Procese

Andrei Vlad (432A)

Dobre Alina Alexandra (432A)

Frincu Ioana Alexandra (432)

# Cuprins

[Cuprins](#h.8cehxkue2gez)

[0. Nota introductiva](#h.gai4nsb52bww)

[1. Introducere](#h.pkfpg37vs00z)

[1.1 Sisteme de operare](#h.blsjpy3gd2d3)

[Linux](#h.cyj2p1x2yz7n)

[Windows](#h.n7z4nmnl7kf8)

[1.2 Procese](#h.kn2mbhehi645)

[1.3 Fire](#h.kdj3r0y54bjz)

[2. Gestiunea Proceselor si apelurile de sistem de gestiune a proceselor](#h.1pkys7ii8o8a)

[2.1 Blocul de control al procesului (PCB)[Tanenbaum]](#h.hb40p5rxcile)

[2.1.1 Identificarea Procesului](#h.jcitp4bp3c0p)

[2.1.2 Informație de stare a procesului](#h.1cixo9pzqkko)

[2.1.2.1 Registri vizibili utilizatorului](#h.4yifs4kqgc6f)

[2.1.2.2 Regiștrii de control și stare](#h.oapc0q4fkc4h)

[2.1.2.3 Stiva Pointerilor](#h.c9xmhsv0fa97)

[2.1.2.4 Programarea și starea informației](#h.p3zii83wpg9t)

[2.1.2.5 Structura Datelor](#h.hduryw97dmtr)

[2.1.2.6 Comunicarea Interproces](#h.aaudc6fs4zc1)

[2.1.2.7 Privilegiile Procesului](#h.t8jtvl3mo8um)

[2.1.2.8 Managementul Memoriei](#h.44y2w9jh1ksz)

[2.1.2.9 Resurse de Proprietate și Utilizare](#h.k6n5chwpuhe2)

[2.2.3 Moduri de execuție](#h.89woj56zzd4h)

[2.2.4 Crearea Procesului](#h.pqonvdyw7eqe)

[2.2.5 Schimbarea stării procesului](#h.zclqj140x58f)

[2.2.6 Executarea sistemului de operare](#h.ohdm6muadwdl)

[2.3 Realizarea unei comunicații semafor[ALP]](#h.xfju6frvtf0i)

[2.3.1 Alocarea si Dezalocare](#h.m2f2uy81bag7)

[2.3.2 Inițializarea Semafoarelor](#h.digjyalu7d9)

[3. Apeluri de Sistem](#h.1boy2blr3ore)

[3.1 Ce este un apel de sistem?[LJ]](#h.96ciqxqb40y5)

[3.2 Comunicarea cu nucleul](#h.n92a5wr7dgre)

[3.3 Application Programing Interface (API)](#h.s802lfb90o4s)

[3.4 Portable Operating System Interface (POSIX)](#h.sizrcyh7cy4r)

[3.5 C Library (Biblioteca C)](#h.vmehnx9t4wq2)

[4. Comparație între implementarea proceselor și firelor de execuție la Windows și LINUX](#h.qranyoqq140q)

[4.1 Generalități](#h.nk6086vx0fli)

[4.2 Implementarea proceselor și firele de execuție în Windows](#h.nk6086vx0fli)

[5. Algoritmul de planificare in LINUX si Windows si diferenta fata de modelul folosit in UNIX](#h.nk6086vx0fli)

[5.1 Problema algoritmilor de planificare](#h.4lionbtk3tjp)

[5.1.1 Algoritmii de planificare și întreruperile de ceas](#h.exf7h350hakh)

[5.1.2 Algoritmi de planificare în funcție de tipul de aplicație folosit](#h.d06kxikejqs2)

[5.2 Concepte pentru proiectarea planificării în sistemele interactive](#h.wfh17jee039a)

[5.2.1 Planificarea Round Robin](#h.u0daf8ou4nfo)

[5.2.2 Planificarea bazată pe priorități](#h.nvpd3aox2mlf)

[5.2.3 Shortest Process Next](#h.u4s0z2lbdl79)

[5.2.4 Planificarea garantată](#h.el03mxdkiuqj)

[5.3 Planificarea în UNIX](#h.iudj2nfotlzr)

[5.3.1 Implementarea algoritmului de planificare](#h.lgudf1eonnm5)

[5.4 Planificarea în LINUX](#h.kiqiv1cjukhu)

[5.5 Planificarea în Windows NTOS](#h.ockfhwv886so)

[6. Procesul de pornire al sistemelor de operare](#h.i6djj8awkprw)

[6.1 Introducere/Notiuni generale](#h.277kezt1jcd2)

[6.2 Procesul de startup WINDOWS](#h.198wivbv0fy5)

[6.2.1 Windows bazat pe DOS](#h.8zh7gcatyd79)

[6.2.1.1 Windows 1.x/2.x](#h.lew45hmj6ofc)

[6.2.1.2 Windows 3.x/9x](#h.5i4malxc4bgf)

[6.2.3 Windows NT](#h.tp423gdd3lgv)

[6.2.4 Windows 7](#h.exs1a4tmbtcf)

[6.3 Bootarea sistemului de operare Linux](#h.nuyteg4gitxy)

[6.4 Concluzii](#h.conn8ttltqqq)

[7. Bibliografie](#h.v0c06rt3zzbv)

# 

# 

# 0. Nota introductiva

Impartirea pe capitole a fost făcută în modul următor:

1. Introducere - Frincu Ioana Alexandra (432A)
2. Gestiunea Proceselor și apelurile de sistem de gestiune a proceselor - Andrei Vlad(432A)
3. [Apeluri de Sistem](#h.1boy2blr3ore) - Andrei Vlad (432A)
4. Comparație între implementarea proceselor și firelor de execuţie la Windows și LINUX - Dobre Alina Alexandra (432A)
5. Algoritmul de planificare în LINUX și Windows și diferența față de modelul folosit în UNIX Dobre Alina Alexandra (432A)
6. Procesul de pornire a sistemului de operare - Frincu Ioana Alexandra (432A)

# 

# 

# 1. Introducere

## 1.1 Sisteme de operare

Un **sistem de operare** (SO) reprezinta un software care manageriaza (coordoneaza) resursele unui calculator atat din punct de vedere hardware cat si software si ofera servicii comune pentru programele care ruleaza pe acesta. El este o componenta esentiala in sistemul software al unui computer. Programele/Aplicatiile solicita un astfel de sistem pentru a functiona.

Un sistem de operare creeaza posibilitatea de a:

* servi unei multitudini de scopuri
* interactiona in moduri variate
* tine pasul cu nevoile in schimbare ale utilizatorului

La nivelul cel mai de baza, un sistem de operare are doua mari responsabilitati:

1. **Manageriaza resursele hardware si software ale sistemului** (ex. procesor, memorie, spatiu pe disk s.a). SO se asigura ca fiecare aplicatie primeste resursele necesare, in acelasi timp oferind resurse si altor aplicatii intr-un mod cat mai “echitabil” pentru buna functionare a computerului;
2. **Ofera un mod stabil si consecvent pentru aplicatii de a interactiona cu resursele hardware** fara a stii totul despre acestea. Un API (application program interface) permite unui developer sa scrie o apicatie portabila si eficienta care poate functiona si pe alt computer de acelasi tip, chiar daca anumite resurse difera (capacitate de stocare, capacitatea memoriei).

Componentele sistemului de operare sunt:

* Nucleul (Kernel) - contine programele necesare gestionarii resurselor calculatorului si controlarii activitatii echipamentelor si programelor;
* Interfata (Shell) - defineste modul in care utilizatorul interactioneaza cu SO.

Exista mai multe tipuri de sisteme de operare, dintre care este necesar sa le amintim pe urmatoarele[WIKI1]:

1. Single/Multi - tasking
2. Single/Multi - user
3. Distribuite
4. Templated
5. Embedded
6. Real Time

**Sisteme de operare Single si Multi - tasking** sunt sisteme care pot rula fie un singur proces, fie un numar mai mare de procese in acelasi timp pe un calculator. Pentru multi-tasking se utilizeaza conceptul de Time Sharing care presupune divizarea timpului de procesare intre mai multe procese diferite care sunt intrerupte repetat de catre un subsistem de programare a taskurilor, oferindu-le acestora “felii de timp” necesare ducerii la bun sfarsit a proceselor.[HSF]

**Sistemele de operare single-user** nu au o modalitate de deosebire a cestor utilizatori, dar pot permite mai multor programe sa ruleze in acelasi timp. Un sistem multi-user extinde conceptul de baza al multi- taskingului cu facilitati de a identifica procesee si resursele care apartin mai multor useri si permite unui numar de utilizatori sa interactioneze cu sistemul in acelasi timp. [HSF]

**Sistemele distribuite** manageriaza un grup de computere distincte si le face sa para un singur computer. Dezvoltarea computerelor in retea care sa fie conectate si sa comunice unul cu celalalt au oferit cresterea sistemelor distribuite. Cand computerele dintr-un grup lucreaza impreuna, formeaza un astfel de sistem intrucat calculul e facut de mai multe masini. [HSF]

**Sistemele templated** se refera la crearea unei masini virtuale ca sistem de operare “oaspete” si apoi il salveaza ca “unealta” (tool) pentru rularea masinilor virtuale. Tehnica este folosita atat in virtualizare cat si managementul computingului in cloud si este foarte comuna in lucrul cu servere. [HSF] [WIKI1]

**Sistemele embedded** sunt create pentru a fi folosite in sisteme de computere embedded. Sunt gandite astfel incat sa opereze pe masini de capacitate mica (ex. PDA). Ele opereaza cu un numar limitat de resurse. SUnt foarte compacte si extrem de eficiente (ex. Windows CE). [HSF]

Un **sistem Real-time (RTOS)** este un sistem de operare care garanteaza ca va procesa evenimente si date intr-o perioada de timp anume. El poate fi single/multi-tasking, dar cand este multitasking foloseste algoritmi de programare pentru a crea o natura determinista a comportamentului masinii. Un sistem bazat pe evenimente schimba intre ele sarcinile bazat pe prioritatea fiecareia sau pe evenimente externe, pe cand sistemele time-sharing bazeaza sarcinile pe intreruperi de ceas. [HSF] [WIKI1]

Funcţiile unui sistem de operare sunt:

* Să asigure comunicarea între utilizator şi S.O.;
* Să asigure controlul execuţiei programelor;
* Să asigure alocarea eficientă a resurselor fizice ale sistemului;
* Să faciliteze utilizatorilor dezvoltarea de noi aplicaţii;
* Să faciliteze gestionarea eficientă a datelor;
* Să asigure securitatea sistemului.

