct* care va fi reprezentata sub doua forme:

- sub forma de vector(tabel de cautare) de structuri *task_struct*;
- sub forma de lista circulara dublu-inlantuita de structuri *task_struct*.

In reprezentarea sub forma de tabel,fiecarui proces ii este asociat un identificator de proces(PID).Relatia dintre PID si indexul unei structuri *task_struct* in vector este urmatoarea(in versiunea 2.4 a kernelului) :
Index = ((((PID) >> 8) ^ (PID)) & (PIDHASH_SZ − 1)) ,unde PIDHASH_SZ este dimensiunea tabelului de cautare.

Reprezentarea sub forma de lista circulara dublu-inlantuita este folosita pentru a stabili ordinea de executie a proceselor. **[RUSLING]**

Structura *task_struct* este foarte complexa insa campurile ei pot fi impartite in urmatoarele categorii

Stari: Orice proces isi schimba starea in functie de contextul in care se afla.
Putem avea urmatoarele tipuri de stari :

- *in executie* : procesul ruleaza sau este capabil sa ruleze dar asteapta sa fie preluat de catre microprocesor;
- *in asteptare :* procesul asteapta un eveniment sau eliberarea unei resurse;procesele in asteptare pot fi intreruptibile(pot fi oprite cu anumite semnale de oprire) si neintreruptibile(functionarea lor este conditionata de partea hardware si nu pot fi intrerupte folosind semnale);
- *oprit :* procesul a primit un semnal de oprire;
- *Zombie :* sunt procese oprite dar care inca mai au o structura *task_struct* alocata in tabelul de cautare.

Informatii utile coordonarii proceselor:in functie de aceste informatii sunt distribuite resursele si prioritatile de executie proceselor.

Identificatori:ID de utilizator,ID de grup s.a. ; in functie de aceste informatii sunt stabilite drepturile de acces ale proceselor la sistemul de fisiere.

Comunicare intre procese: sunt suportate diverse mecanisme de comunicare intre procese : semafoare,semnale,cozi de mesaje,tevi,memorie partajata.

Legaturi : orice proces are un proces parinte.Parintele tuturor proceselor din Linux este procesul init(1).Structura task_struct contine pointeri catre parintele procesului respectiv,catre alte procese care au acelasi parinte(“frati”),catre procesele copil.

Cronometre : contoare care tin evidenta timpului consumat de fiecare proces in parte.In functie de aceste contoare procesul isi poate trimite anumite semnale la anumite momente de timp in executia sa.

Crearea proceselor :
1. La pornirea sistemului in modul nucleu,exista un singur proces initial parinte.
2. Dupa ce sistemul a fost initializat acest proces parinte lanseaza un fir de executie(un proces copil) numit **init** dupa care ramane inactiv.Structura task_struct alocata procesului parinte este singura care nu este alocata dinamic ci este declarata ca variabila statica in codul sursa al kernelului(se numeste *init_task*).
3. Dupa ce a fost lansat in executie,procesul **init** initializeaza sistemul(initializeaza consola,monteaza sistemul principal de fisiere) dupa care,in functie de continutul fisierului /etc/inittab lanseaza in executie alte procese.
4. Din aceasta faza noi procese pot fi create prin clonarea celor deja existente prin apelarea unor proceduri de sistem(clone,fork,vfork). **[TANENBAUM]**

3.3.IMPLEMENTAREA ÎN WINDOWS

La cel mai inalt nivel de abstractie,un proces Windows consta in urmatoarele elemente :

- Un spatiu virtual de adrese privat(memoria pe care o are procesul la dispozitie);
- Un program executabil care contine instructiunile programului si care va fi incarcat in spatiul virtal de

adrese;
- O lista de legaturi spre resurse de sistem,porturi de comunicatie,fisiere;
- Un context de securitate reprezentat de un nume de utilizator,o lista de grupuri din care face parte utilizatorul,privilegiile de care are parte procesul;
- Un identificator de proces unic(intern se mai numeste si identificator de client);
- Cel putin un fir de executie.

Un proces Windows mai contine si un *pointer catre procesul parinte*(din care a fost lansat).Acest pointer poate fi nul.

Firul de executie este componenta fundamentala a unui proces.Un proces fara fire de executie nu se poate executa.Un fir de executie din Windows poate fi descris de urmatoarele componente **[RUSSINOVICH]** :
- Starea curenta a procesorului descrisa de valorile din registri(contextul);
- Doua stive fiecare petru executia intr-unul din cele doua moduri : utilizator sau nucleu;
- Un identificator al firului de executie(face parte din acelasi spatiu cu identificatorii de procese astfell ca nu va putea exista un proces si un fir de executie cu acelasi identificator);
- Uneori firele de executie au si un context de securitate

Formatul contextului unui fir de executie este dependent de arhitectura masinii pe care ruleaza sistemul de operare.Cu ajutorul metodei *GetThreadContext* se pot extrage informatii din contextul firului de executie.

Fibrele sunt niste fire de executie care pot fi lansate in executie manual prin apelul metodei *SwitchToFiber*.Spre deosebire de fibre,firele de executie ruleaza automat cu ajutorul unui mecanism de coordonare bazat pe prioritati.

Ca si in Linux,in Windows exista doua moduri de acces la microprocesor : modul utilizator si modul nucleu.Diferentierea modurilor de acces este necesara pentru a preveni accesul utilizatorului la elemente critice ale sistemului de operare.In acest fel un eventual comportament neobisnuit al unei aplicatii destinate utilizatorului nu va perturba functionarea intregului sistem de calcul. **[TANENBAUM]**

Pentru gestionarea proceselor in Windows,fiecarui proces ii este alocata o structura *EPROCESS*. Pe langa diversi parametri ai procesului,aceasta structura mai contine si pointeri catre alte structuri cum ar fi structuri de tip *ETHREAD* care descriu fire de executie sau spre alte structuri *EPROCESS*(ex. Spre procesul parinte).

Crearea unui proces :
Un proces este creat atunci cand un alt proces apeleaza metoda *CreateProcess* din biblioteca kernel32.dll. Crearea procesului se face in urmatorii pasi :
1. Fisierul executabil este analizat.Daca este un fisier executabil Windows pe 16 biti sau MS-DOS atunci este creat un mediu special de executie pentru acesta;daca fisierul este un executabil Windows pe 32 de biti se verifica registrul pentru a vedea daca are vreo cerinta speciala.Operatiile de la acest pas se fac in modul utilizator.
2. Se creeaza un obiect proces gol cu apelul de sistem NtCreateProcess si se adauga in spatiul de nume al managerului de obiecte. Mai sunt create un obiect nucleu si unul executiv. Managerul de proiecte creeaza un bloc de control al procesului pentru acest obiect si il initializeaza cu Idul procesului si cu alte campuri.Este creat un obiect sectiune pentru a urmari spatiul de adrese al procesului.
3. kernel32.dll preia controlul,efectueaza apelul de sistem *NtCreateThread* pentru a crea un fir de executie initial.Se creeaza stivele utilizator si nucleu.Dimensiunea stivei este data de antetul fisierului executabil.
4. kernel32.dll trimite subsistemului de mediu Win32 referintele catre proces si catre firul de executie care sunt adaugate in niste tabele.
5. Este apelata o procedura de sistem care termina initializarea iar firul de executie incepe sa ruleze.
6. O procedura apelata la executie seteaza prioritatea firului de executie.Se repeta pasul pentru toate firele de executie.Incepe rualrea codului principal al procesului. **[TANENBAUM]**

4. APELURI DE SISTEM DE GESTIUNE A PROCESELOR ÎN LINUX
(*Licu Dragoş, 443A*)

4.1. INTRODUCERE

Procesele, pentru un sistem de operare LINUX sunt considerate singurele entitati active, Avand initial un singur fir de executie de control fiecare proces ruleaza ca un program de sine statator. Fire de executie aditionale sunt create dupa lansearaea executiei procesului.

Datorita faptului ca LINUX permite executarea simultana a mai multor procese individuale, face ca acesta sa fie un sistem de operare multitasking, fiecare utilizator al sistemului putand avea in acelasi timp actine mai multe procese, numarul acestora putand ajunge de ordinal sutelor.Chiar si cand utilizatorul nu ruleaza nici o aplicatie care sa implice activarea rularii unor anumite procese, anumite procese ruleaza in memoria sistemului. Aceste procese se numesc “daemon” si sunt pornite de sistemul de operare la bootare. Un exemplu de astfel de process este deamon-ul crom, care se trezeste odata pe minut si pentru a verifica daca exita vre-un task de executat pentru el. In eventualitatea existentei acestui task acesta este indeplinit dupa care daemonul adoarme, pana la sosirea noului interval de executie. Utilitatea acestor daemon-uri se observa in specil atunci cand dorim sa programam dinainte cu cateva ore sau chiar zile, executarea unui anumit task. Una din responsabilitatile deamonului crop este aceea de a rula activitati periodce cum ar fi verificarea existentei update-urilor pentru sistemele de fisiere, sau rularea aplicatiilor periodice de back-up. Exista daemon-uri care se ocupa de verificarea email-ului, verificarea existentei de pagini libere in memorie, gestionarea ordinii de printare a documentelor. Unul din atuurile daemon-urilor in LINUX este acela ca ei pot fi implementati foarte usor deoarece ei sunt atat independent unul de celalalt cat si de celelalte procese. **[TANENBAUM]**

4.2. CREAREA PROCESELOR

Linux se bucura de un mod aparte de creare al proceselor si anume se creeaza o copie a procesului original(“proces parinte”). Referirea la aceasta copie poarta numele de “proces copil”. Prin creerea acestor copii LINUX are avantajul ca daca variabilele din procesul parinte se modifica variabilele din procesul copil nu sunt afectate, si invers; in acest timp daca un fisier a fost deschis fie de “procesul parinte” fie de “procesul copil”, fisierul va fi vizibil pentru ambii, iar modificarile facute de unul dintre cei doi vor fi vizibile si pentru celalalt.Acest lucru poate fi considerat doar un avantaj partial deoarece aceste modificari sunt vizibile si proceselor care nu au nici o legatura cu cei doi. **[TANENBAUM]**

Un oarecare grad de dificultate il ridica faptul ca variabilele, registri si imaginile memoriei sunt identice pentru copil si pentru parinte, de unde si intrebarea cum disting procesele care dintre ele sa execute codul parinte si care codul copil. Raspunsul se gaseste in faptul ca sistemul de apel fork intoarece valori diferite pentru PID-ul (Processs ID) parintelui si copilului. Pentru copil se intoare valoarea zero iar pentru parinte o valoare diferita de zero.Valoarea intoarsa este verificata de ambele procese iar sitemul se operare se comporta adegvat, dupa cum este aratat mai jos.

```
Pid = fork();              /*daca apelul fork reuseste, pid>0 in parinte */
If (pid <0){
Handle_error();            /*apelul fork a esuat */
} else if (pid>0) {
                           /*aici se gaseste codul parintelui */
} else {
                          /*aici se gaseste codul copilului   */
}
```

Crearea proceselor in LINUX

4.3. COMUNICAREA ÎNTRE PROCESE

In LINUX comunicarea dintre procese este posibila si acest lucru se realizeaza prin transmiterea de mesaje.Un canal este creat intre doua procese iar fiecare dintre procese poate initia un plux de date destinat celuilalt proces.Aceste canale poarta numele de “pipes”(conducte). Sincronizarea se face prin blocarea procesului care incearca sa citeasca date dintr-o conducta goala pana cand vor esista date in aceasta conducta.

