Sistemele Unix/Linux(II)

Stările unui proces Unix şi tranziţia între stări.

Gestiunea memoriei sub Linux

Procese şi semnale sub UNIX

Planificarea sub Linux

Stările unui proces Unix şi tranziţia între stări.

- Un proces sub Linux se poate afla într-una dintre următoarele stări:

  `New`: procesul este creat, dar încă nu este gata de execuţie.

  `ReadyMemory`: procesul ocupă loc în memoria RAM şi este gata pentru a fi executat.
  `ReadySwapped`: procesul ocupă spaţiu numai în memoria secundară (memoria virtuală sau zona swap pe disc), dar este gata de execuţie.

  `RunKernel`: instrucţiunile procesului sunt executate în mod nucleu sau protejat; astfel de instrucţiuni corespund unui apel sistem, unui handler de întreruperi etc.

  `RunUser`: sunt executate instrucţiunile cod maşină ale procesului în mod utilizator.

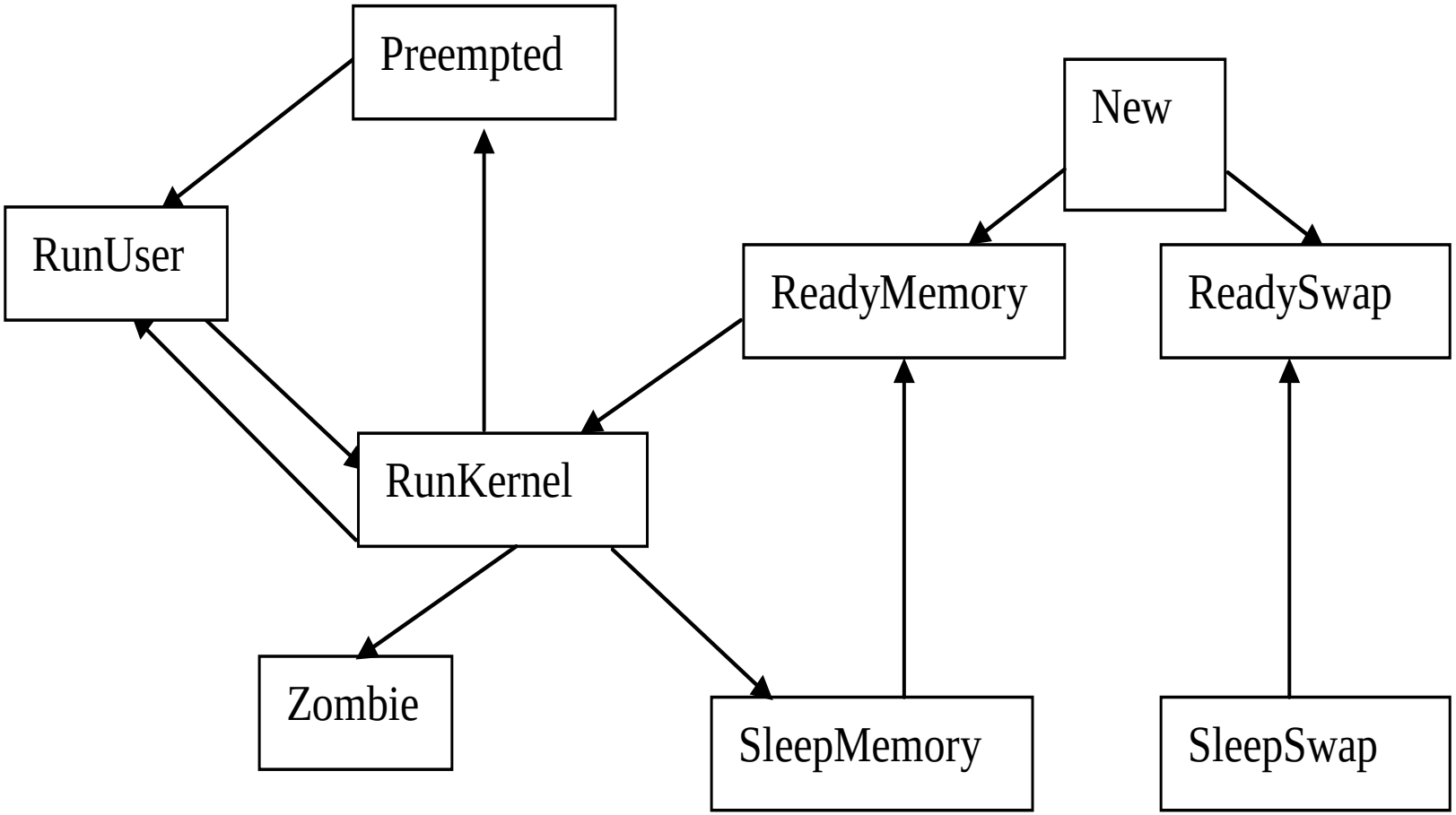
  `Preempted`: corespunde unui proces care a fost forţat de către un alt proces cu o prioritate superioară.