Cele mai importante (respectiv raspandite) sisteme de operare la momentul actual sunt; Microsoft Windows, Mac OS X si Linux.



*Fig. 1: Cele trei sisteme de operare: Windows, Mac OSX si Linux*

### Linux

Linux (sau GNU/Linux) este un **sistem de operare freeware si opensource**, deci sursele sale pot fi modificate in mod liber, folosite si distribuite de oricine. Numele “Linux” vine de la Kernelul Linux, inceput in 1991 de Linus Torvalds. Utilitarele si bibliotecile sistemului vin din sistemul de operare GNU. Linuxul este predominant folosit pe servere, dar este portabil pe toate platformele, de la computere, la supercomputere, sisteme de gaming si telefoane mobile sau routere.

In Linux structura de directoare poate arata asa:

* / (hdd)
* /mnt/floppy (discheta)
* /mnt/cdrom (dvd-rom)
* /mnt/hdd (hdd 2)

Programele din Linux sunt "compilate" de catre dezvoltatorul distributiei in "pachete" (echivalent al setup.exe din windows) specifice fiecareia. Pentru Debian/Ubuntu/Gentoo avem pachete .deb, pentru RedHat/Fedora/CentOS/Mandriva/OpenSuse avem pachete .rpm si pentru Slackware avem pachete .tgz/.txz. De asemenea aceste programe sunt oferite in mod gratuit in asa zisele "repository" specifice fiecarei distributii. Fiecare distributie are propriul manager de programe, un software care se ocupa cu instalarea/dezinstalrea/actualizarea tuturor programelor instalate in acel sistem de operare (exceptie o fac programele "compilate din surse"), asigurarea dependentelor necesare (un anumit program, pentru a functiona, este dependent de un alt program, acest manager asigura satisfacerea acestei cerinte).

### Windows

Microsoft Windows (sau doar Windows) reprezinta o meta-familie de sisteme de operare cu interfata grafica dezvoltate, promovate si vandute de Microsoft. Consista din cateva familii de sisteme de operare care deservesc cate un sector anume al industriei computerelor. Familiitle active Windows includ: Windows NT, Windows Embedded si Windows Phone; acestea, la randul lor au sub-familii (ex: Windows Server). Familiile Windows defuncte includ Windows9x si Windows Mobile.

In Windows structura de directoare este urmatoarea:

* A:\> (discheta)
* C:\> (hdd 1)
* D:\> (dvd-rom)
* E:\> (hdd 2)

## 1.2 Procese

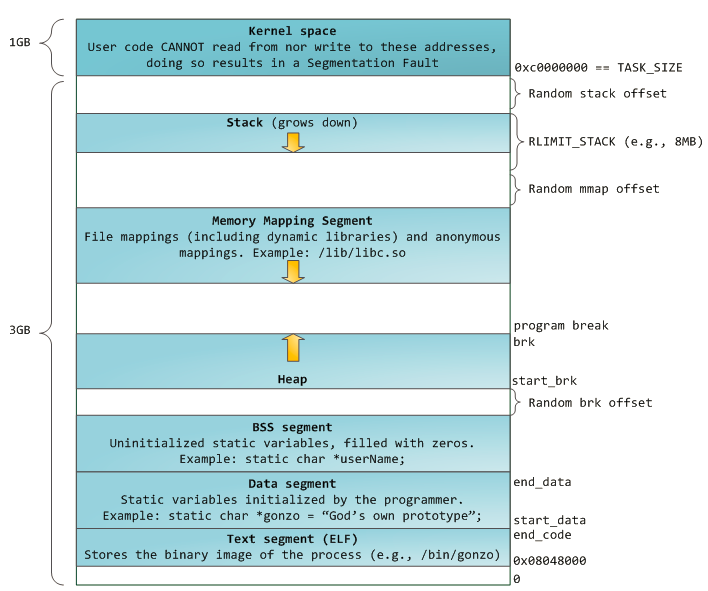
Un *proces* (*task*) este asociat unui program lansat în executie. Termenul se mai foloseste câteodatã ca un sinonim pentru *task*. În majoritatea sistemelor de operare existã o corespondentã unul-la-unul (biunivocã) între *task* si *program (aplicatie).* Existã însã sisteme de operare care permit ca un program sã fie divizat în mai multe *task*-uri, aceste sisteme de operare numindu-se *multithreading*.

Un proces este mereu creat cu un singur fir de executie initial, numit initial thread. Acesta ofera compatibilitatea necesara cu procese single thread din trecut. Stiva firului de executie initial este si stiva procesului.

Procesele sunt formate din:

* un program in execuție;
* o zona de date;
* *stiva* - aici sunt stocate variabilele automate, impreuna cu informatia salvata ori de cate ori se apeleaza o functie; de cate ori o functie este apelata, se salveaza pe stiva valoarea de retur si anumite informatii de control (spre exemplu unele registre); [CS]
* *heap* - zona in care memoria se aloca dinamic (prin intermediul unor apeluri de forma *malloc, calloc, realloc*)
* un PC (program counter).

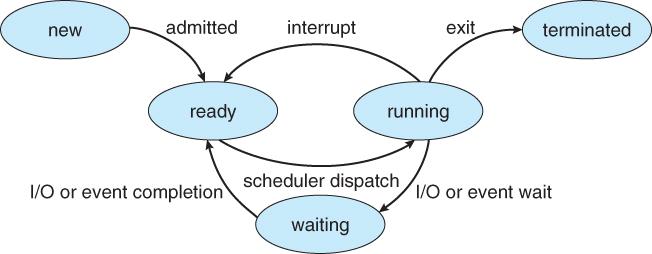
Procesorul poate executa un singur proces/fir la un moment dat. Pentru a oferi utilizatorului senzatia de multiprocesare sistemul de operare manageriaza executia proceselor in asa maniera incat da senzatia de pseudoparalelism. Conceptul de pseudoparalelism implica punerea la dispozitie a procesorului (CPU) pentru o perioada de timp cate unui proces. Cum viteza de procesare este suficient de mare, intr-un timp scurt procesele sunt executate in functie de prioritate si se da impresia de multi-procesare. [IBM1]



*Fig. 2: Layout-ul standard al unui proces in Linux*

Procesele pot fi regasite in 5 stari posibile: [UIC3]

* **New** - Procesul este nou creat.
* **Ready** -Procesul are la dispozitie toate resursele necesare pentru a rula, dar procesorul nu lucreaza cu nici o instructiune a sa in acest moment.
* **Running** - CPU-ul lucreaza cu instructiunile procesului.
* **Waiting** - Procesul nu poate fi rulat deoarece se asteapta ca anumite resurse sa devina disponibile sau ca anumite evenimente sa aiba loc. De exemplu, se poate astepta dupa un input de la tastatura sau pentru un mesaj inter-proces ori ca un proces copil sa fie executat.
* **Terminated -** Procesul a rulat complet.



*Fig. 3: Cele 5 stari posibile ale proceselor*

Procesele pot crea alte procese (procese-copil) prin apeluri de sistem precum fork sau spawn. Procesul care creeaza alt proces este denumit proces parinte. Fiecare proces primeste un identificator de tip integer numit **process identifier** (PID). PID-ul parinte ( PPID ) este de asemenea stocat pentru fiecare proces in parte. [IBM1]

Intr-un sistem UNIX scheduler-ul este numit **sched** si i se atribuie un PID 0. Prima sa actiune la startup este sa porneasca **init** care are PID 1. Init lanseaza mai departe toti daemonii sistemului si user loginurile si devine astfel parintele tuturor proceselor.

Functia de sistem **fork()** este folosita pentru a crea procese. Nu are argumente si returneaza PID-ul unui proces. Scopul lui **fork()** este de a crea procese noi care sunt procesele-copil ale celor apelate. Dupa crearea unui proces ambele vor executa instructiunea urmatoare apelului de sistem ***fork()*** system call. Distingerea proceselor parinte de procesele-copil se poate face prin testarea valorii returnate de functia **fork()**:

* Daca **fork()** returneaza o valoare negativa, atunci creare procesului-copil a esuat;
* **fork()** returneaza zero procesului-copil
* **fork()** returneaza o valoare pozitiva (PID-ul procesului-copil) procesului parinte. PID-ul returnat al procesului e de tipul **pid\_t** definit in **sys/types.h**. Mai mult, un proces poate folosi functia **getpid()** pentru a returna PID-ul acociat unui proces.

Comunicarea intre doua sau mai multe procese se poate face prin pipe-uri sau prin semnale. Pipe-urile permit doar o comunicare unidirectionale. SO-ul permite proceselor sa se conecteze la fiecare din capetele unui pipe fiind posibila atat citirea cat si scrierea de catre mai multe procese la capetele unui pipe. Semnalele sunt receptate de procese si acestea raspund la ele in cateva feluri:

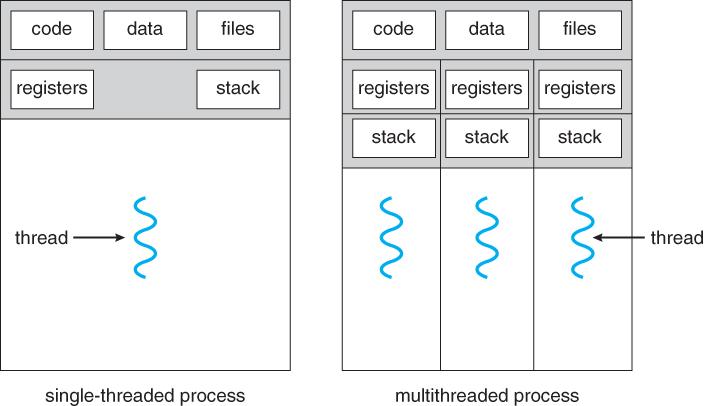
* sa capteze semnalul si sa il trateze corespunzator (signal handler);
* sa ignore semnalul cu pricina;
* sa execute actiunea implicita la primirea unui semnal.