In momentul in care interpretorul de comenzi intalneste o linie precum “sort < f | head ”, doua prcese sunt create si o conducta este initiate intre acestea.Cele doua procese sunt sort si head. Conducta

interconecteaza cele doua procese unind iesirea standard a lui soft cu intrarea standard a lui head, astfel asiguranduse faptul ca datele create de sort sunt directionate catre head in loc ca acestea sa ajunga intr-un fisier.In cazul in care conducta se head nu reuseste sa proceseze datele in ritmul in care sort le initiaza, iar conducta se umple sistemul suspenda activitatea lui sort, pana cand head reuseste sa elibereze o parte din conducta. **[TANENBAUM]**

4.4. ÎNTRERUPERILE SOFT ŞI SEMNALELE CERUTE DE POSIX

O alta modalitate de comunicare intre procese il reprezinta intreruperile soft. Functionarea aceastei de a doua modalitati se bazeaza pe semnale. In momentul in care un proces sesiseaza existent unui semnal, procesul comunica sistemului modul de abordare pe care doreste sa il adopte(sa intercepteze semnalui, sa il ignore sau sa permite semnaluilui terminarea procesului). Trimiterea de semnale de catre un proces este permisa catre membrii grupului din care face parte folosindu-se de un singur apel de gestiune.

In momentul in care un proces primeste un semnal si doreste sa il interpreteze, procesul specifica o procedura de tratare a semnalui, iar controlul se cedeaza procedurii respective la receptia semnalului, controlul revenind la terminarea semnalului.

Semnalele mai sunt utilizate si la detectia erorilor. Spre exemplu daca la executarea de operatii aritmetice se imparte la 0 procesul respective primeste un semnal SIGFPE. Mai jos sunt cateva exemple de semnal cerute de POSIX si cauzele acestora.

Semnale cerute de POSIX

Semnal	Cauza
SIGABRT	Trimis pentru a renunta la un process si a forta golirea memoriei
SIGALRM	Ceasul de alarma s-a oprit
SIGFPE	Eroare de virgule mobile (ex imparirea la zero)
SIGHUP	Linia telefonica folosita de process a fost suspendata
SIGILL	Utilizatorul a apasat tasta DEL pentru intreruperea procesului
SISQUIT	Utilizatorul a apasat tasta cere cere golirea memoriei
SIGKILL	Trimis pentru a omora un proces
SIGPIPE	Procesul a scris intr-o conducta care nu are cititori
SIGSEGV	Procesul s-a referit la o adresa invalida de momorie
SIGTERM	Polosit pentru a cere ca un process sa se termine de buna voie
SIGUSR1	Disponibil pentru scopuri definite de aplicatie
SIGUSR2	Disponibil pentru scopuri definite de aplicatie

4.5. EXEMPLE DE APELURI DE SISTEM DE GESTIUNE A PROCESELOR ŞI FUNCŢIONAREA ACESTORA

In tabelul urmator sunt prezentate cateva din cele mai importante apeluri de sistem de gestiune a proceselor si descrierea acestora.

Apel de sistem	Descriere
Pid = fork( )	Se creeaza un process copil identic cu parintele
Pid = waitpid(pid, &statloc, opts)	Asteapta ca un copil sa se termine
S = execve(name, argv, envp)	Inlocuirea miezului imaginii unui proces
Exit(status)	Terminarea executiei procesului si intoarcerea lui status
S = sigaction(sig, &act, &oldact)	Definirea unei actiuni de indeplinit pe semnale
S = sigreturn(&context)	Intoarcerea dintr-un semnal
S = sigprocmask(how, &set, &old)	Examinarea sau schimbarea unei masti de semnale
S = sigpending(set)	Intoarcerea setului de semnale blocate
S = sigsuspend(sigmask)	Inlocuirea mastii de semnale si suspendarea procesului
S = kill(pid, sig)	Trimiterea unui semnal la un process
Residual = alarm(seconds)	Setarea ceasului de alarma
S = pause ()	Suspendarea apelantului pana la urmatorul semnal

Exemple de apeluri de sistem care au legatura cu procesele

Unica modalitate de creare a unui process nou in LINUX este FORK. Acesta creeaza o copie a a procesului incluzand registrii si descriptorii de fisier. In momentul in care se face Fork absolult toate variabilele au aceleasi valori atat pentru “parinte” cat si pentru “copil”.Dupa momentul de Fork parintele si copilul merg pe cai diferite modificarile survenite ulterior intr-unul dintre ele nu il afecteaza pe pe calalalt. Ca un proces sa poata distinge intre parinte si copil se foloseste PID-ul intors. Apelarea Fork-ului intoarce in parinte o valoare egala cu PID-ul copilului, si valoarea zero in copil. **[TANENBAUM]**

Sa luam exemplul consolei. In aceasta situatie comanda data de utilizator este executata de copil. Pentru aceasta se foloseste apelul de sistem exec care inlocuieste fisierul din primul parametru cu miezul intregii imagini. In continuare este prezentat un interpertor de comenzi si utilizarea lui fork, waitpid si exec.

```
While(TRUE) {                                   /* repeat la nesfarsit */
        type_prompt();                          /* se afiseaza prompt-ul pe ecran */
        read_command(command, params);    /* citire de la tastatura */

                pid  =  fork();                 /* crearea unui process copil cu fork */
                 if (pid<0) {
                 printf(“Unable to fork0);     /* situatie de eroare */
                continue;                       /* repeat bucla */
                }

                  if(pid!=0) {
                     waitpid(1, &status, 0)         /* parintele asteapta copilul */
                }else {
                   Execve(command, params, 0);        /*copilul executa treaba */
                    }
         }
```

Interpretor de comenzi simplificat

Cei trei parametric generali ai lui exec sunt : numele fisierului ce trebuie executat, un pointer la vectorul de argumentare si un pointer la vectorul de mediu.

In majoritatea situatiilor, dupa creearea unui copil, prin apelarea fork, acesta executa cu totul si cu totul alt cod fata de parinte. Procesul tata asteapta de obicei ca copilul sa execute ce are de facut si asteapta ca acesta sa se termine prin executarea apelului de sistem waitpid. Parametri waitpid sunt trei la numar, si anume : primul parametru este folosit pentru a permite apelantului sa astepta un anumit copil, al doilea parametru este adesa unei variabile ce primeste valoarea statutului de iesire al copilului, iar al treilea parametru este folosit pentru a afla daca apelantul se blocheaza. **[TANENBAUM]**

Unele apeluri de sistem lucreaza in stanza legatura cu semnalele. Pentru a intelege acest aspect mai bine sa consideram urmatorul exemplu.Sa presupunem ca dorim sa accesam anumite date dintr-un fisier text, si din greseala punem editorul de fisere text sa deschida intre-gul fisier care este foarte lung. Dandu-ne seama de asta dorim sa anulam deschiderea intregului fisier. Pentru a efectua acest lucru de cele mai multe ori folosim o tasta sau o combinative de taste pentru a trimite un semnal sistemului, iar acesta sat ermine executia procesului de deschidere al fisierului. Pentru interceptarea unui semnal trimis in aceasta maniera sistemul poate folosi apelul de sistem sigaction. Acest apel de sistem mai poate fi folosit si in alte cazuri cum ar fi ignorarea semnalului. In momentul in care un semnal a fost tratat ca atare procedura se inchide si se revine in starea in care a fost initiate intreruperea. Timpul de tratare al intruruperilor in practica este destul de scurt, dar aceste rutine de examinare si executie a semnalelor poate dura atat timp cat doreste. **[TANENBAUM]**

Alarm este un apel de sistem care faciliteaza intreruperea unui proces dupa un anumit interval de timp definit. Acest lucru se face pentru a permine procesului respective sa efectueze anumite functii cum ar fi de exemplu retransmiterea anumitor date care nu au ajuns la destinatie sau au ajuns corupte sau retransmiterea unui pachet pierdut pe o linie de comunicatie unreliable sau best effort delivery. In general astfel de procese care necesita intreruperi sunt specific aplicatiilor in timp real.

Exista cazuri in care un proces, in lipsa unor comenzi date de utilizator sa nu aiba mimic de facut pana la sosirea urmatorului semnal. Pentru a nu consuma resursele sistemului prioritaea accesului ca unitatea de calcul a sistemului este cedata altui process.O solutie foarte buna pentru a evita irosirea resurselor este aceea sa se foloseasca apelul de sitem pause care efectueaza suspendarea procesului pana la sosirea urmatorului semnal. **[TANENBAUM]**

5. PLANIFICAREA PROCESELOR. ALGORITMUL DE PLANIFICARE ÎN UNIX, LINUX, WINDOWS NT

(*Marinescu Raluca, 443 A)*

Problema concurenţei proceselor apare frecvent pe sistemele de calcul ce folosesc multiprogramarea deoarece procesele sunt gata de execuţie simultan. Deoarece există o singură unitate centrală de prelucrare, trebuie făcută o alegere cu privire la procesul care va rula primul folosind un algoritm numit algoritm de planificare.

5.1 PROBLEMA PLANIFICĂRII

În cazul calculatoarelor personale există un singur proces în execuţie în majoritatea timpului. Pe altă parte, calculatoarele au devenit rapide şi capacitatea UCP nu mai este ca pe vremuri o resursă limitată. Singura limită pentru programele pentru PC-uri este în general viteza de introducere a datelor şi nu viteza de procesare a unităţii centrale de prelucrare. Consecinţa este faptul ca planificarea nu contează prea mult pe PC-urile obişnuite. În cazul staţiilor de lucru şi serverelor de reţea de nivel înalt, situaţia este alta. Avem mai multe procese care sunt frecvent în competiţie pentru UCP si planificarea devine foarte importantă.
Conform **[TANENBAUM]** paşii ce sunt urmaşi în cazul unui algoritm de planificare sunt:

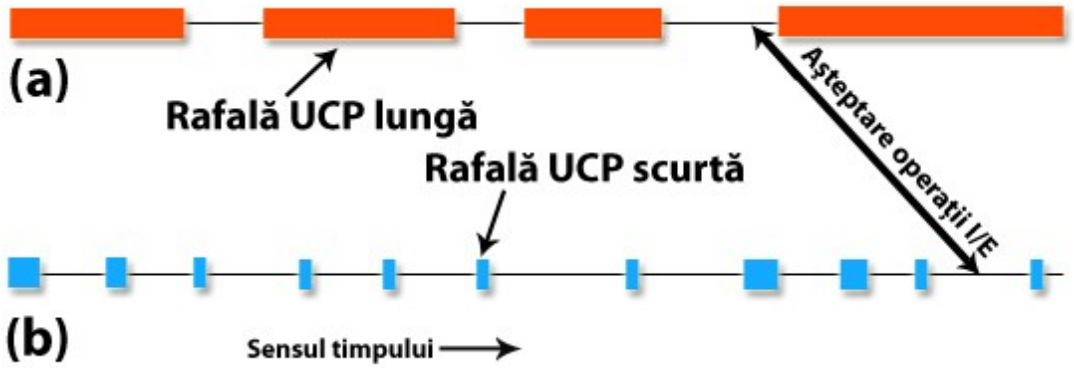
- corectitudinea alegerii programului de executat, astfel incat planificatorul sa folosească eficient UCP-ul;
- producerea comutării din modul utilizator în modul nucleu;
- se salvează starea procesului curent prin stocarea registrelor acestuia în tabela de procese pentru restaurare ulterioară (in cazul multor sisteme trebuie salvată si harta memoriei);
- alegerea unui proces;
- Unitatea de Gestiune a Memoriei trebuie reactualizată cu harta nouă a memoriei pentru un nou proces.