`SleepMemory`: procesul se află în memorie, dar nu poate fi executat până când nu se produce un eveniment care să-1 scoată din această stare.

`SleepSwapped`: procesul este evacuat pe disc şi el nu poate fi executat până când nu se produce un eveniment care să-1 scoată din această stare.

`Zombie`: s-a terminat execuţia procesulului, înaintea execuţiei procesului părinte (noţiune care o vom discuta în mai târziu).

- În figura urm. este prezentată diagrama stărilor în care se poate afla un proces şi a tranziţiilor posibile între aceste stări.


Preempted
New
RunUser
ReadyMemory
ReadySwap
RunKernel
Zombie
SleepMemory
SleepSwap

- Tranziţiile posibile sunt:

  `New→ReadyMemory`: există suficientă memorie internă pentru a fi alocată procesului.

  `New→ReadySwap`: Nu există suficientă memorie internă pentru a fi alocată procesului şi procesului i se alocă numai spaţiu în memoria swap (virtuală).

  `RunUser→RunKernel`: Procesul cere execuţia unui apel de sistem.

  `RunKernel→RunUser`: Revenire dintr-o operaţie executată în mod protejat.

  `RunKernel→Zombie` : Se aşteaptă terminarea procesului părinte, pentru a se actualiza anumite statistici; procesul respectiv nu mai are alocate resurse, el există numai în tabela cu procese a sistemului.

  `RunKernel→Preempted` : Trecerea în starea `Preempted` apare atunci când în sistem se impune execuţia unui proces cu o prioritate mai înaltă; acest lucru este decis de nucleu, deoarece nu poate fi forţat un proces care execută o operaţie în mod nucleu.

`Preempted→RunUser`: Când procesul aflat în starea `Preempted` are prioritatea cea mai înaltă, este trecut în `RunUser`.

`RunKernel→SleepMemory`: Se aşteaptă apariţia unui eveniment ca să se continue execuţia procesului.

`SleepMemory→ReadyMemory` : Procesul a fost trezit de evenimentul corespunzător şi concurează pentru a fi servit de CPU.

`SleepMemory→SleepSwapped` : Pentru procesul respectiv, nu mai există suficientă memorie internă, aşa că este trecut în memoria swapp.

`SleepSwapped→ReadySwap`: S-a întâmplat evenimentul necesar trezirii procesului, dar nu există suficientă memorie internă pentru acesta.

Gestiunea memoriei sub Linux

- **Structura spaţiului de memorie alocat unui proces.**
- Fiecare proces are alocat un spaţiu de adrese din memoria internă şi unul din memoria virtuală.
- Procesul accesează memoria internă în două moduri: **utilizator** şi **nucleu.**
- Partea accesată în mod **utilizator** este formată din mai multe porţiuni.

  -Partea de **instrucţiuni** cod maşină a procesului, care poate fi partajată cu alte procese.

  - Datele iniţializate **numai pentru citire** reprezintă constantele utilizate de către proces.

  - Datele iniţializate pentru **citire** şi **scriere** reprezintă variabilele procesului iniţializate la compilare.
- Conţinutul acestor trei zone corespunde fişierului executabil corespunzător procesului.
- Pe lângă acestea, un fişier executabil mai conţine şi alte informaţii care sunt folosite de către nucleul Linux la iniţializarea modului de lucru protejat (nucleu).