## 

## 

## 1.3 Fire

Un fir de excutie este unitatea de baza a utilizarii CPU si consista dintr-un PC (Program Counter), o stiva, un set de registrii si un thread ID. Firele fac parte din procese, iar un proces poate avea mai multe fire de executie in functie de complexitatea acestuia. Programele rudimentare contineau un singur fir de executie si, implicit un singur ID al acestuia.



*Fig.4: Proces single-thread si multi-thread*

In cazul sistemului de operare Linux, biblioteca standardul de management si sincronizare a firelor de execuție este biblioteca pthreads. In cazul sistemului de operare Windows biblioteca este System.Threads.

Pentru a crea un thread in Linux exista functia pthread\_create, cu apelul:

*pthread\_create (pthread\_t \* thread, pthread\_attr\_t\* attr, void\* (\*start\_routine) (void\* ), void \*arg);*

* 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;

In cadrul sistemului Linux se pot implementa firele de execuție prin intermediul functiei clone(). Aceasta o alternativa la functia fork() (care crea procesul-copil folosind procesul-parinte). Terminarea unui thread se face in Linux apeland functia pthread\_exit() ori prin iesirea din functia start\_routine(). [MTU]

Un thread in Linux se poate termina apeland functia *void pthread\_exit(void \*retval*). Retval este valoarea pe care thread-ul o returneaza la terminare (poate fi introdusa in functia de start a unui alt thread din cadrul aceluiasi proces).luata 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 execuție; in momentul preluarii acestora, resursele lor nu sunt complet dezactivate;
* detached – thread-uri ale caror stari/valori de iesire nu pot fi preluate; dezactiveaza complet resursele proprii in momentul in care se iese din ele.

Pentru a evita ca la iesirea din functia *go()* (Windows) sau *pthread\_exit* (Linux) toate firele sa-si termine executia setului de instructiuni concomitent, este necesar un algoritm de sincronizare. Problema sincronizarii se pune atat la thread-uri, cat si la procese. In cazul in care o multitudine de procese/fire de execuție folosesc resurse comune, rezultatul final al execuției lor poate sa fie instabil intrucat conteaza ordinea in care procesele /firele de execuție returneaza rezultatele executiei insctructiunilor.

Situatiile in care rezultatul execuției unui sistem format din mai multe procese sau fire de

execuție depinde de viteza lor relativa de execuție se numesc **conditii de cursa**. Conditiile de cursa apar atunci cand este solicitat accesul la parti din program care sunt puse la comun. Aceste portiuni care acceseaza sectiuni din program puse la comun se numesc zone/sectiuni critice (critical sections). Trebuie sa ne asiguram ca firele care ruleaza nu executa cod in acelasi timp in zonele critice (excluziune mutuala). [TAN]

O alta metoda de sincronizare este **metoda Semafoarelor**. Semaforul este un tip de date abstract ce poate avea o valoare intreaga ne-negativa si asupra caruia se pot efectua operatiile: init, up si down: [TAN]

* init initializeaza semaforul;
* down realizeaza o operatie de decrementare, daca valoarea este pozitiva. Daca valoarea este nula, procesul se blocheaza;
* up incrementeaza valoarea semaforului, daca nu exista un process blocat pe acel semafor. In caz contrar, valoarea semaforului ramane zero iar unul dintre procese care este blocat va fi deblocat.

# 

# 2. Gestiunea Proceselor si apelurile de sistem de gestiune a proceselor

## 2.1 Blocul de control al procesului (PCB)[Tanenbaum]

### 

Fig. 5:

### 

### 2.1.1 Identificarea Procesului

Identificatorii numerici care pot fi stocați cu Blocul de Control al Procesului includ:

* Identificator de proces;
* Identificator al procesului care a creat procesul (proces părinte);
* Identificatorul de Utilizator;

### 2.1.2 Informație de stare a procesului

#### 2.1.2.1 Registri vizibili utilizatorului

Un registru vizibil utilizatorului este acela care poate fi referit prin intermediul limbajului mașină pe care procesorul îl execută în “user mode”.De obicei sunt intre 8-32 regiștri, dar unele implementări RISC au peste 100.

#### 2.1.2.2 Regiștrii de control și stare

Sunt mulți regiștri ai procesorului care sunt folosiți pentru controlul acestuia.Aceștia includ:

* Contor de program: Conține adresa următoarei instrucțiunii;
* Coduri de stare: Rezultatul celor mai recente operații aritmetice sau logice;
* Informații de stare: Includ întreruperi activate/dezactivate, steaguri, modul de execuție;

#### 2.1.2.3 Stiva Pointerilor

Fiecare proces are una sau mai multe stive de sistem de tipul (LIFO).O stivă este folosită pentru a stoca parametrii si adresele de așteptare pentru procedură si apelurile si sistem.Pointer-ul se introduce în stivă prin partea de sus a acesteia.

#### 2.1.2.4 Programarea și starea informației

Aceste informații sunt necesare sistemului de operare pentru a putea efectuat programarea:

* Starea procesului: Definește disponibilitatea procesului de a fi programat pentru execuție de exemplu:(running, ready, waiting, halted);
* Prioritatea: Unul sau mai multe câmpuri pot fi utilizate pentru a descrie prioritatea programării procesului de exemplu:(default, current);
* Informații referitoare la programare:Acestea vor depinde de algoritmul de programare utilizat.
  + *Exemple: Timpul în care procesorul a fost pus in așteptare si durata de timp scursă de când procesul a fost executat ultima oara.*
* Eveniment: Identitatea evenimentului pentru care procesul a fost pus în așteptare.

#### 2.1.2.5 Structura Datelor

Procesele pot fi legate între ele prin intermediul strcturilor de tip coadă și inel.

*Exemple:Toate procesele care sunt puse în asteptare pot fi legate printr-o structura de tip coadă. Un proces poate prezenta o relație părinte-fiu cu un alt proces.Blocul de control al procesului poate conține pointeri la alte procese pentru a sprijini structurile.*

#### 2.1.2.6 Comunicarea Interproces

Diferite steaguri, semnale și mesaje pot fi asociate prin comunicarea între două procese independente.Unele sau toate aceste informații pot fi menținute în blocul de control al procesului

#### 2.1.2.7 Privilegiile Procesului

Proceselor le sunt acordate privilegii în ceea ce privește memoria care poate fi accesată și tipurile de instrucțiunii care pot fi executate.În plus privilegiile se pot aplica utilizării de utilitățti și servicii de sistem

#### 2.1.2.8 Managementul Memoriei

Această secțiunea poate cuprinde indicii de segment și/sau tabele care descriu memoria virtuală alocată acestui proces.

#### 2.1.2.9 Resurse de Proprietate și Utilizare

Resursele controlate de proces pot fi indicate precum fișierele deschise.Un istoric de utilizare al procesorului sau a altor resurse pot fi de asemenea incluse.Aceste informații pot fi necesare planificatorului.

### 2.2.3 Moduri de execuție

Exista doua moduri de executie:

1. Modul Utilizator
   * Cel mai puțin privilegiat mod;
   * Programele de utilizator se execută de obicei în acest mod.
2. Mod Sistem, modul de control sau modul kernel
   * Cel mai privilegiat mod;
   * Nucleul sistemului de operare.

### 2.2.4 Crearea Procesului

Pasii prin care se creeaza un proces sunt:

* Se atribuie un indentificator de proces unic;
* Alocarea spațiului pentru proces;
* Inițializarea Blocului de Control al procesului;
* Configurarea legăturii apropiate;
* Crearea sau extinderea altor structurii de date;

### 2.2.5 Schimbarea stării procesului

Aceasta se face prin:

* Proces de ieșire.
  + Salvează contextul, inclusiv contorul de program și alte registre;
  + Actualizează blocul de control al procesului care rulează;
  + Mută blocul de control al procesului în zona potrivită;
* Selectați un alt procedeu de execuție.
* Proces de intrare
  + Actualizează blocul de control al procesului selectat;
  + Actualizează structurile de date ale managemntului memoriei;

### 2.2.6 Executarea sistemului de operare

* Non-process Kernel
  + Executarea nucleului în afara oricărui proces;
  + Codul sistemului de operare este executat ca o entitate separată care funcționează în mod privilegiat;
* Execuția în cazul proceselor de utilizare
  + Software-ul sistemului de operare în contextul unui proces de utilizator;
  + Procesul execută în modul privilegiat când execută codul unui sistem de operare;

## 2.3 Realizarea unei comunicații semafor[ALP]

Semafoarele pot fi descrise ca find contoare folosite pentru a controla accesul resurselor partajate de mai multe procese.Cel mai des acestea au rol de mecanism de blocare astfel încât blocheaza accesul unui proces la resurse cât timp un alt proces folosește acele resurse.Semaforul este considerat cel mai dificil concept dintre cele trei care fac parte din SIstemul V IPC.



Fig. 6:

### 2.3.1 Alocarea si Dezalocare

Apelurile “semget” si “semctl” alocă si respectiv dezalocă semafoare, care este același lucru cu “shmgt” si “shmectl” pentru partajarea memoriei. Semaforul continuă să existe chiar și după ce toate procesele care îl folosesc s-au terminat.Ultimul proces ce folosește semafoarul trebuie șters pentru a fi siguri că sistemul de operare nu oprește folosirea acestora. Pentru a putea face acest lucru se apelează “semctl” ce are ca argumente: ID-ul semaforului, numărul acestora si IPC\_RMID. ID-ul procesului apelant trebuie să se potriveasca cu cel al semaforului alocator.Spre deosebire de segmente, ștergerea unui semafor cauzează sistemului de operare Linux dealocare imediată.