Dacă ţinem cont că de obicei se poate produce un page fault când apare comutarea de procese atunci reîncărcarea paginii din memoria principală se face de două ori, odată în modul nucleu şi încă o dată la ieşire şi după cum se ştie asta mănâncă timp şi resurse.

5.2. O CLASIFICARE A PROCESELOR

Procesele alternează între calculele în rafală şi cererile de operaţii de I/E. UCP-ul rulează o perioadă de timp, apoi se efectuează un apel de sistem pentru citirea sau scrierea într-un fişier. În momentul finalizării apelului, UCP-ul calculează din nou până când are nevoie de alte date de intrare sau trebuie să scrie date.

Figura 5.2. Utilizarea UCP în rafale şi aşteptarea operaţiilor I/E. (a) proces ce foloseşte rafale UCP lungi. (b) proces ce foloseşte rafale UCPscurte.



În figura 5.2. se poate vedea utilizarea UCP în rafale de către un proces şi timpii de aşteptare pentru operaţiile I/E. În acest moment se poate face o prima clasificare a proceselor:

1. procese ce fac aproape mereu calcule, procese numite procese limitate de calcul.
2. procese ce aşteaptă aproape mereu completarea operaţiilor de I/E se numesc procese limitate de I/E.

Procesele limitate de calcule au în mod obişnuit rafale de utilizare a UCP mai lungi şi perioade de aşteptare a operaţiilor de I/E rare, în timp ce procesele limitate de I/E au rafale UCP scurte şi perioade frecvente de aşteptare a operaţiilor de I/E. Pe măsură ce UCP-urile devin mai rapide, procesele tind să devină din ce în ce mai limitate de I/E. Aceste observaţii se pot observa foarte bine şi pe figura 5.2.

5.4. CATEGORII ŞI PRINCIPII DE PROIECTARE ALE ALGORITMILOR DE PLANIFICARE

5.4.1. ALGORITMII DE PLANIFICARE ŞI MODUL ÎN CARE TRATEAZĂ ÎNTRERUPERILE DE CEAS

Deciziile de planificare conform **[TANENBAUM]** se pot lua la fiecare întrerupere de ceas sau la un număr fix de întreruperi. Acest lucru se poate întâmpla doar dacă un ceas hard oferă periodic întreruperi. Algoritmii de planificare pot fi grupaţi în două categorii cu privire la modul în care tratează întreruperile de ceas.

- **algoritmii non-preemptivi** aleg un proces pentru execuţie şi apoi îl lasă să ruleze până se blochează de la sine sau până când procesul vrea el să renunţe la UCP; chiar dacă rulează ore întregi, procesul nu va fi suspendat forţat, ci doar după ce s-a terminat procesarea întreruperii, procesul care rula înainte de întrerupere poate fi executat în continuare;
- **algoritmii de planificare preemptivi** aleg un proces şi îl lasă să se stea o durată maximă fixă; în caz că procesul este încă în execuţie la sfârşitul intervalului de timp, este suspendat si planificatorul alege alt proces (daca există un proces disponibil); efectuarea unei planificări preemptive necesită o întrerupere de ceas la sfârşitul intervalului menţionat pentru a muta controlul asupra UCP înapoi la planificator.

  Dacă nu există un ceas, singura opţiune este planificarea non-preemptivă.

5.4.2. FOLOSIREA ALGORITMILOR DE PLANIFICARE ÎN FUNCŢIE DE TIPUL DE APLICAŢIE FOLOSIT

Algoritmii de planificare diferă în mod normal de la aplicaţie la aplicaţie. Ştim 3 tipuri de sisteme ce au aplicaţie diferită şi acoperă aproximativ tot domeniul sistemelor de calcul: sisteme cu prelucrare pe loturi, sisteme interactive şi sisteme în timp real.

În sistemele cu prelucrare pe loturi nu există utilizatori care aşteaptă un răspuns rapid la terminalele lor. În consecinţă sunt acceptabili algoritmi non-preemptivi sau algoritmi preemtivi cu perioade mari de timp pentru fiecare proces. Aceşti algoritmi îmbunătăţesc performanţele acestui sistem.

În sistemele interactive (de care ne vom ocupa în 5.5.), algoritmii preemptivi sunt necesari pentru a nu avea un proces care să ocupe tot UCP-ul şi să nu lase nici alte procese să-l folosească. Chiar dacă nici un proces nu ar rula intenţionat la infinit, un proces ar putea să le lase pe celelalte o perioadă de timp nedefinită datorită unei erori de program. Algoritmii preemptivi sunt necesari în sistemele interactive.

În sistemele de timp real, utilizarea algoritmilor preemptivi nu este întotdeauna necesară pentru că procesele ştiu că nu pot rula perioade lungi de timp şi ca atare îşi efectuează sarcina şi se blochează rapid. Diferenţa faţă de sistemele interactive este că sistemele de timp real execută numai programe care sunt create pentru scopul aplicaţiei curente. Sistemele interactive au un scop general şi pot rula programe arbitrare care nu sunt cooperante sau sunt chiar râu intenţionate.

5.4.3. SCOPURILE GENERALE ALE ALGORITMILOR DE PLANIFICARE

Conform **[TANENBAUM]** pentru a proiecta un algoritm de planificare, trebuie să ştim ce ar trebui să facă un algoritm bun.

Corectitudinea este un element important. Procese comparabile trebuie să bencificieze de servicii comparabile. Nu trebuie sa aloce mai multă capacitate UCP unui proces decât altuia care este echivalent.

Respectarea politicilor sistemului este un concept egal cu cel al corectitudinii. Daca vrem ca procesele de control al siguranţei să se execute când doresc, planificatorul trebuie să se asigure că această politică este respectată.

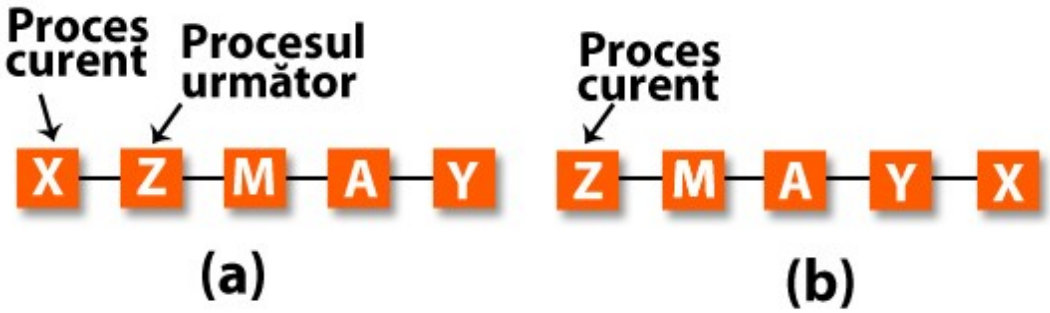
Componentele sistemului să fie ocupate cât mai mult timp. Daca UCP-ul şi toate dispozitivele de I/E pot fi menţinute în operare tot timpul, se efectuează mai multe operaţii în fiecare secundă.

5.5.CONCEPTE DE PROIECTARE A PLANIFICĂRII ÎN SISTEMELE INTERACTIVE

5.5.1. PLANIFICAREA ROUND ROBIN

Conform algoritmului Round Robin, fiecare proces primeşte o cuantă de timp de rulare. Mai precis să zicem că la sfărşitul cuantei de timp, procesul este încă în execuţie, unitatea centrală de prelucrare lasă procesul curent şi trece la alt proces gata de execuţie. Pe de altă parte să zicem că înaintea terminării cuantei de timp procesul a rămas blocat sau în cazul fericit s-a terminat atunci, unitatea centrală de prelucrare comută pe alt proces fix în momentul în care s-a blocat procesul.

Figura 5.5.1. Planificarea Round Robin. (a) O listă de procese pregătite de execuţie. (b) Lista proceselor pregătite de execuţie după ce X şi-a consumat cuanta.



Planificarea Round Robin poate avea următoare formă din Figura 5.5.1 pentru o stare curentă şi o stare viitoare după ce un proces şi-a consumat cuanta. Cum se poate implementa acest algoritm? Planificatorul trebuie să aibă o listă a proceselor pregătite de execuţie şi când procesul îşi termină cuanta să fie pus la sfărşitul listei de procese. În general comutarea între procese ţine un timp de salvare a registrelor şi a memoriei, etc. Astfel la algoritmul Round Robin se pune problema cât să fie cuanta de mare? Conform **[TANENBAUM]** dacă comutarea durează 1 msec si valoarea cuantei este de 4 msec douăzeci la sută din timpul UCP este irosit pe supraîncărcarea cauzata de sarcini administrative. Este prea mult si de aceea am putem folosi şi cuante de 50-150 msec.

Concluzia este că folosirea unei cuante prea scurte poate cauza prea multe comutări de proces şi micşorează eficienţa UCP, dar folosirea unei valori prea mari poate cauza timpi de răspuns neadecvaţi pentru cereri interactive scurte. O cuantă de 20-50 msec este de multe ori bună.

5.5.2. PLANIFICAREA BAZATĂ PE PRIORITĂŢI

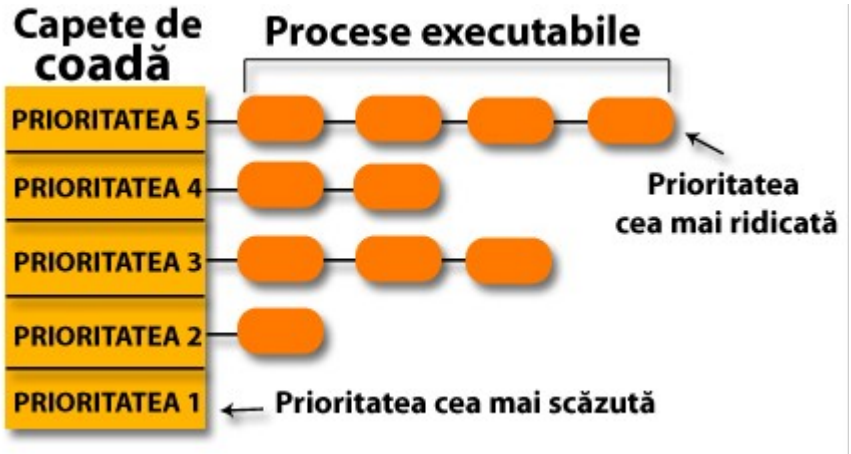
Faţă de planificarea Round Robin care pornea de la faptul că toate procesele nu au o importanţă diferită şi se primea o cuantă predefinită, la planificarea bazată pe priorităţi se ia în calcul că fiecare proces are o anumită prioritate pentru sistemul de operare şi astfel lăsăm mai întâi procesul pregătit de execuţie cu cea mai mare prioritate. Ca să nu cădem în extrema cealaltă şi să rulăm la infinit procese cu prioritate mare, planificatorul scade prioritatea procesului curent de fiecare dată când se produce o întrerupere de ceas. Normal poate să existe cazul când procesul curent scade sub următorul din listă şi acest proces are posibilitatea de a rula. Priorităţile se pot atribui proceselor în mod static sau dinamic.