- **Date neiniţializate** este zona ocupată de către restul variabilelor, cu excepţia celor cărora li se alocă spaţiu pe stivă sau în zona heap. Pentru acestea nu se rezervă spaţiu în fişierul executabil.
- **Heap** este zona unde se alocă spaţiu pentru variabilele dinamice.
- **Stiva** este zona de memorie folosită pentru transferul parametrilor, în cazul apelului de funcţii.
- **Tabela proceselor** conţine câte o intrare pentru fiecare proces;
- **Tabela paginilor de memorie virtuală** coţine câte o intrare pentru fiecare pagină din memorie, în care sunt păstrate diverse informaţii, printre care: procesul care foloseşte pagina respectivă, drepturi de acces etc. În cazul sistemelor Linux, care folosesc o schemă de alocare segmentată şi paginată, mai există şi o tabelă de translatare segment - pagină de memorie virtuală.
- **Tabela descriptorilor de fişiere** deschise din sistem conţine câte o intrare pentru fiecare descriptor deschis.
- **Tabela inodurilor** conţine câte o intrare pentru fiecare i-nod de pe disc

- **Porţiunea utilizată în mod nucleu** este administrată de către nucleul Linux şi este accesibilă procesului numai în modul de execuţie protejat, prin apeluri de sistem specifice. Ea are o parte statică (de dimensiune fixă) şi una dinamică (de dimensiune variabilă). Această zonă conţine:
  - **prioritatea procesului**; un număr natural cuprins între 1 şi 39 folosit la planificarea execuţiei;
  - **semnalele netratate** trimise procesului;
  - **statistici de timp** (timpul de utilizare CPU, din momentul creării lui, folosit de algoritmul de planificare execuţiei);
  - **statutul memoriei** procesului (imaginea procesului se află în memoria principală sau în memoria swap);
  - **pointer** la următorul proces din coada de procese aflate în starea *Ready* (dacă procesul curent este în starea *Ready*);
  - **descriptorii evenimentelor** care au avut loc cât timp procesul a fost în starea *Sleeping*.
  - **Zona User**: este memorată în spaţiul de adrese al procesului, dar este accesibilă numai din modul de execuţie nucleu (intrarea în tabela proceselor se află în nucleu). Ea conţine informaţii de control, care trebuie să fie accesibile nucleului sistemului de operare când se execută în contextul acestui proces şi anume:

- pointer la intrarea în tabela proceselor care corespunde acestei zone;
- UID, EUID, GID şi EGID - folosite la determinarea drepturilor de acces ale procesului;
- timpul de execuţie al procesului, atât în modul user, cât şi în modul nucleu;
- vectorul acţiunilor de tratare a semnalelor; la recepţionarea unui semnal, în funcţie de valorile acestui vector, procesul poate să se termine, să ignore semnalul sau să execute o anumită funcţie specificată de utilizator;
- terminalul de control al procesului (cel de pe care a fost lansat în execuţie, dacă există);
- valorile returnate şi posibilele erori rezultate în urma efectuării unor apeluri sistem;
- directorul curent şi directorul rădăcină;
- zona variabilelor de mediu;
- restricţii impuse de nucleu procesului (dimensiunea procesului în memorie etc.);

- tabela descriptorilor de fişiere conţine datele relative la toate fişierele deschise de
către proces;
- masca drepturilor implicite de acces pentru fişiere nou create de către proces.
- tabela regiunilor din memorie ale procesului este o tabelă de translatare a adreselor virtuale în adrese fizice pentru regiunile (secţiunile) programului, care este executat în contextul acestui proces;
- stiva nucleu este memoria alocată pentru stiva folosită de nucleul sistemului de operare în modul de execuţie nucleu;
- contextul registru (hardware) este o zonă de memorie unde se salvează conţinutul regiştrilor generali, de stivă, de execuţie, de control şi de stare în momentul în care algoritmul de planificare decide să dea controlul procesorului unui alt proces. Aceşti regiştrii ai procesorului se salvează pentru a putea relua execuţia procesului, de unde s-a rămas