*Exemplu:*

*#include <sys/ipc.h>*

*#include <sys/sem.h>*

*#include <sys/types.h>*

*/\* Trebuie definită propria unitate semun.\*/*

*union semun {*

*int val;*

*struct semid\_ds \*buf;*

*unsigned short int \*array;*

*struct seminfo \*\_\_buf;*

*};*

*/\*Se obține ID-ul semaforului într-un fromat binar și se alocă dacă este necesar. \*/*

*int binary\_semaphore\_allocation (key\_t key, int sem\_flags)*

*{*

*return semget (key, 1, sem\_flags);*

*}*

*/\* Dezalocarea binară a semaforului.Toți utilizatorii trebuie să fi terminat deja utilizarea. \*/*

*int binary\_semaphore\_deallocate (int semid)*

*{*

*union semun ignored\_argument;*

*return semctl (semid, 1, IPC\_RMID, ignored\_argument);*

*}*

### 2.3.2 Inițializarea Semafoarelor

Alocarea și inițializarea semafoarelor sunt două operații total diferite.Pentru a inițializa un semafor se folosește comanda “semctl”, cu zero al doilea argument și “SETAL” al treilea argument.Pentru al patrulea argument trebuie să creezi o unitate obiect “semun” si un vector de valori mici, neînsemnate .Fiecare valoare este folosită pentru a inițializa un semafor.

*Exemplu:*

*/\*Inițializarea binară a unui semafor cu valoare 1. \*/*

*int binary\_semaphore\_initialize (int semid)*

*{*

*union semun argument;*

*unsigned short values[1];*

*values[0] = 1;*

*argument.array = values;*

*return semctl (semid, 0, SETALL, argument);*

*}*

**2.3.3 Operațiile Wait and Post**

Fiecare semafor are o valoare pozitivă și suportă operațiile “wait” si “post”.Apelul de sistem “semop” implementează ambele operații.Primul parametru specifică id-ul semaforului.Al doilea parametru este un vector ce conține elementele strcturii “sembuf” ce specifică operațiile care se doresc efectuate.Al treilea parametru este lungimea vectorului.

Câmpurile structurii “sembuf” sunt :

1. sem\_num : reprezintă numărul semaforului din set la care se efectuează operația;
2. sem\_op : este un întreg care specifică operația semaforului;
3. sem\_op>0 : numărul se adaugă la valoarea semaforului;
4. sem\_op<0 : numărul se scade din valoarea semaforului;
5. sem\_op=0 : se blocheză operația până când valoarea semaforului devine 0;
6. sem\_flg : valoarea steagului;

|  |  |
| --- | --- |
| *Exemplul 1* | *Exemplul 2* |
| *Wait*  */\*Blochează până când valoarea semaforului*  *devine pozitivă, apoi decrementează cu 1 . \*/*  *int binary\_semaphore\_wait (int semid)*  *{*  *struct sembuf operations[1];*  */\* Use the first (and only) semaphore. \*/*  *operations[0].sem\_num = 0;*  */\* Decrement by 1. \*/*  *operations[0].sem\_op = -1;*  */\* Permit undo’ing. \*/*  *operations[0].sem\_flg = SEM\_UNDO;*  *return semop (semid, operations, 1);}* | *Post*  */\*Incrementează valoarea semaforului cu 1. \*/*  *int binary\_semaphore\_post (int semid)*  *{*  *struct sembuf operations[1];*  */\* Use the first (and only) semaphore. \*/*  *operations[0].sem\_num = 0;*  */\* Increment by 1. \*/*  *operations[0].sem\_op = 1;*  */\* Permit undo’ing. \*/*  *operations[0].sem\_flg = SEM\_UNDO;*  *return semop (semid, operations, 1);*  *}* |

# 3. Apeluri de Sistem

## 3.1 Ce este un apel de sistem?[LJ]

Un apel de sistem este un apel către nucleu pentru a executa o funcție specifică care controlează un dispozitiv sau execută o instrucțiune privilegiată. Modul în care apelurile de sistem sunt tratate se face de către procesor. De obicei, un apel de sistem are ca rezultat o întrerupere sau o excepție;în apel există o cerere de a executa ceva special. De exemplu, portul serial poate fi programat să afirme o întrerupere atunci când un personaj a ajuns, în loc să voteze pentru el. În acest fel procesorul poate fi folosit pentru procese și serviciile portului serial, numai atunci când este necesar.

In orice sistem de operare modern nucleul oferă un set de interfețe prin care procesele care rulează în “user-space” pot interacționa cu sistemul. Aceste interfețe dau aplicației acces controlat la hardware, un mecanism cu care să creeze noi procese si să comunice cu cele existente, precum si capacitatea de a solicita resursele altui sistem de operare.

## **3.2** **Comunicarea cu nucleul**

Apelurile de sistem oferă un strat între procesele hardware și “user-space”. Acest strat servește la trei scopuri principale:

* În primul rând acesta oferă o interfață hardware abstractă pentru “user-space”. La citirea sau scrierea dintr-un fișier, de exemplu, cererile nu sunt vizate de tipul sistemului de fișiere pe care se afla acesta.
* În al doilea rând apelurile de sistem asigură securitatea sistemului si stabilitatea acestuia.Nucleul care acționează ca un intermediar între resursele de sistem și “user-space”, poate avea permisiunea utilizatorului de a arbitra. Acest arbitraj împiedică aplicațiile să folosească incorect hard-ul și furtul resurselor altor procese.
* În al treila rând, doar “Managementul proceselor” este considerat singurul strat comun între “user-space” și restul sistemului.Dacă cererile au fost libere să acceseze resusele de sistem fară ca nucleul să ia la cunostință, este imposibil să pună în aplicare multitasking-ul și memoria virtuală și cu siguranță imposibil să facă acest lucru cu stabilitatea si securitatea sistemului.

În Linux apelurile de sistem sunt singurele mijloace prin care utilizatorul interacționează cu nucleul, acestea sunt singurele puncte de intare legală în nucleu cu excepția capcanelor. Interesant la Linux este faptul că implementează mult mai puține apeluri de sistem fată de celelalte sisteme de operare. Urmatorul capitol abordează implementarea apelurilor de sistem în LINUX.

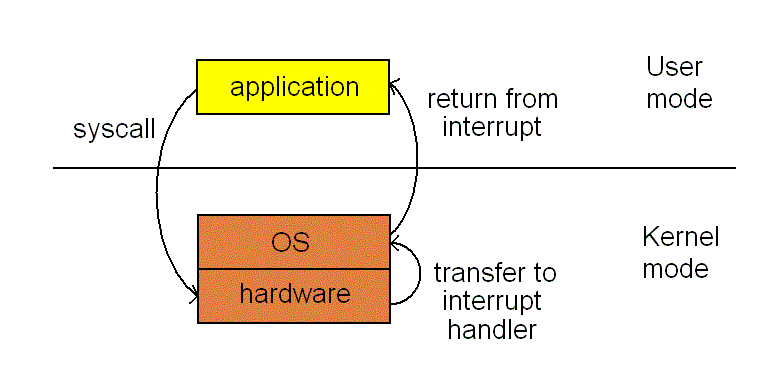


Fig. 7:

## 3.3 Application Programing Interface (API)

De obicei aplicatiile sunt programate pe o interfata de programare specifica acestora (API “Application Programming Interface”) si sunt implementate in spatiul utilizatorului nu direct in apelurile de sistem. Acest lucru este important, deoarece este nevoie de o corelatie directa intre interfetele folosite de aplicatii si interfata reala oferita de nucleu. Un API defineste un set de interfete de programare folosite de aplicatii.Acestea pot fi implementate ca un apel de sistem.

Acelasi API poate exista pe mai multe sisteme si ofera aceeasi interfata pentru aplicatii in timp ce punerea in aplicarea a acestuia poate diferi mult de la un sistem la altu.

## 3.4 Portable Operating System Interface (POSIX)

Una dintre cele mai frecvente interfețe de programare a aplicațiilor în UNIX este bazată pe standardul POSIX.Tehnic POSIX este compus dintr-o serie de standarde din IEEE2 care urmăresc să ofere un sistem de operare portabil bazat pe UNIX.Linux de altfel este compatibil cu acest standard.

POSIX este un exemplu excelent al relației dintre API-uri si apelurile de sistem.Pe marea majoritate a sistemelor UNIX, POSIX sunt definite ca apeluri API ce au o corelație puternică cu apelurile de sistem. Intra-adevar, standardul POSIX a fost creat să reasambleze interfețele prevăzute de sistemul UNIX anterior.Pe de altă parte și sistemul Microsoft Windows oferă biblioteci compatibile POSIX.

## 3.5 C Library (Biblioteca C)

Interfața apelului de sistem în LINUX ca și în majoritatea sistemelor UNIX este prevăzut ca făcând parte din biblioteca C. Biblioteca C implementează API-ul în sistemele UNIX inclusiv pe cel standard și al sistemului de apel. Această bibliotecă este folosită de majoritatea programelor și prin prisma naturii sale limbajul C poate fi foarte ușor de adoptat și de alte limbaje de programare.

Interfața apelului de sistem în LINUX ca și în majoritatea sistemelor UNIX este prevăzut ca făcând parte din biblioteca C. Biblioteca C implementează API-ul în sistemele UNIX inclusiv pe cel standard și al sistemului de apel. Cu alte cuvinte apelurile de sistem în UNIX există pentru a oferi funcții specifice într-un mod abstract. Modul în care este utilizată functia nu este treaba nucleului.

# 

# 

# 4. Comparație între implementarea proceselor și firelor de execuție la Windows și LINUX

## 4.1 Generalități

Sistemele de operare moderne se bazează pe anumite concepte de bază cum ar fi: procesele, memoria, fişierele. Nu este necesar ca un utilizator normal al unui sistem de operare să cunoască aceste concepte, în schimb un programator ar trebui să le cunoască pentru a putea dezvolta aplicaţii cât mai eficiente, care să utilizează corect toate resursele sistemului de calcul, de preferat acest resurse trebuie să fie portabile de pe un sistem de operare pe altul. Cel mai simplu mod de a defini un proces este: *Un proces este un program aflat în curs de execuţie.*

Un sistem de operare modern are sarcina de a crea şi termină procese, suspendă şi continuă procese, sincroniza procese prin metode specifice, asigura comunicarea între procese, rezolvă situaţiile de impas șamd. Pentru realizarea aplicaţiilor, programatorul se va folosii de aceste funcţii ale sistemului de operare. Problema care apare este că sistemele de operare existente abordează diferit aceste funcţii, fiecare dintre aceste abordări având unele avantaje, dar şi unele dezavantaje.