Conform **[TANENBAUM]** sistemul UNIX are o comandă, nice, care permite unui utilizator să îşi reducă singur prioritatea procesului. Priorităţile pot fi asignate dinamic de către sistem pentru a îndeplini anumite scopuri. De exemplu, anumite procese sunt puternic limitate de I/E şi îşi petrec majoritatea timpului aşteptând să se termine operaţiile de I/E. Când un astfel de proces doreşte UCP, acesta ar trebui să-i fie atribuit imediat, pentru a permite procesului să facă următoarea cerere de I/E, care apoi se poate desfăşura în paralel cu alte procese care efectuează calcule. A face procesul orientat I/E să aştepte mult timp UCP ar însemna ca acesta să ocupe memorie inutil pentru o perioadă lungă de timp. Un algoritm simplu pentru a oferi un serviciu de calitate proceselor orientate I/E constă în a folosi o prioritate de 1/f, unde f este fracţiunea din cuanta de timp pe care procesul a folosit-o ultima dată. Un proces care a folosit doar 1 msec din cuanta sa de 50 msec ar primi prioritatea 50, în timp ce un proces care a rulat 25 msec înainte să se blocheze ar primi prioritatea 2, iar un proces care a folosit toată cuanta ar avea prioritatea 1.
Este adesea util să se grupeze procesele în clase de prioritate şi să se folosească planificarea bazată pe priorităţi între clase şi algoritmul round robin în cadrul fiecărei clase.

5.5.3.COMUTAREA DE PROCES ŞI PROBLEMA PRIORITĂŢILOR

Fiindcă comutarea de proces se face destul de lent pentru că memorarea proceselor se face unul câte unul, problema mare a planificării bazate pe priorităţi este comutarea de proces. Astfel nu este bine să dăm proceselor destinate calculelor o cuantă mică mereu ci putem da o cuantă mare din când în când. Dar

Figura 5.5.3. Algoritm de planificare cu 5 clase de priorităţi



bine nu este nici să dăm o cuantă mare de timp pentru că o să avem un timp slab de răspuns. Soluţia este de a crea clase de prioritate. Procesele din cea mai mare clasă sunt rulate pentru o cuantă de timp.

Procesele din următoarea clasă se execută pentru două cuante de timp, procesele din următoarea clasă pentru patru cuante, etc. Când un proces foloseşte în întregime cuanta alocată, este mutat în jos o clasă. Dacă să zicem un proces este mutat spre priorităţi mai mici atunci acesta va fi executat din ce în ce mai rar şi astfel UCP-ul execută procese scurte. Politica descrisă a fost adoptată pentru a preîntâmpina cazul în care un proces este amanat constant dacă are nevoie să se execute o durată mare de timp la început. În Figura 5.5.3 puteţi vedea un algoritm de planificare cu cinci priorităţi.

5.5.4. CEL MAI SCURT PROCES E URMĂTORUL (SHORTEST PROCESS NEXT)

Procesele interactive urmează de obicei modelul în care se aşteaptă o comandă, se execută comanda, se aşteaptă o comandă, se execută comanda, ş.a.m.d. Dacă privim execuţia fiecărei comenzi ca pe o lucrare separată, atunci putem minimiza întreg timpul de răspuns executând în continuare cea mai scurtă lucrare. Marea problema constă în a afla care din procesele executabile este cel mai scurt.

O abordare constă în a face estimări bazate pe comportamentul anterior şi a rula procesul cu cel mai mic timp de execuţie estimat. Metoda prin care se estimează următoarea valoarea într-o serie prin considerarea sumei ponderate a valorii curente măsurate şi a valorii estimate anterioare este câteodată numită îmbătrânire. Această tehnică se aplică în numeroase situaţii în care trebuie efectuată o prezicere bazată pe valori anterioare. îmbătrânirea este foarte uşor de implementat pentru $a = 1/2$ **[TANENBAUM] .**

5.5.5. PLANIFICAREA GARANTATĂ

O altă metodă de planificare constă în tehnica promisiunilor făcute utilizatorilor şi respectarea lor. În **[TANENBAUM]** o promisiune realistă şi uşor de respectat este: Dacă există n utilizatori conectaţi în sistem cât timp tu lucrezi, vei primi aproximativ l/n din puterea de calcul. In mod similar, pe un sistem cu un singur utilizator pe care rulează n procese, toate fiind egale, fiecare ar trebui să beneficieze de l/n din ciclurile procesorului.

Pentru a respecta această promisiune, sistemul trebuie să ţină evidenţa capacităţii de calcul consumate de fiecare proces de la crearea sa. Se calculează apoi capacitatea la care are dreptul fiecare, mai precis intervalul de la creare de împărţit la *n.* Deoarece capacitatea consumată de fiecare proces este cunoscută, se poate calcula raportul capacităţii consumate la capacitatea la care are dreptul fiecare. Un raport de 0,5 înseamnă că procesul a consumat numai jumătate din capacitatea la care are dreptul, iar un raport de 2,0 înseamnă că un proces a consumat de două ori mai multă capacitate de calcul decât ar fi avut dreptul. Algoritmul decide apoi să ruleze procesul cu cel mai mic raport până când raportul său devene mai mare decât cel al procesului imediat următor.

5.6. PLANIFICAREA ÎN UNIX

UNIX a fost dintotdeauna un sistem multiprogramare si de aceea algoritmul sau de planificare a fost proiectat sa asigure un raspuns bun pentru procesele interactive.

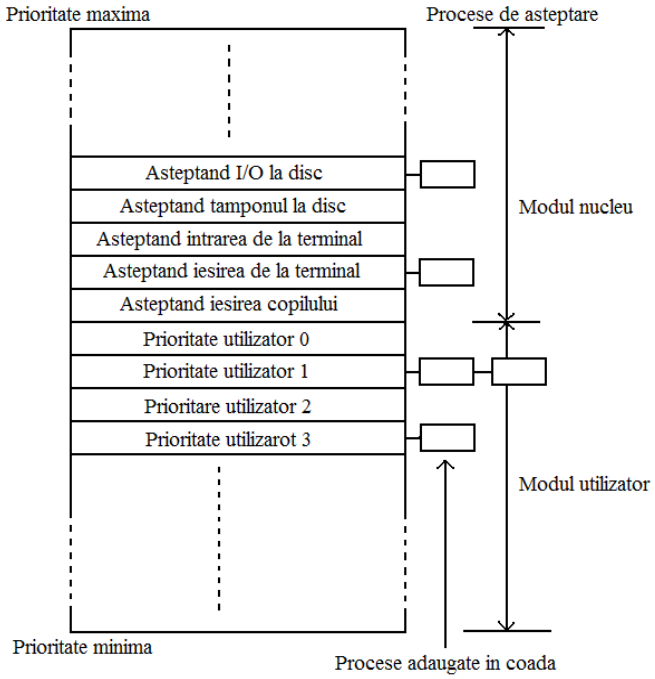
5.6.1. ALGORITMUL DE PLANIFICARE CU DOUĂ NIVELURI

Tocmai de aceea a fost creat un algoritm cu doua niveluri :

1. **algoritmul de nivel jos** care alege un proces care sa se execute din setul de procese din memorie şi care sunt pregătite să ruleze;
2. **algoritmul de nivel inalt** mută procesele între memorie şi disc astfel ca toate procesele să aibă o şansă de a fi în memorie şi de a se executa.

Fiecare versiune de UNIX are un algoritm de planificare de nivel jos diferit.

Figura 5.6.1. Algoritmul de planificare la UNIX.



În Figura 5.6.1. puteţi observa algoritmul de planificare la Unix. Procesele care se execută în mod utilizator au valori pozitive. Procsele care se execută în modul nucleu (făcând apeluri de sistem) au valori negative. Valorile negative au prioritatea cea mai mare şi valorile pozitive mari pe cea mai mică. Doar procesele aflate în memorie şi gata să ruleze sunt situate în cozi.

5.6.2. IMPLEMENTAREA ALGORITMULUI DE PLANIFICARE LA UNIX

Din **[BOVET]** ştim că atunci când planificatorul (de nivel jos) rulează, el caută în cozi începand cu prioritatea cea mai mare până când gaseşte o coadă care este ocupată. Este ales şi pornit primul proces din acea coadă. Îi este permis să ruleze maximum o cuantă de timp, în general 100 msec, sau până se blochează. Dacă un proces işi termină cuanta, este pus la loc la sfârşitul cozii sale şi algoritmul de planificare rulează din nou. Astfel, procesele din acelaşi interval de priorităţi împart UCP-ul folosind algoritmul Round-Robin.

Din **[RUSLING]** şi din **[TANENBAUM]** ştim că o data pe secundă este recalculată prioritatea fiecărui proces corespunzător unei formule care implică trei componente:

Prioritatea = consum_CPU + nice + base

Pe baza noii sale priorităţi, fiecare proces este ataşat cozii corespunzătoare, de obicei împărţind prioritatea cu o constantă pentru a lua numărul cozii.

Consum_CPU reprezintă numărul mediu de tacturi de ceas pe secundă pe care procesorul le-a avut în ultimele câteva secunde. La fiecare tact de ceas, numărătorul de utilizare al UCP din tabela de procese a procesului care se execută este incrementat cu 1. Acest contor va fi adăugat în cele din urmă la prioritate procesului, dându-i o valoare numerică mai mare şi punându-l astfel într-o coadă mai puţin prioritară. Totuşi, UNIX nu permite unui proces utilizarea la infinit a UCP, deci CPU_usage scade cu timpul. Diferite versiuni de UNIX realizează scăderea puţin diferit.

Fiecare proces are o valoare nice asociată lui. Valoarea implicită este 0, dar intervalul permis este în general de la -20 la + 20. Un proces poate seta nice la o valoare în intervalul de la 0 la 20 prin apelul de sistem nice. Doar administratorul de sistem poate cere pentru un serviciu mai bun decât normal.

Când un proces este prins în nucleu pentru a face un apel de sistem, este cu totul posibil ca procesul să se blocheze înainte de a termina apelul de sistem şi a se întoarce în modul utilizator. Când se blochează,

este şters din structura de cozi, deoarece este incapabil să ruleze. Ideea din spatele acestei scheme este de a scoate rapid procesele din nucleu.

5.7. PLANIFICAREA ÎN LINUX

Conform **[TANENBAUM]** planificarea este unul dintre puţinele domenii în care Linux foloseşte un algoritm diferit de UNIX. Tocmai am examinat algoritmul de planificare al UNIX-ului, aşa că ne vom uita acum la algoritmul Linux-ului. Pentru început, firele de execuţie din Linux sunt fire de execuţie în nucleu, aşa că planificarea se bazează pe fire de execuţie, nu pe procese. În scopurile planificării, Linux face distincţie între trei clase de fire de execuţie:

1. FIFO în timp real.
2. Rotaţie (Round-Robin) în timp real .
3. Cu partajarea timpului.

Firele de execuţie FIFO în timp real au cea mai mare prioritate şi nu sunt preemptibile. Firele de execuţie prin rotaţie în timp real sunt la fel ca cele FIFO în timp real, exceptând faptul că sunt preemptibile de ceas. Dacă sunt pregătite mai multe fire de execuţie prin rotaţie în timp real, fiecare este executat cu o cuantă de timp după care se duce la sfârşitul listei de fire de execuţie prin rotaţie în timp real.