- Nucleul Linux trebuie să administreze toate aceste zone din memoria virtuală ale unui proces.
- În tabela procesului, există un pointer către o structură numită `mm_struct`.
- Această structură conţine informaţii despre imaginea executabilă şi un pointer către tabelele cu paginile procesului.
- Ea conţine pointeri către o listă cu elemente de tipul `vm_area_struct`; cele mai importante câmpuri ale structurii `vm_area_struct` sunt:

  `vm_start`, `vm_end`; reprezintă adresa de început, respectiv de sfârşit ale zonei de memorie (aceste câmpuri apar în `/proc/*/maps`);

  `vm_file`, pointer-ul la structura de fişiere asociată (dacă există);

  `vm_pgoff`, deplasamentul zonei în cadrul fişierului;

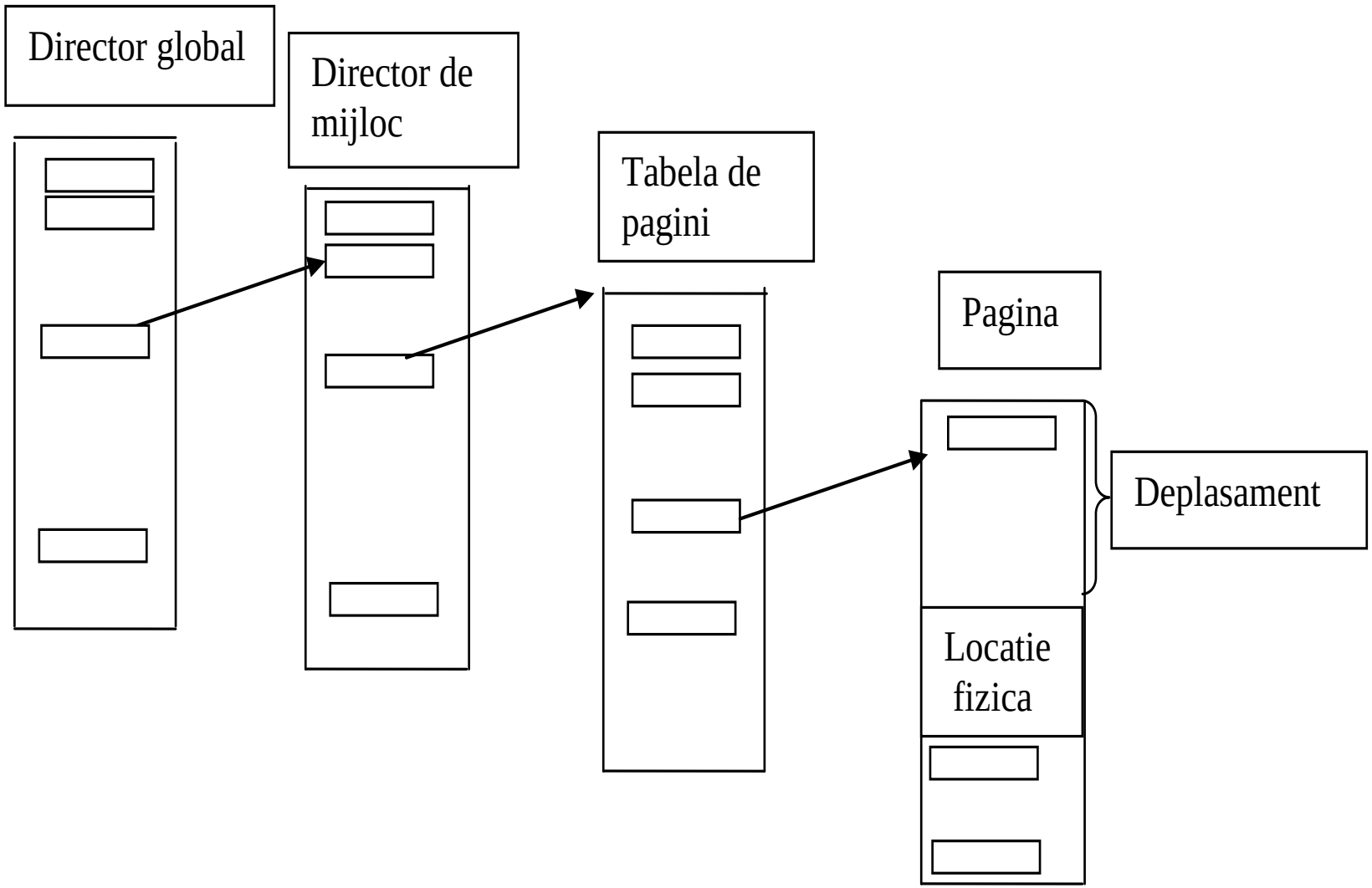
  `vm_flags`, un set de indicatori;