Cele mai utilizate sisteme de operare până în prezent sunt cele bazate pe UNIX şi diferitele versiuni ale sistemelor Windows. Părerile despre aceste două feluri de sisteme diferă, unii sunt de părere că sistemele bazate pe UNIX au un avantaj faţă de sistemele Windows pentru că oferă o flexibilitate, stabilitate şi securitate mai mare, însă pe de altă parte alţii pretind că sistemele Windows au un avantaj deoarece sunt mai uşor de folosit. De-a lungul anilor diferenţele între aceste două sisteme s-au micşorat considerabil.

## 4.2 Implementarea proceselor și firele de execuție în Windows

La cel mai înalt nivel de abstracție,un proces Windows constă în urmatoarele elemente:

* Un spațiu virtual de adrese privat(memoria pe care o are procesul la dispoziție);
* Un program executabil care conține instrucțiunile programului și care va fi încărcat în spațiul virtual de adrese;
* Lista de legături spre resurse de sistem,porturi de comunicație,fișiere;
* Un context de securitate reprezentat de un nume de utilizator,o listă de grupuri din care face parte utilizatorul;
* Un identificator de proces unic;
* Cel puțin un fir de execuție.

Un proces Windows conține și un *pointer catre procesul părinte*(procesul din care a fost lansat). Acest pointer poate fi și nul.

Firul de execuție este componenta fundamentală a unui proces. Un proces fără fire de execuție nu se poate executa. Un fir de execuție din Windows poate fi descris de componentele următoare [RUS]:

* Starea curentă a procesorului descrisă de valorile din registri;
* Doua stive fiecare petru executia intr-unul din cele doua moduri : utilizator sau nucleu;
* Un identificator al firului de execuție;
* Uneori firele de execuție au și un context de securitate.

Formatul contextului unui fir de execuție este dependent de arhitectura mașinii pe care rulează sistemul de operare. Cu ajutorul metodei *GetThreadContext* se pot extrage informații din contextul firului de execuție.

Fibrele sunt niște fire de execuție care pot fi lansate în execuție manual prin apelul metodei *SwitchToFiber*. Spre deosebire de fibre, firele de execuție rulează automat cu ajutorul unui mecanism de coordonare bazat pe priorități.

**Crearea unui proces în Windows**

Un proces este creat atunci când un alt proces apelează metoda *CreateProcess* din biblioteca kernel32.dll

Crearea procesului se face parcurgând următorii pași :

* Fișierul executabil este analizat. Dacă este un fișier executabil Windows pe 16 biți sau MS-DOS atunci este creat un mediu special de execuție pentru acesta, dacă fișierul este un executabil Windows pe 32 de biți se verifică registrul pentru a vedea dacă are vreo cerință specială. Operațiile de la acest pas se fac în modul utilizator.
* Se creează un obiect proces gol cu apelul de sistem NtCreateProcess și se adaugă în spațiul de nume al managerului de obiecte. Mai sunt create un obiect nucleu și unul executiv. Managerul de proiecte creează un bloc de control al procesului pentru acest obiect și îl inițializează cu Idul procesului și cu alte câmpuri. Este creat un obiect secțiune pentru a urmări spațiul de adrese al procesului.
* kernel32.dll preia controlul, efectuează apelul de sistem NtCreateThread pentru a crea un fir de execuție inițial. Se creează stivele utilizator și nucleu. Dimensiunea stivei este dată de antetul fișierului executabil.
* kernel32.dll trimite subsistemului de mediu Win32 referințele către proces și către firul de execuție care sunt adăugate în niște tabele.
* Este apelată o procedură de sistem care termină inițializarea iar firul de execuție începe să ruleze.
* O procedură apelată la execuție setează prioritatea firului de execuție. Se repetă pasul pentru toate firele de execuție. Începe rularea codului principal al procesului. [TEN]

**4.3 Implementarea proceselor și firele de execuție în Linux**

În Linux nu există o diferență precisă între procese și fire de execuție. Un proces “părinte” poate crea mai multe procese “copil” care accesează același spațiu de memorie alocat procesului părinte. Atunci când unul dintre copii încearcă să modifice o zonă de memorie a procesului părinte, este creată o copie a acelei zone de memorie pe care va operă în continuare procesul copil, nefiind astfel nevoie să fie creată o copie a întreg spațiului de memorie al procesului părinte pentru fiecare copi l(se folosește mecanismul copy-on-write). Procesele copil pot fi asemănate cu firele de execuție. [LDP]

Pentru gestionarea proceselor în Linux, kernelul alocă fiecărui proces o structură de date numită task\_struct. Rezultă o colecție de structuri de date task\_struct care va fi reprezentată sub două forme:

* sub formă de vector (tabel de căutare) de structuri task\_struct;
* sub formă de listă circulară dublu-înlănțuită de structuri task\_struct.

În reprezentarea sub formă de tabel, fiecărui proces îi este asociat un identificator de proces (PID). Relația dintre PID și indexul unei structuri task\_struct în vector este urmatoarea: Index = ((((PID) >> 8) ^ (PID)) & (PIDHASH\_SZ − 1)), unde PIDHASH\_SZ este dimensiunea tabelului de căutare.

Reprezentarea sub formă de listă circulară dublu-înlănțuită este folosită pentru a stabili ordinea de execuție a proceselor.[LDP] Structura task\_struct este foarte complexă însă câmpurile ei pot fi împărțite în următoarele categorii:

**1. Stări**

Orice proces își schimbă starea în funcție de contextul în care se află. Astfel putem avea urmatoarele patru tipuri de stări :

* *în execuție:* procesul rulează sau este capabil să ruleze dar așteaptă să fie preluat de către microprocesor;
* *în așteptare:* procesul asteaptă un eveniment sau eliberarea unei resurse. Procesele în așteptare pot fi întreruptibile (pot fi oprite cu anumite semnale de oprire) și neîntreruptibile (funcționarea lor este condiționată de partea hardware și nu pot fi întrerupte folosind semnale);
* *oprit:* procesul a primit un semnal de oprire;
* *zombie:* sunt procese oprite dar care încă mai au o structură task\_struct alocată în tabelul de căutare.

**2.** **Informații utile coordonării proceselor**

În funcție de informațiile utile sunt distribuite resursele și prioritățile de execuție ale proceselor.

**3.** **Identificatori**

Identificator de utilizator, identificator de grup, în functie de aceste informații sunt stabilite drepturile de acces ale proceselor la sistemul de fișiere.

**4.** **Comunicare între procese**

Sunt suportate diverse mecanisme de comunicare între procese: semafoare, semnale, cozi de mesaje, țevi sau memorie partajată.

**5.** **Legături**

Toate procesele au un proces părinte. Părintele tuturor proceselor din Linux este procesul init(1). Structura *task\_struct* conține pointeri către părintele procesului respectiv, către alte procese care au același părinte (“frați”), către procesele copil.

**6.** **Cronometre**

Sunt contoare care țin evidența timpului consumat de fiecare proces în parte. În funcție de aceste contoare procesul își poate trimite anumite semnale la anumite momente de timp în execuția sa.

Etapele necesare pentru crearea proceselor:

* La pornirea sistemului în modul nucleu, există un singur proces inițial părinte;
* După ce sistemul a fost inițializat acest proces părinte lansează un fir de execuție (un proces copil) numit *init* după care rămâne inactiv. Structura *task\_struct* alocată procesului părinte este singura care nu este alocată dinamic ci este declarată ca variabilă statică în codul sursă al kernelului (se numește *init\_task*);
* După ce a fost lansat în execuție, procesul init inițializează sistemul (inițializează consola, montează sistemul principal de fișiere) după care, în funcție de continuțul fișierului */etc/inittab* lansează în execuție alte procese;
* Din aceast moment noi procese pot fi create prin clonarea celor deja existente prin apelarea unor proceduri de sistem. [TEN]

# 

# 

# **5. Algoritmul de planificare in LINUX si Windows si diferenta fata de modelul folosit in UNIX**

Datorită faptului că procesele sunt gata de execuție simultan apare frecvent problema concurenței pe sistemele de calcul ce folosesc multiprogramarea. Din cauză că 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 algoritmilor de planificare

Pentru calculatoarele personale există un singur proces în execuţie în majoritatea timpului. Pe de altă parte, calculatoarele au devenit tot mai rapide şi capacitatea UCP nu mai este ca pe vremuri, o resursă limitată. Singura limită pentru programele calculatoarelor este în general viteza de introducere a datelor şi nu viteza de procesare a unităţii centrale de prelucrare. Consecinţa este faptul că planificarea nu contează prea mult pe calculatoarele 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 și planificarea devine foarte importantă.

Paşii care trebuie urmați în cazul unui algoritm de planificare sunt:

* corectitudinea alegerii programului de executat, astfel încât planificatorul să 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;
* alegerea unui proces;
* Unitatea de Gestiune a Memoriei trebuie reactualizată cu harta nouă a memoriei pentru un nou proces. [TEN]

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, o dată în modul nucleu şi încă o dată la ieşire şi după cum se ştie asta mănâncă timp, dar mai ales resurse.

# 

### 5.1.1 Algoritmii de planificare și întreruperile de ceas

Se pot lua decizii de planificare la fiecare întrerupere de ceas sau la un număr fix de întreruperi, acest fapt se poate întâmpla doar dacă un ceas hardware oferă periodic întreruperi [TEN]. Algoritmii de planificare sunt grupaţi ținând cont de modul în care tratează întreruperile de ceas astfel:

* *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. Procesul poare să ruleaze ore întregi și 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 alt proces disponibil). Efectuarea unei *planificări preemptive* necesită o întrerupere de ceas la sfârşitul intervalului menţionat pentru a muta controlul asupra UCP înapoi la planificator.

În cazul în care nu există un ceas, singura opţiune viabilă este *planificarea non-preemptivă*.

### 5.1.2 Algoritmi de planificare în funcție de tipul de aplicație folosit

De la aplicaţie la aplicaţie se observa ca algoritmii de planificare diferă. Există trei tipuri de sisteme care au aplicaţii diferite şi acoperă aproximativ tot domeniul sistemelor de calcul:

* sisteme cu prelucrare pe loturi;
* sisteme interactive;
* sisteme în timp real.

În sistemele cu prelucrare pe loturi nu există utilizatori care aşteaptă un răspuns rapid la terminalele lor. Din acest motiv 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, 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 rău intenţionate.