Fiecare fir de execuţie are o prioritate de planificare. Valoarea implicită este 20, dar poate fi modificată folosind apelul de sistem nice la valoarea 20-valoare. Din moment ce valoare trebuie să fie în intervalul de la -20 la +19, priorităţile cad mereu în intervalul: 1 ≤ prioritate ≤ 40. Intenţia este de a face calitatea serviciilor proporţională cu prioritatea, cu firele de execuţie cu prioritate mai mare primind un timp de răspuns mai rapid şi o fracţiune mai mare de timp UCP decât firele de execuţie cu prioritate mai mică.

În plus faţă de prioritate, fiecare fir de execuţie are o cuantă asociată. Cuanta reprezintă un număr de tacturi de ceas cât poate să mai ruleze firul de execuţie. Ceasul merge implicit la 100 Hz, deci fiecare tact este 10 msec, ceea ce se numeşte moment. Planificatorul foloseşte prioritatea şi cuanta pentru algoritmul de planificare din Linux.

5.8. PLANIFICAREA ÎN WINDOWS NTOS

Din **[RUSSINOVICH]** ştim că Windows NT nu are un fir de execuţie central pentru planificare. În schimb, atunci când un fir de execuţie nu mai poate rula, firul intră în mod nucleu şi rulează el însuşi planificatorul pentru a vedea care fir de execuţie să comute.

Următoarele condiţii din **[TANENBAUM]** cauzează firul de execuţie curent să execute codul planificatorului:

1. Firul de execuţie se blochează la un semafor, mutex, eveniment, I/O, etc.
2. Semnalizează un obiect.
3. Cota de rulare a firului de execuţie a expirat.

În cazul 1, firul de execuţie se află deja în modul nucleu pentru a îndeplini operaţia asupra dispecerului sau obiectului de I/O. Nu există nici o posibilitate de a continua, aşa că trebuie să işi salveze propriul context, să ruleze codul planificatorului pentru a-şi alege succesorul şi să încarce contextul firului de execuţie pentru a-l porni.

Şi în cazul 2, firul de execuţie rulează în modul nucleu. În orice caz, după semnalizarea unui obiect, el poate să continue deoarece semnalizarea unui obiect nu se blochează niciodată. Totuşi, firul de execuţie trebuie să ruleze planificatorul pentru a vedea dacă acţiunea sa nu a avut ca rezultat eliberarea unui fir de execuţie cu o prioritate mai mare şi care poate acum să ruleze. Dacă s-a întâmplat aşa, va apare o comutare a firului de execuţie deoarece Windows 2000 este complet preemptiv **[RUSSINOVICH].**

Planificatorul este şi el apelat în două cazuri:

1. Se termină o operaţie de I/O.
2. Expiră o aşteptare bine stabilită.

Am ajuns acum la algoritmul efectiv de planificare. API-ul Win32 furnizează două moduri în care procesele pot influenţa algoritmul de planificare. Aceste moduri determină algoritmul în mare. În primul rând este apelul SetPriorityClass care setează clasa de prioritate a tuturor firelor de execuţie din procesul apelant. Valorile permise sunt: timp real, mare, peste normal, normal, sub normal şi inactiv.

În al doilea rând este apelul SetThreadPriority care setează prioritatea relativă a unui fir de execuţie (posibil, dar nu neapărat, a firului de execuţie apelant) în comparaţie cu celelalte fire de execuţie din procesul său. Valorile permise sunt: de timp critic, cel mai mare, peste normal, normal, sub normal, cel mai mic şi inactiv. Cu şase clase de procese şi şapte clase de fire de execuţie, un fir de execuţie poate avea una dintre cele 42 de combinaţii. Aceasta reprezintă intrarea algoritmului de planificare.

Algoritmul de planificare funcţionează în felul următor. Sistemul are 32 de priorităţi, numerotate de la 0 la 31. Cele 42 de combinaţii sunt puse în corespondenţă cu cele 32 de clase de priorităţi conform tabelului 5.8. Numărul din tabel determină prioritatea de bază a firului de execuţie. În plus, fiecare fir de execuţie are o prioritate curentă, care poate fi mai mare (dar nu mai mică) decât prioritatea de bază.

Pentru folosirea priorităţilor la planificare, sistemul menţine un şir cu 32 de înregistrări ce corespund priorităţilor de la 0 la 31 derivate din tabelul 5.8. Fiecare înregistrare din şir, indică către începutul unei liste de fire de execuţie pregătite, având prioritatea corespunzătoare. Algoritmul de planificare de bază constă din parcurgerea şirului de la prioritatea 31 la prioritatea 0.

Stim din **[TANENBAUM]** că atunci când se găseşte o fantă ocupată, firul din capul listei este selectat să ruleze pentru o cuantă de timp. Dacă cuanta expiră, firul de execuţie va fi plasat la sfârşitul listei la nivelul său de prioritate iar firul de execuţie de la inceputul listei va fi următorul ales. Cu alte cuvinte, atunci când există mai multe fire de execuţie pregătite la cel mai înalt nivel de prioritate, ele rulează conform Round Robin pentru o cuantă fiecare. Dacă nu există nici un fir de execuţie pregătit, este rulat firul de execuţie inactiv.

		Clase de priorităţi ale proceselor Win32					
Priorităţile firelor de execuţie Win32		Timp real	Mare	Peste normal	Normal	Sub normal	Inactiv
	Timp critic	31	15	15	15	15	15
	Cel mai mare	26	15	12	10	8	6
	Peste normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Sub normal	23	12	9	7	5	3
	Cel mai mic	22	11	8	6	4	2
	Inactiv	16	1	1	1	1	1

Tabelul 5.8

6. EMULAREA PROGRAMELOR MS-DOS PE WINDOWS (*Casandra Mihai, 443 A*)

Emularea programelor mostenite de la sistemul de operare MS-DOS a fost un aspect foarte important al sistemului de operare Windows pe 32 de biti si anume asigurarea compatibilitatii inapoi a programelor deja existente. Aceasta compatibilitate ridica totusi cateva probleme cum ar fi stabilitatea si securitatea sistemului de operare functie de implementare.

Sistemul de operare Windows pe 32 de biti a aparut in doua variante diferite: platforma Win9x si platforma WinNT. Varianta 9x a fost orientata mai mult spre utilizator si de aceea arhitectura lui este mai simpla, ceea ce il face vulnerabil la anumite atacuri ale unor programe malicioase. Emularea programelor MS-DOS pe platforma 9x a fost facuta in aceeasi maniera: asigurarea maxima a compatibilitatii (in limita posibilitatilor)

fara se pune prea mult problema securitatii care este practic irelevanta in cazul unui sistem simplu al unui utilizator obisnuit. Pe de alta parte platforma NT a fost initial gandita special pt servere deci securitatea este vitala, chiar cu pretul reducerii compatibilitatii si a functionalitatii programelor emulate.

6.1. EMULAREA PE PLATFORMA WINDOWS 9x

Pentru a descrie mai bine procesul de emulare al programelor MS-DOS pe Win9x este necesara o prezentare initiala a arhitecturii a acestei platforme. Sistemul de operare Win95 a fost gandit asemenea lui Win3.1 ca un add-on peste MS-DOS. Nucleul sistemului este pe 32 de biti dar sunt permise si drivere vechi pe 16biti mostenite de la MS-DOS. Pentru a asigura aceasta compatibilitate procesorul este automat trecut din protected mode inapoi in real mode si invers functie de ce bucata de cod ce trebuie executata. Din acesta cauza sistemul de operare isi permite sa execute programele MS-DOS in forma lor nativa adica initial rulate in real mode, emuland functiile de baza ale DOS-ului prin intermediul intreruperii 21h. Deci cand un program MS-DOS este executat pe Win9x procesorul este trecut in real mode si astfel programul preia integral controlul. Daca programul intampina probleme si corupe starea procesorului sau blocheaza rutina de tratare a intreruperii prin intermediul careia Windows-ul preia controlul programul poate sa destabilizeze sistemul de operare sau mai rau sa injecteze cod in programe aflate in executie, sa modifice date, practic sa faca absolut orice pentru ca preia controlul si trebuie sa-l dea inapoi. In cazul in care programul MS-DOS emulat trece el procesorul in protected mode atunci mecanismul este acelasi doar ca sistemul de operare trebuie sa salveze/refaca toate descriptoarele initializate de programul respectiv.
Avantajul acestei implementari este ca programul MS-DOS are direct acces la hardware: poate sa faca input/output pe porturi dupa cum doreste, accesul la memorie este nelimitat. Acestea sunt pe de o parte avantaje dar si dezavantaje. Unele programe pot sa corupa memoria si sa blocheze sistemul de operare accidental pentru ca atunci cand au fost ele proiectate nu s-a luat in calcul rularea intr-un astfel de mediu. Un alt avantaj minor este viteza de executie ceva mai mare fata de implementarea de pe platforma WinNT.

6.2. EMULAREA PE PLATFORMA WINDOWS NT

Platforma WinNT a fost gandita initial exclusiv pentru servere si a fost proiectata dupa principiul conform caruia securitatea si stabilitatea sistemului de operare si a programelor sunt cele mai importante. Din aceasta cauza s-a folosit o alta metoda pentru a emula programe MS-DOS pe platforma WinNT decat cea de pe Win9x. In primul rand sistemele WinNT contin exclusiv componente pe 32 de biti, deci nu suporta drivere pe 16 biti mostenite de la MS-DOS. Procesorul niciodata nu este scos din protected mode pentru a nu da niciodata ocazia unui program sa corupa memoria sau sa preia controlul si sa nu-l mai dea inapoi. Solutia aleasa este tot cea folosita de programele comerciale VMWare si Microsoft Virtual PC care emuleaza calculatoare virtuale.
Ideea de baza este ca tot codul care ar trebui sa ruleze in real mode si in ring3 se executa tot in ring 3 fara sa il afecteze teoretic cu nimic iar daca un program MS-DOS ar incerca sa bage procesorul in protected mode se provoaca o exceptie iar codul care ar fi plasat de acesta in ring0 este de fapt pus in ring1. Astfel codul “crede” ca este in ring0 pentru ca are de fapt mai multe privilegii decat cel din ring3 dar nu are de fapt acces la nucleul sistemului de operare care ruleaza in ring0. Ruland in ring1 nu toate instructiunile din ring0 sunt valide dar acestea sunt emulate. De cate ori ar trebui sa se execute o instructiune nepermisa in ring1 se genereaza o exceptie si sistemul de operare o emuleaza pentru a pacali programul ca ruleaza in mediul sau nativ.
Aceasta metoda are un mare avantaj si anume asigura o securitate si o stabilitate net superioara fata de cealalta (mai ales ca platforma Win9x este stiuta pentru numeroasele probleme referitoare la stabilitate provocate in mare parte de drivere implementate prost sau componente mostenite care corup sistemul de operare neintentionat). Dar totusi pretul platit este functionalitatea: accesul la hardware este sistat. Se emuleaza totusi o parte din functii dar programul MS-DOS nu mai poate comunica direct cu hardware-ul ci trece prin HAL si este filtrat tot ce nu ii este permis. Din aceasta cauza foarte multe programe MS-DOS nu mai pot fi rulate pe platforma WinNT pentru ca incearca sa acceseze anumite porturi hardware iar sistemul de operare nu le permite. O alta problema minora este viteza de executie putin mai mica decat la varianta Win9x. Intarzierea este rezultata de generarea exceptiilor si emularea instructiunilor exclusiv rezervate pentru ring0 care se face software. Acesta totusi nu este un impediment pentru aceste instructiuni sunt in medie nesemnificative ca numar fata de restul care nu trebuiesc emulate.