  `vm_ops`, un set de funcţii de lucru asupra acestei zone.
- Toate structurile `vm_area_struct` corespunzătoare unui proces sunt legate printr-o listă; pentru ca regăsirea datelor să se facă mai rapid, ordinea în listă este dată de ordinea crescătoare a adreselor virtuale de unde încep zonele respective.
- Zonele de memorie ale unui proces pot fi vizualizate inspectând `procfs`.

- **Exemplu.**

```
$cat /proc/1/maps
08048000-0804f000 r-xp 00000000 03:01 401624 /sbin/init
0804f000-08050000 rw-p 00007000 03:01 401624 /sbin/init
08050000-08071000 rw-p 08050000 00:00 0
40000000-40016000 r-xp 00000000 03:01 369654 /lib/ld-
  2.3.2.so
40016000-40017000 rw-p 00015000 03:01 369654 /lib/ld-
  2.3.2.so
40017000-40018000 rw-p 40017000 00:00 0
4001d000-40147000 r-xp 00000000 03:01 371432
  /lib/tls/libc-2.3.2.so
40147000-40150000 rw-p 00129000 03:01 371432
  /lib/tls/libc-2.3.2.so
40150000-40153000 rw-p 40150000 00:00 0
bffff000-c0000000 rw-p bffff000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

- **Relocarea paginilor virtuale de memorie sub Linux**.
- În cazul sistemelor de calcul cu procesor pe 64 de biţi, o adresă virtuală este formată din 4 câmpuri: **director global**, **director de mijloc**, **pagină** şi **deplasament**.
- Astfel, pentru translatarea unei adrese virtuale într-una fizică se foloseşte o structură arborescentă pe 3 niveluri (figura urm).
- Fiecare proces are alocată o intrare în tabela director global.
- Această intrare conţine un pointer către tabela director de mijloc şi în această tabelă există o intrare care conţine un pointer către tabela de pagini; în această tabelă, există o intrare care conţine un pointer către o pagină din memoria fizică, din care pe baza deplasamentului se obţine o locaţie fizică în care este încărcată pagina virtuală respectivă.
- În cazul sistemelor care funcţionează cu un microprocesor pe 32 de biţi, funcţionează o schemă pe două niveluri.


Director global
Director de mijloc
Tabela de pagini
Pagina
Deplasament
Locatie fizica

- **Algoritmul de înlocuire a paginilor**.
- Fiecare pagină virtuală încărcată într-o pagină fizică alocată unui proces, are asociată o variabilă reprezentată pe 8 biţi.
- De fiecare dată când se face referinţă la o pagină se incrementează variabila asociată.
- Aceste variabile (asociate paginilor virtuale incarcate in pagini fizice), formează o listă.
- Periodic, sistemul parcurge această listă şi decrementează valorile acestor variabile.
- O pagină pentru care variabila asociată are valoarea `0`, este eligibilă pentru a fi înlocuită în momentul când se pune problema încărcării unei noi pagini virtuale.
- Dacă există mai multe pagini eligibile, se alege una dintre ele folosind regula FIFO.
- Această metodă este o variantă a algoritmului clasic LRU (**L**east **R**ecently **U**sed).
- Linuxul admite existenţa modulelor încărcate dinamic, de obicei drivere de dispozitiv. Acestea pot fi de o dimensiune arbitrară şi necesită o zonă contiguă din memoria internă gestionată de nucleu. Pentru a se obţine astfel de zone de memorie, se foloseşte metoda alocării de memorie prin camarazi.

- **Obs**.

  - Fiecare proces Linux pe un calc. de 32 de biti are alocat un spatiu virtual de adrese propriu de 32 de GB; 1 GB este rezervat pentru tabelele de pagini si alte date ale nucleului, necesar atunci cind procesul intra in mod protejat.