## 5.2 Concepte pentru proiectarea planificării în sistemele interactive

### 5.2.1 Planificarea Round Robin

Conform algoritmului de planificare Round Robin, fiecare proces primeşte o cuantă de timp de rulare. Să zicem că la sfărşitul cuantei de timp alocate, procesul este încă în execuţie, unitatea centrală de prelucrare lasă procesul curent şi trece la alt proces gata de execuţie (următorul proces). 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.

În figura de mai jos este prezentată planificarea Round Robin, unde (a) este o listă de procese pregătite de execuţie și (b) este o listă proceselor pregătite de execuţie după ce procesul curent (Procesul X) şi-a consumat cuanta.

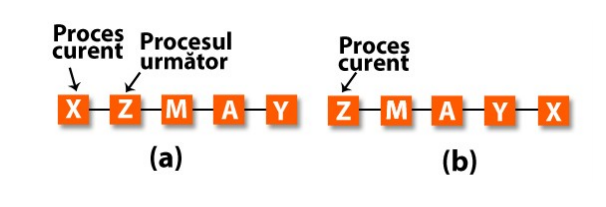


Fig. 8:  *Planificarea Round Robin*

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. Astfel la algoritmul Round Robin se pune problema cât să fie cuanta de mare?

Conform [TEN] dacă comutarea durează o milisecundă și valoarea cuantei este de 4 milisecunde douăzeci la sută din timpul UCP este irosit pe supraîncărcarea cauzată de sarcini administrative. Este prea mult și de aceea am putem folosi şi cuante de 50-150 milisecunde. 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 milisecunde este de multe ori bună.

# 

### 5.2.2 Planificarea bazată pe priorități

Diferența dintre algoritmul Round Robin și planificarea bazată pe priorități este că Round Robin 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.

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 [TEN]. 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 o milisecundă din cuanta sa de 50 milisecunde ar primi prioritatea 50, în timp ce un proces care a rulat 25 milisecunde 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.

### 5.2.3 Shortest Process Next

Procesele interactive urmează de redulă modelul în care se aşteaptă o comandă, se execută comanda, se aşteaptă o comandă, se execută comanda, şi tot așa. Privind execuţia fiecărei comenzi ca pe o lucrare separată putem minimiza întreg timpul de răspuns executând în continuare cea mai scurtă lucrare. Marea problemă constă în a afla care din procesele executabile este cel mai scurt cu putință. Pentru aceasta abordarea 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. Tehnica aceasta se aplică în numeroase situaţii în care trebuie efectuată o prezicere bazată pe valori anterioare. Îmbătrânirea este foarte uşor de implementat pentru a = ½. [TEN]

# 

### 5.2.4 Planificarea garantată

Planificarea garantată este o altă metodă de planificare care constă în tehnica promisiunilor făcute utilizatorilor şi respectarea lor. O promisiune realistă, potrivit [TEN], 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. În 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 (50%) înseamnă că procesul a consumat numai jumătate din capacitatea la care are dreptul, iar un raport de 2,0 (200%) 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.

# 

## 5.3 Planificarea în UNIX

UNIX a fost încă de la apariție un sistem multiprogramare tocmai de aceea algoritmul său de planificare a fost proiectat să asigure un răspuns bun pentru procesele interactive.

Acesta este si motivul pentru care a fost creat un algoritm cu două niveluri :

* *Algoritmul de nivel jos*, acest algoritm alege un proces care să se execute din setul de procese din memorie şi care sunt pregătite să ruleze;
* *Algoritmul de nivel înalt,* acest algoritm mută procesele între memorie şi disc astfel că toate procesele au 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.

### 5.3.1 Implementarea algoritmului de planificare

Știm 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. Astfel este ales şi pornit primul proces din acea coadă. Acestui process îi este permis să ruleze maxim o cuantă de timp, în general 100 milisecunde, 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. O dată pe secundă este recalculată prioritatea fiecărui proces corespunzător unei formule care implică trei componente: Prioritatea = consum\_CPU + nice + base [RUS][TEN]. 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.

## 5.4 Planificarea în LINUX

Planificarea este unul dintre puţinele domenii în care Linux foloseşte un algoritm diferit de UNIX [TEN]. 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:

* FIFO în timp real;
* Round Robin sau rotație în timp real;
* Cu partajarea timpului.

Firele de execuţie FIFO în timp real dețin cea mai mare prioritate şi acestea nu sunt preemptibile.

Firele de execuţie prin rotaţie în timp real sunt la fel ca cele FIFO în timp real, direfența constă în faptul că sunt preemptibile de ceas. În cazul în care 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.

## **5.5 Planificarea în Windows NTOS**

Windows NT nu are un fir de execuţie central pentru planificare. Dar, atunci când un fir de execuţie nu mai poate rula, firul intră în mod nucleu şi rulează chiar el planificatorul pentru a vedea care fir de execuţie să comute. Condițiile care cauzează firul de execuţie curent să execute codul planificatorului [TEN]:

* Firul de execuţie se blochează la un semafor, mutex, eveniment, I/O;
* Semnalizează un obiect;
* Cota de rulare a firului de execuţie a expirat.

În primul caz, 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.

La fel și în al doilea caz, 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 preemptive [RUS].

Planificatorul poate fi apelat în două cazuri:

* Se termină o operaţie de I/O;
* 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 două moduri determină algoritmul în mare.

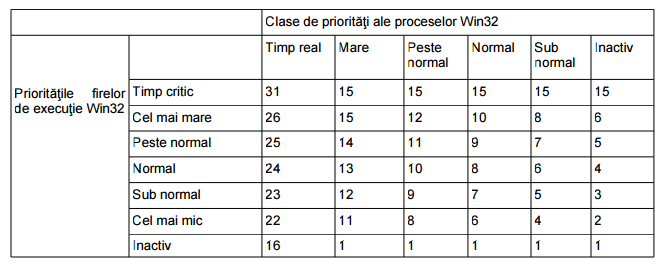


Fig. 9: *Planificare în Windows NTOS*

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 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 de mai jos. 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 tabel. 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. 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.

Altfel spus, atunci când există mai multe fire de execuţie pregătite la cel mai înalt nivel de prioritate, ele rulează conform algoritmului Round Robin pentru o cuantă fiecare. Dacă nu există nici un fir de execuţie pregătit, este rulat firul de execuţie inactiv.

# 

# 

# 

# 6. Procesul de pornire al sistemelor de operare

## 6.1 Introducere/Notiuni generale

Pentru ca un computer sa boot-eze cu succes trebuie ca BIOS-ul sau, sistemul de operare si componentele hardware sa lucreze conform cu specificatiile; cedarea oricareia dintre acestea trei elemente va rezulta cel mai probabil intr-o secventa de pornire esuata.

Cand se porneste alimentarea UCP-ul (Unitatea Centrala de Procesare) se auto-initializeaza, fapt care este pornit de o secventa de cicluri de ceas generare de ceasul sistemului. Parte din initializare o reprezinta verificarea BIOS-ului ROM al sistemului pentru prima instructiune din programul de startup. ROM BIOS-ul contine prima instructiune, care este cea de Power On Self Test (POST) intr-o adresa de memorie predeterminata. POST incepe prin a verifica chip-ul BIOS si apoi testeaza CMOS RAM-ul. Daca POST nu detecteaza o problema de baterie, atunci continua sa initializeze UPC-ul verificand componentele hardware inventariate de sistem (ex: placa video), apoi continua cu mediile de stocare, porturi si alte componente hardware.

In momentul in care POST a decis sa toate componentele functioneaza in parametrii si UCP-ul s-a initializat cu succes, BIOS-ul cauta un SO de incarcat. Defel BIOS-ul cauta pentru chip-ul CMOS sa ii spuna unde sa gaseasca sistemul de operare. De obicei pe PC-uri gasim pe partitia C sistemul, desi BIOS-ul are capacitatea de a cauta si incarca sistemul si de pe alte medii (CD-ROM, Floppy Disk etc.) Ordinea partitiilor in care se uita CMOS pentru a localiza SO-ul se numeste secventa de bootare. Ea poate fi schimbata prin alterarea CMOS-ului. In momentul in care se uita in boot drive-ul corect BIOS-ul va intalni intai boot record-ul care ii spune unde sa gaseasca inceputul SO-ului si programele care vor initializa SO-ul. In momentul in care SO-ul se initializeaza, BIOSul copiaza fisierele in memorie si SO-ul ia controlul asupra procesului de bootare. [WP]

## 6.2 Procesul de startup WINDOWS

In functie de generatia din care face parte sistemul Windows folosit au existat/exista diferite moduri de bootare. Desi secventele de bootare sunt in mare principial aceleasi, exista diferente in meta-familia Windows enumerate mai jos.

### 6.2.1 Windows bazat pe DOS

#### 6.2.1.1 Windows 1.x/2.x

In versiunile Windows de la Windows 1.01 si pana la Windows/386, SO-ul era incarcat in momentul in care fisierul *WIN.COM* era executat. In continuare incarca *win100.bin* sau *win200.bin* si *win100.ovl* sau *win200.ovl*, impreuna cu fisierul de configurare *WIN.INI*. Shell-ul default era MS-DOS Executive.

Modulele *GDI.EXE* , *KERNEL.EXE* si *USER.EXE*, precum si driverele unor dispozitive (ca SUMM.DRV, MOUSE.DRV, KEYBOARD.DRV) sunt parte din *win100.bin/win200.bin* si *win100.ovl/win200.ovl*. [WIKI3]

#### 6.2.1.2 Windows 3.x/9x

In Windows 3.x si 95/98/ME secventa de boot este manageriata de *MS-DOS*. In timpul bootarii, fisierele CONFIG.SYS *si AUTOEXEC.BAT* sunt executate, impreuna cu fisierele de configurare *WIN.INI* si *SYSTEM.INI*.

Cand toate fisierele de configurare si toate driverele dispozitivelor sunt deja incarcate, modulele de 16 biti *krnl386.exe*, *gdi.exe* si *user.exe* sunt si ele incarcate si apoi se trece la DLL-urile pe 32 biti (*kernel32.dll*, *gdi32.dll* si *user32.dll*). Serverul de mesaje pe 32 de biti VxD (Msgsvr32) lanseaza in execuție *Mprexe.exe* (responsabil cu incarcarea clientului de login pentru retea). Cand un utilizator se logheaza in Windows, sunetul de inceput este lansat, *Explorer.exe* (responsabil de GUI)este incarcat din sectiunea de boot a fisierului *SYSTEM.INI.*

In toate versiunile de Windows bazate pe DOS (cu exceptia Windows ME) este posibila bootarea sistemului de operare prin scrierea *win* intr-un prompt de DOS. Sunt o serie de comenzi care pot fi folosite alaturi de comanda *win*:

* /d switch (SO porneste in modul “Safe mode”)
* /d:n switch (SO porneste in modul “Safe mode” folosind *putand* insa si reteaua).