Programul MS-DOS emulat nu apare ca proces pe 32 de biti. In schimb in locul lui apare un NTVDM.exe care tine locul lui iar restul proceselor il vad pe acesta si nu programul real emulat.
Practica a aratat ca pentru emularea unei varietati cat mai mari de aplicatii MS-DOS platforma Win9x este mult mai potrivita pentru ca nu limiteaza programele iar platforma WinNT asigura o securitate care desi nu este totala este totusi de multe ori necesara chiar si pentru sisteme simple, nu neaparat servere. Ideea este ca nu exista un emulator perfect pt programe MS-DOS, singurul mediu in care toate ruleaza neconditionat este mediul lor nativ, sistemul de opeare MS-DOS dar totusi de cativa ani a aparut o solutie de mijloc care inglobeaza principiul celor doua platforme si anume: sistemul de operare sa nu poata fi afectat de programul emulat dar si la randul lui programul sa nu fie limitat atat de mult incat sa nu poata sa mai ruleze deloc. Solutia este orice program tip masina-virtuala ca VMWare sau Microsoft Virtual PC: principiul de baza este acelasi ca la emularea pe WinNT doar ca problema cu accesul la hardware este rezolvata prin emularea unui calculator virtual, deci a hardware-ului. Programul poate rula nestingherit pentru ca el crede ca dialogheaza cu un hardware real, chiar daca acesta este virtual iar sistemul de operare gazda isi pastreaza securitatea si stabilitatea. Executarea programelor MS-DOS in masini virtuale nu se face direct ci tot prin intermediul unei versiuni reale de MS-DOS, pe scurt se genereaza un calculator virtual in care se instaleaza direct MS-DOS si apoi programele sunt executate in aceast calculator virtual iar legaturile catre alte programe care ruleaza pe calculatorul fizic sau catre alte retele/calculatoare/device-uri o asigura masina virtuala functie de cum este configurata.
Acest mod de emulare pe procesoare care nu suporta instructiuni de virtualizare si anume un nivel de privilegii –1, care sa fie mai important decat ring0 nu prezinta siguranta deplina. De exemplu un program malicios poate sa ajunga sa preia controlul asupra sistemului de operare gazda, iata si un scenariu posibil: sa presupunem ca sistemul de operare gazda este un Windows XP pe 32 de biti, un program rulat fie direct in NTVDM (deci emulat direct de sistemul de operare) sau intr-o masina virtuala poate sa incerce sa treaca procesorul in protected mode dar codul care el ar fi vrut sa fie in ring0 de fapt ruleaza in ring1. Acest fapt protejeaza nucleul sistemului de operare de codul programului din ring1 la acces direct dar nu si aplicatiile mai putin importante care ruleaza in ring3 si care sunt ale sistemului de operare gazda. Codul din ring1 are acces direct la orice segment din ring3 deci poate injecta fara nici o problema cod in absolut orice proces care ruleaza atata timp cat e ring3. La randul sau codul injectat atunci cand este executat poate (daca drepturile ii permit si anume sa ruleze pe un cont de administrator) sa inregistreze un driver scris pe hdd chiar de el insusi iar cand driverul respectiv a fost pornit acesta ruleaza in ring0 deci concluzia e ca desi e o solutie greu de implementat injectarea indirecta de cod din ring1 in ring0 este posibila dar evident depinde de sistemul de operare gazda, deci acest mod de emulare nu este absolut sigur.
Emularea programelor MS-DOS si a programelor pentru Win3.1 a fost abandonata in varianta pe 64 de biti a Windows. Motivul nu este imposibilitatea tehnica de implementare ci faptul ca sistemul de operare MS-DOS si Win3.1 sunt invechite, nu mai sunt de mult folosite pe scara larga (cu mult sub 1% in lume) deci nu mai este rationala pierderea timpului pentru rezolvarea unei probleme care nu mai exista. Totusi programele MS-DOS pot fi executate si pe Win64 cu ajutorul unor programe gen VMWare.
Emularea programelor MS-DOS mai poate fi facuta si in alta maniera: emularea totala software a setului de instructiuni al procesorului. Aceste solutii ofera protectie totala pentru sistemul de operare gazda pentru ca nu se executa direct codul care poate fi malicios ci este emulat software un procesor care il executa. Marele dezavantaj al acestei metode este ca prin emularea totala se pierde enorm de mult timp. Una dintre cele mai bune solutii de acest tip este programul. Pentru a face o comparatie din punctul de vedere al performantei alegem ca exemplu primul joc cu engine cu adevarat 3D: Descent, facut de Interplay si Parallax Software in 1994. Capabilitatile engine-ului 3D sunt echivalente cu cele oferite de Direct3D din DirectX 6 cel mult: obiecte 3D oricat de complexe, texturi, lumini, alpha blending, reflexii. Engine-ul face randarea exclusiv software iar jocul in sine este multithreaded chiar daca randarea se face pe un singur fir de executie. Daca acest joc este emulat complet atunci pe un Athlon64 3000 abia este la limita de 20-25fps, deci este abia acceptabil pe cand daca este emulat prin una din celelalte metode (Win9x sau WinNT) merge impecabil pentru ca el a fost proiectat sa mearga bine pe 486 care au ajuns la maxim 100MHz (486DX4) dar el tot merge mai bine si pe 486DX@33MHz.. Concluzia este ca emularea totala nu este o solutie acceptabila decat in cazuri extreme in care varianta cealalta este absolut imposibil de folosit pentru ca se iroseste enorm de mult timp de procesare (chiar daca implementarea se foloseste de o eventuala arhitectura hardware multi-core sau multi-cpu) iar acest lucru nu mai este rational.

7. PROCESUL DE PORNIRE A SISTEMULUI DE OPERARE: COMPARAŢIE WINDOWS ŞI LINUX (Despa Valentin, 443A)

7.1 LINUX

Linux-ul este o implementare a conceptului de sistem de operare UNIX. Sistemul de operare UNIX a fost derivat din sistemul de operare "Sys V" (System 5) al corporaţiei AT&T. Procesul de iniţializare este proiectat să controleze pornirea şi oprirea serviciilor de sistem, sau demonilor ("daemons") şi permite diferite configuraţii de startup pe nivele de execuţie ("run levels").

Unele distribuţii Linux, cum ar fi Slackware, folosesc procesul de iniţializare BSD, care a fost dezvoltat la Universitatea Berkeley. Sistemul de operare Sys V foloseşte un set mult mai complex de fişiere de comenzi şi directoare pentru a determina care servicii sunt disponibile pe diferite nivele de execuţie decât procesul de iniţializare BSD.

Primul lucru pe care un calculator trebuie să-l facă este să pornească un program special numit sistem de operare. Treaba sistemului de operare este să ajute alte programe să meargă având grijă de detaliile dezordonate de a controla partea hardware a calculatorului.

Procesul de pornire a sistemului de operare este numit bootare (original acesta era numit bootstrapping). Calculatorul ştie să booteze deoarece intrucţiunile de bootare sunt construite în unul din cipurile sale, cipul BIOS (sau Basic Input/Output System). Cipul BIOS îi spune să citească într-un loc fixat pe hard discul cu numărul cel mai mic (discul de boot) pentru un program special numit încărcător boot (sub Linux încărcătorul de boot este numit LILO sau Grub). **[LINUXONLINE]**

Atunci când calculatorul este pornit, primul sector al discului de iniţializare înregistrarea principală de iniţializare este citită în memorie şi executată. Acest sector conţine un mic program (512 octeţi) care încarcă un program de sine stătător numit boot de pe dispozitivul de pornire de obicei un disc IDE sau SCSI. Încărcătorul de boot este adus în memorie şi pornit. Treaba încărcătorului de boot este să pornească adevăratul sistem de operare.

Programul boot se copiază mai întâi la o adresă mare, fixă, de memorie pentru a elibera memoria de jos pentru sistemul de operare. Odată mutat, boot citeşte directorul rădăcină al dispozitivului de pornire. Pentru a face asta, el trebuie să înţeleagă sistemul de fişiere şi formatul directoarelor. Apoi citeşte nucleul (kernel-ul) sistemului de operare şi sare la el. In acest moment, boot şi-a terminat treaba şi nucleul rulează.

Codul de pornire a nucleului este în limbaj de asamblare şi este foarte dependent de maşină. Munca obişnuită include setarea stivei nucleului, identificarea tipului UCP, calcularea cantităţii de RAM prezente, dezactivarea întreruperii, activarea MMU şi în sfârşit apelarea procedurii main din limbajul C pentru a porni partea principală a sistemului de operare. **[LINUXONLINE]**

Codul C are de asemenea de făcut iniţializări considerabile, dar aceasta este mai mult logic decât fizic şi începe prin alocarea unui tampon de mesaje pentru a ajuta găsirea problemelor de pornire. Pe măsură ce iniţializarea continuă, mesajele despre ce se întâmplă sunt scrise aici, pentru a putea fi scoase la iveală după o ratare a secvenţei de pornire de către un program special de diagnosticare.

Ulterior vor fi alocate structurile de date ale nucleului, majoritatea de dimensiune fixă dar existând câteva dependente de memoria RAM disponibilă. In acest moment sistemul începe autoconfigurarea. Folosind fişiere de configurare care spun ce tipuri de dispozitve I/E ar putea fi prezente, începe să verifice dispozitivele pentru a vedea care sunt prezente. Atunci când un dispozitiv răspunde el este adăugat într-o tabelă de dispozitive ataşate. Dacă un dispozitiv nu răspunde atunci este presupus absent şi este ignorat.

După ce tabela este completă, trebuie localizate driverele de dispozitive. Este interesant de menţionat ca Linux-ul poate încărca drivere dinamic.

Blocul de boot se află întotdeauna la aceeaşi locaţie – track 0, cilindru 0, capul 0 al device-ului de sistem de pe care se face bootarea. Acest bloc de boot conţine un program numit loader, în cazul Linux-ului este LILO (Linux LOader) sau Grub, care se ocupă efectiv de bootarea sistemului de operare. Aceste loadere din Linux permit, în cazul unei configuraţii multi-boot (mai multe sisteme de operare pe acelaşi computer), selectarea sistemului de operare care să boot-eze. LILO şi Grub se instalează ori în MBR (Master Boot Record), ori în primul sector de pe partiţia activă.