In Windows 3.1 mai sunt o serie de optiuni la dispozitie:

* /3 porneste Windows in 386 enhanced mode
* /S porneste Windows in modul standard [WIKI3]

### 6.2.3 Windows NT

In Windows NT bootloaderul se numeste NTLDR. El e responsabil de accesarea sistemului de fisiere pe drive-ul de bootare, de startarea ntoskrnl.exe si de incarcarea driverelor dispozitivelor de bootare in memorie. In momentul in care toare driverele de boot si de sistem sunt deja incarcate kernelul (fir de executie al sistemului) porneste Session Manager Subsystem (smss.exe), care la randul sau porneste [winlogon](http://en.wikipedia.org/wiki/Winlogon), care incarca biblioteca [graphical identification and authentication](http://en.wikipedia.org/wiki/Graphical_identification_and_authentication) (GINA).

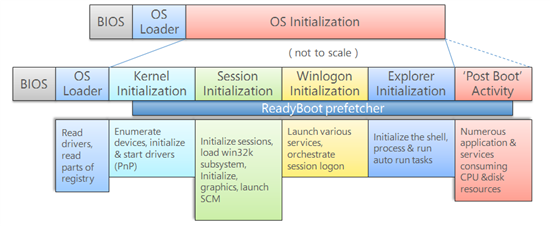
Dupa ce un user s-a autentificat cu succes pe masina winlogon face urmatorii pasi:

1. Sunt aplicate setarile de User and Computer Group Policy;
2. Programele de startup sunt pornite de la urmatoarele locatii:

* *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Runonce*
* *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\Explorer\Run*
* *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run*
* *HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows\Run*
* *HKCU\Software\Microsoft\Windows\CurrentVersion\Run*
* *HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce*
* *All Users ProfilePath\Start Menu\Programs\Startup\* (aceasta cale este localizata pe masinile care ruleaza versiuni ale Windows in alte limbi decat engleza)
* *Current User ProfilePath\Start Menu\Programs\Startup\* (aceasta cale este localizata pe masinile care ruleaza versiuni ale Windows in alte limbi decat engleza) [WIKI3]

### 6.2.4 Windows 7

Secventa de start a Windows incepand cu Windows Vista este usor diferita de versiunile anterioare de Windows care folosesc NT Kernelul. Boot loader-ul Windows Vista se numeste winload.exe si este apelat de Windows Boot Manager. Aditional, GINA, care a fost folosit de toate versiunile anterioare este inlocuit in totalitate de *Credential Providers*.



*Fig. 10: Schema de principiu a procesului de pornire in Windows*

Subfazele pornirii sistemului de operare Windows 7 sunt: [MICROSOFT]

* **PreSMSS: Initializarea Kernelului**

Aceasta sub-faza incepe in momentul apelarii kernelului. In timpul acesteia, kernelul initializeaza structurile de date si componentele. De asemenea, porneste PnP Manager-ul care initializeaza BOOT\_START driverele care sun incarcare in faza OSLoader.

* **SMSSInit: Initializarea sesiunii**

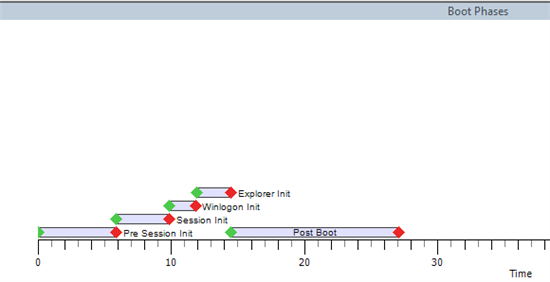
Aceasta etapa incepe in momentul in care kernelul paseaza controlul manageului de sesiune (smss.exe). In timpul acestei sub-faze, sistemul initializeaza registrii si porneste procesele de subsistem. Procesul inceteaza cand controlul este preluat de winlogon.exe.

* **WinLogonInit: Initializarea Winlogon**

In timpul acestui proces apare ecranul de logare, incepe service control managerul si ruleaza scriptul Group Policy. Etapa se incheie la pornitea Explorer-ului

* **ExplorerInit: Initializarea Explorer-ului**

Aceasta etapa incepe odata cu procesul explorer.exe. In timpul ei sistemul creeaza procesul desktop window manager (DWM), care initializeaza desktopul si il afiseaza pentru prima oara.



*Fig. 11: Reprezentarea temporala a bootarii sistemului Windows*

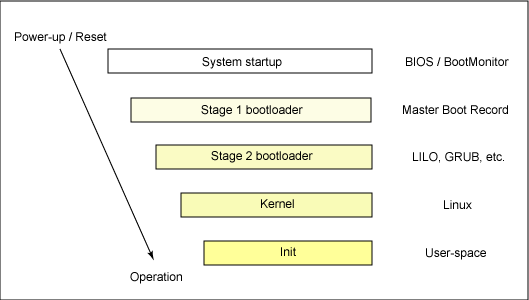
Ultima faza, cea **Post Boot** include toate activitatile care au loc dupa ce desktopul este gata. Utilizatorul poate interactiona cu desktopul, dar, in spate, sistemul poate porni servicii, tray icon-uri si coduri de apricatie, avand un potential raspuns in modul in care utilizatorul percepe responsivitatea sistemului.

## 6.3 Bootarea sistemului de operare Linux

Atunci când un sistem este mai întâi pornit, sau este resetat, procesorul execută codul într-o locație bine-cunoscute. Intr-un calculator personal (PC), această locație este în BIOS, care este stocat in memorie flash pe placa de baza. Unitatea centrală de procesare (CPU) într-un sistem embedded invoca vectorul de resetare pentru a începe un program de la o adresă cunoscută in flash / ROM. In ambele cazuri, rezultatul este acelasi. Deoarece PC-uri oferă atât de mult flexibilitate, BIOS-ul trebuie să determine care dispozitive sunt candidați pentru boot.

Când un dispozitiv de pornire este găsit, first stage bootloader-ul este încărcat în memoria RAM și executat. Acest bootloader are mai putin de de 512 octeți în lungime (un singur sector), iar sarcina sa este de a încărca al doilea stadiu de boot.

Când al doilea stadiu de boot este încărcat în RAM și executat, un ecran splash este defel afisat, și Linux și un disc opțional RAM (sistem de fișiere rădăcină temporar) sunt încărcate in memorie. Când imaginile sunt încărcate, a doilea stadiu de boot preda controlul imaginii kernel si kernelul este decompresat si initializat. In acest moment, al doilea stadiu de bootloader verifica hardware-ul sistemului, enumera dispozitivele hardware atasate, monteaza dispozitivul de root ai apoi incarca moddulele kernel necesare. Cand este completa secventa, primul program din spatiul de lucru al userului porneste (init) si initializarea high-level a sistemului are loc.



*Fig. 12: Secventele principale din procesul de bootare al Linux*

**Pornirea sistemului**

Pornirea sistemului depinde de hardware-ul pe care bootam LINUX. Pe o platforma embedded un mediu bootstrap este folosit cand sistemul e pornit sau resetat. Exemplele include U-boot, RedBoot si MicroMonitor de la Lucent. Platformele embedded sunt de obicei livrate cu un boot monitor. Aceste programe au dedicate o regiune speciala in memoria flash al hardware-ului target si ofera mijloacele necesare LINUX-ului pentru a downloada o imagine kernel intr-o memorie flash si pentru a o executa. În plus față de capacitatea de a stoca și boot-a o imagine Linux, aceste boot monitors pot efectua o testare de sistem și de inițializare hardware. Într-o țintă embedded, aceste boot monitors acoperă de obicei atât primul cat si al doilea nivel al bootloaderului.

Pentru a vedea continutul MBR-ului din sistemul nostru LINUX folosim urmatoarea comanda:

*# dd if=/dev/hda of=mbr.bin bs=512 count=1*

*# od -xa mbr.bin*

Intr-un PC secventa de boot a LINUX-ului incepe la adresa 0XFFFF0. Primul pas al BIOS-ului este POST-ul al carui scop este verificarea hardware-ului. Al doilea pas al BIOS-ului este enumerarea dispozitivelor locale si initializarea acestora. Dat fiind diferitele moduri de utilizare ale functiilot BIOS, BIOS-ul este alcatuit din 2 parti: codul POST si serviciile runtime.

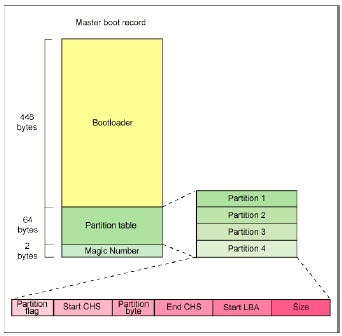
Prima actiune efectuata de calculator la pornire este testul POST (Power On Self Test) care se testeaza componentele hardware care exista in sistem (procesorul, memoria, placa video, tastatura, mouse-ul, s.a.) Tot aici este testat *mediul* de BOOT (hard disk, unitate floppy, CDROM). Dupa POST, loader-ul din ROM incarca *sectorul* de BOOT, care la randul lui incarca SO-ul de pe partitia activa cautata in tabelul de partitii.

Pentru a boota un sistem de operare, BIOS-ul cauta dispozitive care sunt in acelasi timp active si bootabile in ordinea preferintelor definite de CMOS (complementary metal oxide semiconductor) un dispozitiv de boot poate fi un floppy disk, CD-ROM, o partitie pe hard disk, un dispozitiv din retea, sau chiar un USB flash memory stick.

In mod normal LINUX este bootat de pe un HDD, unde un MBR (Master Boot Loader) contine boot loader-ul primar. MBR-ul este un sector de 512 bytes localizat in primul sector de pe disc (sector 1 al cilindrului 0, capul 0). Dupa ce MBR-ul este incarcat in RAM, BIOS-ul detine controlul asupra sa. [TGS] [IBM2]

**Stage 1 Boot loader-ul**

Bootloaderul primar care se afla in MBR este o imagine de 512 bytes care contine atat codul de program cat si un mic table de partitii. Primi 446 de bytes fac parte din primary boot loader, care contine atat cod executabil cat si text pentru mesajele de eroare. Urmatorii 64 de bytes sunt tabelul de partitii care contine o inregistrare pentru fiecare din cele 4 partitii (fiecare de 16 bytes). MBR-ul se termina cu 2 bytes care sunt definiti ca “numarul magic” (0xAA55). Rolul acestui numar serveste ca verificare pentru validare a MBR-ului.



*Fig. 13: Anatomia MBR (Master Boot Loader)*

Rolul Primary Bootloader-ului este de a gasi si a incarca Secondary Bootloaderul (Stage 2). Face asta prin cautarea in tabelul de partitii a unei particii active. Cand o gaseste, scaneaza partitiile ramase in tabel pentru a se asigura ca acestea sunt inactive. Cand verificarea este facuta, boot record-ul partitiei active este citit in RAM si executat. [IBM2]

**Stage 2 Bootloader-ul**

Bootloader-ul secundar sau Stage 2 poate fi mai precis numit *kernel loader.* Sarcina acestui stadiu este de a incarca kernel-ul LINUX si si, optional, discul RAM.

Stage 1 bootloader-ul si Stage 2 Bootloader-ul combinatese numesc Linux Loader (LILO) sau GRandUnified Bootloader (GRUB) in mediul x86 PC.

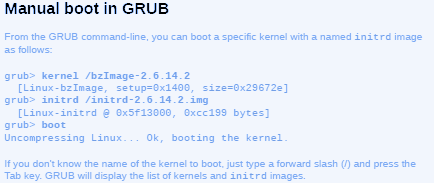
GRUB cunoaste structura fisierelor de sistem LINUX si, in loc sa se uite pe sectoare de disc, poate incarca kernelul LINUX dintr-un fisier de sistem ext2 sau ext3. Face asta ptin transformarea bootloader-ului format din doua stadii in trei stadii. Stadiul 1 (MBR) booteaza un stadiu 1.5 al bootloaderului care intelege fisierul particular care contine imaginea Linux kernel.. *Exemple: reiserfs\_stage1\_5 (pentru a incarca dintr-un fisier de sistem Reiser journaling) sau e2fs\_stage1\_5 (pentru a incarca dintr-un fisier de sistem ext2 or ext3).*

Avand stadiul 2 incarcat, GRUB poate, la cerere sa afiseze o lista de kerneluri disponibile (definite in */etc/grub.conf*, cu linkuri din */etc/grub/menu.lst* si */etc/grub.conf*). Pot fi selectate kerneluri si chiar adaugati parametri acestora. Acest lucru se poate face si din linia de comanda din interfata. [IBM2]



*Fig. 14: Mostra din codul GRUB-ului CentOS*

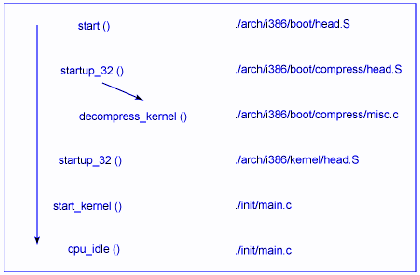
**Kernelul**



*Fig. 15: Bootarea unui Kernel anume din linia de comanda a GRUB*

Avand imaginea kernel incarcata in memorie si controlul asupra OS-ului dat lui Stage 2 Bootloader, poate incepe stadiul kernelului. Imaginea kernel este o compresie mai mult decat un fisier executabil. De obicei aceasta este o imagine zIimage (compressed image, mai mica de 512KB) sau bzImage (big compressed image, mai mare de 512KB) care a fost compresata in preazabil cu zlib. La inceputul imaginii se afla o rutina care face un minim de setup hardware si apoi decompreseaza imaginea kernel continuta in fisierul compresat si o plaseaza in memorie. Daca este presenta o imagine disc RAM aceasta rutina o muta in memorie si o retine pentru utilizare mai tarziu. Rutina mai apoi apeleaza kernelul si bootarea kernelului incepe.

Cand o imagine bzImage (pentru un i386) este apelata, incepem la *./arch/i386/boot/head.S* in rutina de asamblare de start (vezi figura de mai jos). Aceasta rutina face verificari de baza ale hardware-ului si invoca rutina startup\_32 din *./arch/i386/boot/compressed/head.S*. Aceste seturi de rutine pun la punct un mediu de baza (stack, etc.) si curata Block Started by Symbol (BSS). Kernelul este apoi decompresat prin chemarea unei functii in C numita decompress\_kernel (locaalizata in *./arch/i386/boot/compressed/misc.c*). Cand kernelul este decompresat in memorie este chemat. Aceasta este de asemenea o alta functie startup\_32, localizata in *./arch/i386/kernel/head.S*. [TGS] [IBM2]



*Fig. 16: Fluxul functiilor pentru bootarea kernelului Linux i386*

La apelul lui start\_kernel, o lunga lista de initializari sunt chemate pentru a seta intreruperile, pentru a configura mai departe memoria si pentru a incarca discut initial de RAM. In final se apeleaza tokernel\_thread (in arch/i386/kernel/process.c) pentru a incepe functia init, care este primul proces din user-space. La sfarsit este pornit idle task si schedulerul poate prelua controlul (dupa apelarea lui cpu\_idle). Cu intreruperile activare, schedulerul pre-emptiv poate prelua controlul pentru a asigura multi-taskingul.

In timpul bootarii kernelului, discul initial-RAM care a fost incarcat in memorie de Stage 2 Bootloader este copiat in RAM si montat. Acest initrd serveste ca fisier de sistem root temporar si permite kernelului sa booteze fara a monta discuri fizice. Desi kernelul este foarte mic acesta permite configurarea unei multitudini de periferice deoarece acestea pot fi parte din initrd. Dupa ce kernelul booteaza este pivotat fisierul de sistem root (via pivot\_root) unde initrd este demontat si fisierul real de sistem root este montat.

Functia *initrd* permite crearea de mici kerneli Linux cu drivere compilate ca module incarcabile. Aceste module dau kernelului mijloacele de a accesa discuri si fisiere de sistem pe acele discuri precum si drivere pentru alte dispozitive. Pentru ca fisierul root este un fisier de sistem pe disc, functia initrd ofera mijloacele de bootstrapping necesare accesului la disc si la montarea fisierului real de sistem, root. Intr-un sistem embedded fara un HDD initrd poate fi fisierul final root sau acest fisier poate fi montat via Network File System(NFS).

**Init**

Dupa ce kernelul este bootat si initializat, acesta porneste user-space-ul. Acesta este primul program invocat care este compilat in biblioteca standard C. Inainte de acest punct nu au fost utilizate aplicatii standard C.

Intr-un sistem desktop Linux prima aplicatie pornita este de obicei */sbin/init.* Intr-un sistem embedded putem invoca un script shell care sa porneasca aplicatiile necesare.

Init vede nivelul de rulare al Linux verificant fisierul */etc/inittab*. Pentru a identifica nivelul default de rulare al sistemului executam comanda: *grep initdefault /etc/inittab.* [IBM2]

Nivelele de rulare existente sunt urmatoarele: [TGS]

* + 0 – halt
  + 1 – Single user mode
  + 2 – Multiuser, without NFS
  + 3 – Full multiuser mode
  + 4 – unused
  + 5 – X11
  + 6 – reboot

## 6.4 Concluzii

In privinta startarii sistemului de operare procesele sunt similare pe masini diferite, desi sistemele de operare nu sunt aceleasi. Acest fapt reprezinta respectarea principiilor de portabilitate si capacitate de a instala SO-uri diferite pe masini asemanatoare. Avand principii arhitecturale similare si masinile la nivel fizic, este evident faptul ca SO-urile vor avea metose similare de pornire.

# 

# 

# 7. Bibliografie

1. [WIKI1] http://en.wikipedia.org/wiki/Operating\_system
2. [HSF] http://computer.howstuffworks.com/operating-system1.htm
3. [CS] http://swarm.cs.pub.ro/~razvan/so-2005/lab2/lab2/node1.html
4. [UIC1] http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4\_Threads.html
5. [IBM1] https://www-01.ibm.com/support/knowledgecenter/ssw\_aix\_61/com.ibm.aix.kernextc/kernextc\_kern\_env.htm
6. [UIC2] http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3\_Processes.html
7. [MTU] http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
8. [TAN] http://stst.elia.pub.ro/news/SO/Modern%20Operating%20System%20-%20Tanenbaum.pdf
9. <http://siber.cankaya.edu.tr/OperatingSystems/ceng328/ceng328.html>
10. <http://reiber.org/nxt/pub/Linux/LinuxKernelDevelopment/Linux.Kernel.Development.3rd.Edition.pdf>
11. [ALP]<http://www.advancedlinuxprogramming.com/alp-folder/advanced-linux-programming.pdf>
12. <http://www.tldp.org/LDP/lpg/node7.html>
13. [Tanenbaum]<http://stst.elia.pub.ro/news/SO/Modern%20Operating%20System%20-%20Tanenbaum.pdf>
14. [LJ]http://www.linuxjournal.com/article/3326
15. [SO] Ştefan Stăncescu, Note de curs: Sisteme de operare, 2007
16. [TEN] Andrew S. Tanenbaum, Sisteme de operare moderne, Byblos, 2004
17. [LDP] http://tldp.org/LDP/tlk/tlk.html
18. [WIKI3] <http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library>
19. [LIN] http://www.linux.ro
20. [RUS] Russinovich, M.E. și Salomon, D.A. Microsoft Windows Internals, 2005
21. [WP] <http://www.webopedia.com/DidYouKnow/Hardware_Software/BootProcess.asp>
22. [WIKI3] <http://en.wikipedia.org/wiki/Windows_startup_process>
23. [IBM2] <https://www.ibm.com/developerworks/library/l-linuxboot/>
24. [TGS] <http://www.thegeekstuff.com/2011/02/linux-boot-process/>
25. [MICROSOFT1] <http://social.technet.microsoft.com/wiki/contents/articles/11341.the-windows-7-boot-process-sbsl.aspx>