In continuare vom face referire la LILO ca loader pentru sistemul de operare. Acesta se instalează de obicei în sectorul de boot, numit şi MBR. Dacă utilizatorul alege să booteze Linux, LILO încearcă să încarce kernel-ul sistemului de operare. În continuare sunt prezentaţi paşii pe care îi urmează LILO pentru a încărca sistemul de operare. În cazul configuraţiilor multi-boot, LILO permite utilizatorului selectarea sistemului de operare pe care să-l încarce. Setările pentru LILO se află în /etc/LILO.conf. Administratorii de sistem folosesc acest fişier pentru o configurare în detaliu a loader-ului. Aici se pot stabili manual ce sisteme de operare sunt instalate, precum şi modul de încărcare pentru fiecare în parte. Dacă pe un computer există instalat numai Linux-ul, se poate configura LILO să încarce direct kernel-ul şi să sară peste pasul de selecţie al sistemului de operare. Kernelul Linux-ului este instalat comprimat şi conţine un mic program care-l decomprimă. Imediat după pasul 1, are loc decomprimarea kernel-ului şi începe procesul de încărcare al acestuia. Dacă kernel-ul recunoaşte că în sistem există instalată o placă video care suportă moduri text mai speciale, Linux-ul permite utilizatorului să selecteze ce mod text să folosească. Modurile video şi alte opţiuni pot fi specificate ori în timpul recompilării kernel-ului ori prin intermediul lui LILO sau al altui program (rdev de exemplu). **[LINUXONLINE]**

Kernel-ul verifică configuraţia hardware (hard disk, floppy, adaptoare de reţea, etc) şi configurează driverele de sistem. In tot acest timp sunt afişate mesaje pentru utilizator cu toate operaţiile care se execută. Kernel-ul încearcă să monteze sistemul de fişiere şi fişierele de sistem. Locaţia fişierelor de sistem este configurabilă la recompilare, sau folosind alte programe – LILO sau rdev. Tipul fişierelor de sistem este detectat automat. Cele mai folosite tipuri de sistem de fişiere pe Linux sunt ext2 şi ext3. Dacă montarea fişierelor de sistem eşuează, va fi afişat mesajul kernel panic şi sistemul va îngheţa. Fişierele de sistem sunt montate de obicei în modul read – only, pentru a se permite o verificare a acestuia în timpul montării. Nu este indicat să se execute o verificare a fişierelor de sistem dacă acestea au fost montate în modul read – write. **[TANENBAUM]**

După aceşti paşi, kernel-ul porneşte programul init, care devine procesul numărul 1 şi care va porni restul sistemului. INIT este primul proces al Linux-ului şi este părintele tuturor celorlalte procese. Este procesul care rulează prima dată pe orice sistem Linux sau UNIX şi este lansat de kernel la bootare. Acest proces, la rândul lui, încarcă restul sistemului de operare. ID-ul acestui proces este întotdeauna 0. Prima dată, procesul de iniţializare (init) examinează fişierul /etc/inittab pentru a determina ce procese să lanseze în continuare. Acest fişier de configurare specifică ce nivel de execuţie să se lanseze şi descrie procesele ce trebuie rulate pe fiecare nivel. Apoi, procesul init caută prima linie cu acţiunea sysinit (system initialization) şi execută fişierul de comandă identificat în acea linie, în acest caz /etc/rc.d/rc.sysinit. După executarea script-urilor din /etc/rc.d/rc.sysinit, init începe să execute comenzile asociate cu nivelul de execuţie iniţial. Următoarele câteva linii din /etc/inittab sunt specifice diferitelor nivele de execuţie. Fiecare linie rulează ca un singur script (/etc/rc.d/rc), care ia un număr între 1 şi 6 ca argument pentru specificarea nivelului de execuţie.

Cea mai folosită acţiune pentru aceste nivele de execuţie specifice intrărilor din /etc/inittab este wait, ceea ce înseamnă că procesul init execută fişierul de comandă pentru un nivel de execuţie specific şi apoi aşteaptă ca acel nivel de execuţie să se termine.

Din **[LINUXONLINE]** am aflat că există nişte comenzile care sunt definite în intrările de iniţializare a sistemului din /etc/inittab sunt executate numai odată şi numai de către procesul init, de fiecare dată când bootează sistemul de operare. În general, aceste scripturi rulează ca o succesiune de comenzi care realizează următoarele:

1) Determină dacă sistemul face parte dintr-o reţea, în funcţie de conţinutul fişierului /etc/sysconfig/network.
2) Montează /proc, sistem de fişiere folosit intern de Linux pentru a urmări starea diverselor procese din sistem.
3) Setează ceasul sistemului în funcţie de setările din BIOS precum şi realizează alte setări (setarea timpului, setarea zonei), stabilite şi configurate pe parcursul instalării Linux-ului.
4) Porneşte memoria virtuală a sistemului, activând şi montând partiţia swap, identificată în fişierul /etc/fstab (File System table).
5) Setează numele de identificare (host name) pentru reţea şi mecanismul de autentificare al sistemului (system wide authentication), cum ar fi NIS (the Network Information Service,), NIS + (o versiune îmbunătăţită de NIS) şi aşa mai departe.
6) Verifică sistemul de fişiere al root-ului şi dacă nu sunt probleme, îl montează.
7) Verifică celelalte sisteme de fişiere identificate în fişierul /etc/fstab.
8) Identifică, dacă este cazul, rutine speciale care sunt folosite de sistemul de operare pentru a recunoaşte hardware-ul instalat, pentru a configura device-uri plug and play existente şi pentru a activa alte device-uri primare, cum ar fi placa de sunet de exemplu.
9) Verifică starea disk device-urilor specializate, cum ar fi de exemplu discurile RAID (Redundant Array of Inexpensive Disks).
10) Montează toate sistemele de fişiere identificate în fişierul /etc/fstab.
11) Execută alte task-uri specifice de sistem.

Directorul /etc/rc.d/init.d conţine toate comenzile care pornesc şi opresc serviciile care sunt asociate cu toate nivelele de execuţie. Toate fişierele de comenzi din directorul /etc/rc.d/init.d au un scurt nume care descrie serviciul cu care sunt asociate. De exemplu, fişierul /etc/rc.d/init.d/amd porneşte şi opreşte demonul automount, care montează gazda NFS şi device-uri ori de câte ori este nevoie.

După ce procesul init a executat toate comenzile, fişierele şi scipturile, ultimele câteva procese pe care le startează sunt procesele /sbin/mingetty care afişează banerul şi mesajul de login al distribuţiei pe care o aveţi instalată. Sistemul este încărcat şi pregătit pentru ca utilizatorul să facă login.

Când cineva introduce numele de conectare, getty se termină executând /sbin/login, programul de conectare. Login cere apoi o parolă, o criptează şi o verifică cu parola aflată în /etc/passwd. Dacă e corectă, login se înlocuieşte cu interpretorul utilizatorului care va astepta comenzi. Dacă parola se dovedeşte a fi incorectă, pur şi simplu se va cere alt nume de utilizator urmat de o altă parolă.

Nivelele de execuţie reprezintă modul în care operează computerul. Ele sunt definite de un set de servicii disponibile într-un sistem la orice timp dat de rulare.Nivelele de execuţie reprezintă modalităţi diferite pe care sistemul de operare Linux le foloseşte pentru a fi disponibil dumneavoastră ca utilizator sau ca administrator.

Din **[LINUXONLINE]** ştim că nivelul de execuţie multi-utilizator face (în mod transparent) disponibile serviciile pe care dumneavoastră aşteptaţi să vă fie puse la dispoziţie în momentul în care folosiţi Linux într-o reţea. In continuare sunt prezentate cele 6 nivele de execuţie:

0: Halt (Opreşte toate procesele şi execută shutdown pentru sistemul de operare.)
1: Cunoscut sub numele de "Single user mode,", sistemul rulează în acest caz un set redus de daemoni. Sistemul de fişiere al root-ului este montat read-only. Acest nivel de execuţie este folosit când celelalte nivele de execuţie eşuează în timpul procesului de boot-are.
2: Pe acest nivel rulează cele mai multe servicii, cu excepţia serviciilor de reţea (httpd, nfs, named, etc.). Acest nivel de execuţie este folositor pentru debug-ul serviciilor de reţea, menţinând sistemul de fişiere shared.
3: Mod multi-utilizator complet, cu suport pentru reţea.
4: Nefolosit în marea majoritate a distribuţiilor. Pe Slackware, acest nivel de execuţie 4 este echivalent cu nivelul de execuţie 3, cu logon grafic activat.
5: Mod multi-utilizator complet, cu suport pentru reţea şi mod grafic.

6: Reboot. Termină toate procesele care rulează şi reboot-ează sistemul la nivelul iniţial de execuţie.

Modificarea nivelelor de execuţie. Cea mai folosită facilitate a procesului init, şi poate cea mai confuză, este abilitatea de a muta de pe un nivel de execuţie pe altul. Sistemul bootează pe un nivel de execuţie specificat în /etc/inittab, sau într-un nivel de execuţie specificat la prompt-ul LILO. Pentru a schimba nivelul de execuţie, folosiţi comanda init. De exemplu, pentru a schimba nivelul de execuţie la 3, se va folosi init 3. Aceasta opreşte majoritatea serviciilor şi aduce sistemul în mod multi-utilizator cu suport pentru reţea. Atenţie, pentru că în acest moment se pot închide forţat daemoni care sunt folosiţi în acel moment.

Directoarele nivelelor de execuţie. Fiecare nivel de execuţie are un director cu legături simbolice care pointează scripturilor corespunzătoare din directorul init.d. Numele legăturii simbolice găsite în aceste directoare este semnificant. El specifică care serviciu trebuie oprit, pornit şi când. Legăturile care încep cu “S” sunt programate să pornească ori de câte ori sistemul într-un alt nivel de execuţie. Pe lângă aceasta, fiecare legătură simbolică are un număr la începutul numelui.

Când sistemul de operare schimbă nivelul de execuţie, init compară lista cu procesele terminate (legături care încep cu “K”) din directorul nivelului curent de execuţie, cu lista proceselor catre trebuie startate, cele care încep cu “S”, aflate în directorul destinaţie. Atfel se determină care daemoni trebuie porniţi sau opriţi. Schimbarea nivelului curent de execuţie. Pentru a schimba nivelul de execuţie curent de xemplu pe nivelul 3, se editează fişierul /etc/inittab într-un editor de texte, următoarea linie: id:3:initdefault:

Bootarea către un nivel de execuţie alternativ. La promptul LILO, se scrie nivelul de execuţie dorit înainte de boot-area sistemului de operare. Astfel, pentru a boota pe nivelul de execuţie 3, se va scrie: linux 3. Eliminarea unui serviciu dintr-un nivel de execuţie. Pentru a dezactiva un serviciu de pe un nivel de execuţie, puteţi simplu şterge sau modifica legătura sa simbolică din directorul nivelului de execuţie de care aparţine. Adăugarea de servicii unui nivel de execuţie. Pentru a adăuga un serviciu unui nivel de execuţie, este nevoie să se creeze o legătură simbolică care să pointeze către scripturile de servicii din directorul init.d. În momentul în care se crează legătura, aveţi grijă să-i asignaţi un număr astfel încât serviciul să fie pornit la timpul potrivit.

7.2 WINDOWS (NT, 2000, XP, Server 2003, VISTA)

Vom trata în continuare pornirea Windows NT, 2000 şi XP. Procesul de pornire creează un proces iniţial care porneşte sistemul. Procesul constă în citirea de pe primul sector al primului disc şi saltul către acesta. Se citeşte tabela de partiţii pentru a determina ce partiţie conţine sistemul de operare ce poate fi pornit.

Din **[MINASI]** ştim că atunci când se găseşte partiţia sistemului de operare, se citeşte primul sector, denumit sectorul de pornire (boot sector) şi sare la acesta. Programul din boot sector citeste directorul rădăcină al partiţiei în căutarea unui fişier ntldr pe care îl scrie în memorie şi îl execută. Ntldr încarcă Windows-ul. Există însă câteva versiuni ale sectorului de pornire, depinzând dacă partiţia este formatată ca FAT 16, FAT 32 sau NTFS.

Fişierul NTLDR este compus din două parţi. Prima este modulul de StartUp şi osloader.exe, ambele stocate în acelaşi fişier. Când Ntldr este în memorie şi controlul este deţinut de StartUp, procesorul operează în modul real. Modulul StartUp are rolul de a trece procesorul în modul protejat, pentru a facilita accesul la memorie pe 32 biţi, creând astfel tabela descriptorului de întreruperi, tabela de descriptori globali, lista de pagini. **[MINASI]**

Osloader.exe include functionalitate de bază pentru a accesa discuri de tip IDE formatate pentru sisteme de fisiere NTFS sau FAT, sau pentru CDFS, ETFS sau UDFS în sisteme de operare mai noi. Discurile sunt accesate prin BIOS, prin rutine ARC pe sisteme ARC sau prin reţea prin protocolul TFTP. Toate apelurile în BIOS se fac prin modul 8086 virtual. Pentru un disc SCSI se va încărca un fişier adiţional, Ntbootdd.sys. Mai departe Ntldr citeşte un fişier Boot.ini, care este singura informaţie de configurare ce nu se află în registru. Aceasta va lista toate versiunile de hal.dll şi ntoskrnl.exe disponibile pentru pornirea din această partiţie. Fişierul conţine de asemenea mulţi parametrii, cum ar fi câte UCP-uri şi câtă memorie să folosească şi la ce rată să seteze ceasul de timp real. Dacă boot.ini lipseşte, boot loaderul va încerca să găsească informaţie din directorul standard de instalare. Pentru sistemele Windows NT, va încerca să pornească din C:\WINNT. Pentru Windows XP and 2003, va porni din C:\WINDOWS. **[RUSS_SOLOMON]**

Din **[TANENBAUM]** ştim că la acest moment, ecranul este şters şi în Windows 2000 sau versiuni ulterioare ce suporta hibernarea, se va căuta fişierul de hibernare, hiberfil.sys. Dacă fişierul este găsit şi există un set de memorie activă în el, conţinutul fişierului este încărcat în memorie şi controlul este

transferat către kernelul Windows-ului. Daca boot.ini conţine mai multe sisteme de operare, un meniu va fi afişat, dând posibilitatea utilizatorului de a alege ce sistem de operare să încarce. Dacă un sistem de operare non-NT este selectat, atunci Ntldr încarcă sectorul de boot ascociat listat in boot.ini (implicit bootsect.dos) şi îi predă controlul. Dacă un sistem de tip NT este selectat, Ntldr rulează ntdetect.com care adună informaţii de bază despre partea hardware a computerului aşa cum raportează BIOS-ul. La acest punct Ntldr şterge ecranul şi afişează bara de progres textuală. In Windows 2000 se mai afişează şi textul "Starting Windows...". Dacă utilizatorul apară tasta F8 în timpul acestei faze, se va afişa meniul de boot avansat care conţine mai multe moduri de boot, printre care şi Safe mode, cu ultima configuraţie cunoscută a fi bună.

Dacă este încărcată versiune x64 de Windows, procesorul este trecut acum în Long mode, permiţând adresare pe 64 de biţi.

Atunci ntldr va selecta şi va încărca fişierele hal.dll (Hardware Abstraction Layer) şi ntoskrnl.exe precum şi bootvid.dll, driverul video standard pentru scrierea pe ecran în timpul procesului de pornire. Ntldr citeşte apoi registrul pentru a determina driverele ce trebuie încărcate pentru finalizarea pornirii (cum ar fi driverele de mouse şi tastatură, drivere pentru controlarea diferitelor chip-uri de pe placa de bază). După ce citeşte toate aceste drivere, dă controlul către ntoskrnl.exe. **[RUSS_SOLOMON]**

După pornire, sistemul de operare începe să facă unele iniţializări generale şi apoi apelează componentele executivului să îşi facă propria lor iniţializare. De exemplu, managerul de obiecte îşi pregateşte spaţiul de nume pentru a permite altor componente să îl apeleze pentru insearea obiectelor lor în spaţiul de nume.

Multe componente fac lucruri specifice, strâns legate de funcţia lor. Managerul de memorie setează tabelele iniţiale cu pagini iar managerul de ,,plug and play” găseşte ce dispozitive de I/E sunt prezente şi încarcă driverele lor. Sunt implicaţi zeci de paşi, timp în care bara de progres afişată pe ecran creşte în lungime după cum paşii sunt terminaţi. **[MINASI]**

Ultimul pas este crearea primului proces utilizator, managerul de sesiune (the session manager), smss.exe. Odată ce acest proces a început să ruleze pornirea este terminată. Managerul de sesiune este un proces nativ, efectuând apeluri de sistem reale şi nu foloseste subsitemul de mediu win32, care nici măcar nu rulează încă. Una din primele sale îndatoriri este să îl (csrss.exe) pornească. Treaba sa constă în introducerea mai multor obiecte în spaţiul de adrese al managerului de obiecte, crearea de fişiere de paginare suplimentare dacă acestea sunt necesare şi deschiderea unor DLL-uri importante ce trebuie să fie tot timpul deschise. Ulterior crează demonul de autentificare, winlogon.exe. **[RUSS_SOLOMON]**

winlogon.exe este responsabil pentru toate autentificarile utilizatorilor. Dialogul de autentificare efectiv este ţinut de un program separat din msgina.dll pentru a face posibil ca terţe părţi să înlocuiască autentificarea standard cu identificarea amprentelor sau orice altceva în afară de nume şi parolă. După o autentificare reuşită, winlogon.exe ia profilul utilizatorului din registru şi determină ce interfaţă cu utilizatorul să ruleze. Spatiul de lucru Windows standard este doar explorer.exe cu câteva opţiuni setate.

WINDOWS VISTA

La Windows Vista procesul pe scurt ar fi următorul: Bios > Master Boot Record > Sector Boot > Windows Boot Manager > Citire din BCD > Căutare fişier hibernare > Start winload.exe > Start ntoskernl.exe > Start smss.exe > Start winlogon.exe > Start servicii şi interfaţă login.

Din **[TECHNET]** ştim că secvenţa de boot la Windows Vista e uşor diferită de versiunile anterioare de Windows care folosesc kerneul NT. Când calculatorul este pornit, fie BIOS-ul sau EFI-ul (Extensible Firmware Interface) sunt încărcate. Pentru cazul BIOS, MBR-ul discului este accesat, urmat de sectorul de boot care încarcă celelalte blocuri. Acest proces de boot este comun tuturor sistemelor de operare.

Pentru Windows Vista, sectorul de boot încarcă Windows Boot Manager (Bootmgr.) care accesează BCD (Boot Configuration Data). BCD-ul este o bază de date independentă de firmware şi înlocuieşte boot.ini folosit anterior de Ntldr. BCD-ul este stocat întru-un fişier localizat fie pe partiţia EFI (pe sistemele cu EFI) sau în \Boot\Bcd pe sistemele de tip IBM-PC. BCD-ul poate fi modificat folosit un tool de tip command-line (bcdedit.exe) sau folosind Windows Management Instrumentation.

BCD-ul conţine intrări de meniu care înclud opţiuni de a porni Vista prin invocarea winload.exe, opţiuni de a reporni Vista din hibernare folosind winresume.exe, opţiuni de a porni o versiune anterioară de Windows NT prin invocarea Ntldr-ului acestuia, opţiuni de a încărca şi executa Volume Boot Record.

Winload.exe este încărcătorul de boot a sistemului de operare. Este lansat de Windows Boot Manager pentru a încărca kernelul (ntoskrnl.exe). **[TECHNET]**

C. CONCLUZII GENERALE

Putem spune că procesele dar mai ales firele de execuţie oferă nişte concepte simplificatoare pentru a ascunde defectele întreruperilor. Pentru multe aplicaţii software este folositor să folosim fire de execuţie multiple ştiind că fiecare fir este planificat independent şi fiecare are stiva sa proprie. Mai mult am analizat în această lucrare şi implementarea firelor de execuţie în mod utilizator, nucleu sau hibrid ajungând la concluzia că cea mai avantajoasă situaţie este cea hibridă pentru că oferă beneficile ambelor metode anterioare. Am analizat şi primitivele de comunicare între procese cum ar fi semafoarele şi monitoarele care ajută ca 2 procese să nu ajungă niciodată în regiunea lor critică în acelaşi timp.

Pentru fiecare sistem de operare în parte Windows şi Linux am prezentat cele mai importante implementări ale conceptelor teoretice cum ar fi: implementarea proceselor şi a firelor de execuţie, planificarea, sistemul de pornire, apelurile de sistem folosite pentru gestiunea proceselor. Chiar dacă multe lucruri diferă la ele din punct de vedere al proceselor şi firelor de execuţie ambele sisteme permit un bun sistem de planificare, o bună sincronizare a firelor de execuţie fiind două sisteme de operare moderne din toate punctele de vedere. Există şi o parte mică la sfârşit despre sistemul de pornire al Windows Vista uşor diferit faţă de versiunile precedente care relevă anumite schimbări în simplitate faţă alte sisteme.

D. BIBLIOGRAFIE SELECTIVĂ

[STĂNCESCU] Ştefan Stăncescu, **Note de curs: Sisteme de operare**, 2007

[CRK] **CurriculumResourceKit-CRK Units** - set de resurse teoretice oferit de Microsoft în programul Windows Research Kernel;

[TANENBAUM_87] Andrew S. Tanenbaum, **Operating Sistems. Design and Implementation**, Prentice Hall, 1987

[TANENBAUM] Andrew S. Tanenbaum, **Sisteme de operare moderne**, Byblos, 2004;

[BOVET] Daniel P. Bovet şi Marko Cesati, **Understanding the Linux Kernel**, O'Reilly, 2000, disponibilă online la http://www.sti.uniurb.it/acquaviva/ulk.pdf ,

[RUSLING] David A Rusling, **The Linux Kernel**, 1999, carte disponibilă gratuit la adresa: http://tldp.org/LDP/tlk/tlk.html

[RUSSINOVICH] Russinovich, M. E. and Solomon, D.A. Microsoft Windows Internals, 4th Edition, Microsoft Press, 2005 (face parte din Windows Research Kernel)

[YOLINUX] Tutorial YOLINUX pentru gestionarea proceselor, disponibil la adresa http://www.yolinux.com/TUTORIALS/

[MSDN] Librărie online oferită de Microsoft în scopul studierii Windows OS(aici folosit pentru procese şi fire de execuţie), disponibil la http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/processes_and_threads.asp

[MSDN FORUMS] http://forums.microsoft.com/msdn/default.aspx?siteid=1

[WIKIPEDIA] http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

[EVANJONES] http://evanjones.ca/software/threading.html

[RUSS_SOLOMON] Russinovich, Mark; David Solomon (2005). "Startup and Shutdown", Microsoft Windows Internals, 4th edition, Microsoft Press.

[MINASI] Mark Minasi, John Enck. Troubleshooting NT Boot Failures. Administrator's Survival Guide: System Management and Security. Windows IT Library

[TECHNET] http://technet.microsoft.com/en-us/library/bb457123.aspx

[LINUXONLINE] http://www.myl.ro şi http://www.linux.ro