  - Spatiul virtual de adrese este impartit in zone sau regiuni omogene, contigue, aliniate la pagina, adica fiecare zona reprezinta un numar de pagini consecutive, cu aceleasi drepturi si proprietati. Marimea paginii poate fi de 4K sau 8K.

- Fiecare zona este descrisa in nucleu de o inregistrare `vm_area_struct`.

Utilizarea semnalelor

- Linux foloseste semnalele pt. :

  - a comunica cu procesele care sunt in executie.

  - a opri, a lansa si a omori procese.

  - se poate controla o anumita actiune in cadrul unui script, in sensul ca ea sa se execute cind se primeste un anumit semnal de la sistemul Linux.
- Principalele semnale sunt:

Nr.	Denum.	Efect
1	SIGHUP	Amână procesul.
2	SIGINT	Interupe procesul.
3	SIGQUIT	Stopeaza procesul.
9	SIGKILL	Termina procesul neconditionat.
15	SIGTERM	Termina procesul daca este posibil.
17	SIGSTOP	Stopeaza neconditionat, dar nu termina procesul.
18	SIGTSTP	Stopează procesul sau face o pauză in exec. lui, dar nu il termina.
19	SIGCONT	Continua un proces stopat.

- Implicit, bash ignora semnalele SIGQUIT si SIGTERM pe care le primeste (un script interactiv nu poate sa fie terminat accidental).
- Bash proceseaza semnalele SIGHUP si SIGINT pe care le primeste.
- Daca bash primeste un semnal SIGHUP, executa operatia `exit`. Inainte de a o face, transmite semnalul SIGHUP la toate procesele lansate de bash (cum ar fi scriptul unui utilizator).
- Semnalul SIGINT provoaca intreruperea shellului. Nucleul Linux returneaza timpul CPU. De as., shell transmite un semnal SIGINT la toate procesele lansate de shell, ptr. a le notifica aceasta situatie.
- Shellul transmite aceste semnale scriptului ptr. prelucrare. Comportamentul implicit al unui script, este de a ignora semnalele, ceea ce poate avea un efect invers, fata de logica scriptului. Pt. a evita aceasta situatie, se poate programa scriptul pt. a recunoaste semnalele si a executa anumite comenzi in acest sens.
- **Generarea semnalelor**
- Bash permite generarea a doua semnale de baza folosind combinatii de taste.
- **Interuperea proceselor. Ctrl-C** genereaza un semnal SIGINT si il transmite proceselor care sunt in mod curent in executie in cadrul shell-ului. Acest lucru se poate testa executind o comanda care in mod normal are un timp lung de executie.

- **Exemplu**: Daca tastam: `$ sleep 100` terminarea executiei comenzii s-ar produce dupa `100` sec. Daca tastam Ctrl-C, executia se termina imediat.
- **Stoparea procesului.** Combinatia Ctrl-Z genereaza un semnal SIGTSTP, carea stopeaza orice proces care se executa in cadrul shell.
- **Stoparea procesului** este diferita de terminarea lor. Executia unui proces stopat poate continua.
- Cind folosim Ctrl-Z, shell-ul informeaza ca procesul a fost stopat.
- **Exemplu**: Daca tastam: `$ sleep 100` si apoi Ctrl-Z, se va afisa:

  ```
  [1]+ Stopped sleep 100
  ```
- Numarul dintre paranteze drepte este numarul *job-lui* asignat de catre shell. Shell-ul se refera la fiecare proces executat ca un fiind un *job*, si asigneaza fiecarui job un numar unic. Numerele sunt alocate in ordinea lansarii in executie.
- Daca avem un job stopat, assignat sesiunii shell, bash ne va avertiza daca incercam sa iesim din sesiune:

  ```
  $ exit
  logout
  There are stopped jobs.
  ```
- Daca dorim cu adevarat sa parasim shell-ul, in conditiile in care job-ul stopat este inca activ, tastam iar comanda `exit`.

Identificatorul de proces(PID)

- Fiecare proces este reprezentat printr-un identificator unic (*pid*).
- Procesul cu pid-u `0` (*idle process)* — procesul pe care nucleul il executa cind in sistem nu sunt procese executabile.
- Procesul cu *pid 1* este procesul *init*, invocat de nucleu la sfârşitul procedurii de încărcare a sistemului de operare.
- In afara cazului in care utilizatorul specifica explicit nucleului care proces sa-l execute (prin intermediul comenzii *init*), nucleul trebuie sa identifice un proces corespunzator (`init`) al carui proprietar este.
- Nucleul Linux incearca unul dintre urmatoarele procese, in urmatoarea ordine:

  1. */sbin/init*: Locatia cea mai des intilnita.
  2. */etc/init*: O alta locatie ptr. procesul `init`.
  3. */bin/init*: O posibila locatie ptr. procesul `init`.
  4. */bin/sh*: Locatia pentru shell-ul Bourne, pe care nucleul incearca sa-l execute atunci cind esueaza in cautarea procesului `init`.
- Primul dintre aceste procese care exista este executat ca proces `init`. Daca niciunul dintre aceste patru procese nu poate fi executat, nucleul Linux intra intr-o stare de asteptare.
- Dupa gasirea de catre nucleu, procesul `init` manipuleaza restul proceselor de boot-are.
- De obicei, aceasta include initializarea sistemului, lansarea unor servicii, lansarea programului de loginare.

- **Alocarea PID-ului**
- Implicit,nucleul impune o valoare maxima a PID-ului, valoarea 32768 (pentru compatibilitate cu sistemele Unix mai veichi care au functionat pe un procesor pe 16 biti).
- Administratorul de sistem poate seta valoarea maxima prin programul */proc/sys/kernel/pid_max.*
- Valorile PID se aloca secvential de catre sistemul de operare; cele alocate anterior nu pot fi reutilizate.
- **Ierarhia de procese**
- Fiecare proces are un tata, cu exceptia procesului `init` (procesul in interiorul caruia este declarat); procesul resp. este un proces fiu.
- Aceasta relatie este cuantificata prin identificatorul procesului părinte(PPID)
- Fiecare proces apartine unui *user* si unui *group*. Aceasta relatie este utilizata ptr. a controla drepturile de acces. Aceste valori sunt numere intregi corespunzatoare unor identificatori specificati in fisierele */etc/passwd* si */etc/group.*
- Fiecare proces fiu mosteneste *user*-ul si *group*-ul tatalui.
- De as., fiecare proces face parte dintr-un *grup* de *procese*, care exprima o relatie cu alte procese (de **exemplu**, doua procese legate prin pipe). Procesele fiu si tata apartin aceluiasi grup.
- Din perspectiva utliz., un grup de procese formeaza un *job*.

- **Tipul PID-ului** unui proces este `pid_t` care este definit in header-ul <*sys/types.h*>.
- **Obtinerea PID-ului si PPID-ului**. Apelul de sist. `getpid( )` returneaza PID in cadrul procesului. Sintaxa este:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

- Apelul de sist. `getppid( )` returneaza PID-ul tatalui in procesul care il invoca. Sintaxa este:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```

- **Exemplu:**

```
printf ("My pid=%d\n", getpid ( ));
printf ("Parent's pid=%d\n", getppid ( ));
```

Crearea unui proces (funcţia de sistem fork)

- În sistemul de operare Unix, un proces se creează prin apelul funcţiei de sistem `fork ().`
- Efectul acesteia este următorul: *se copiază imaginea procesului într-o zonă de memorie liberă, această copie fiind noul proces creat, în prima fază identic cu procesul* ***iniţial.***
- Cele două procese îşi continuă execuţia în paralel cu instrucţiunea care urmează apelului `fork`. Procesul nou creat poartă numele de *proces fiu,* iar procesul care a apelat funcţia `fork()` se numeşte *proces părinte.*
- Exceptând faptul că au spaţii de adrese diferite, procesul fiu diferă de procesul părinte doar prin identificatorul său de proces (PID), prin identificatorul procesului părinte (PPID) şi prin valoarea returnată de apelul `fork().`
- Funcţia `fork` returnează:

  - *pid* fiu în procesul părinte şi `0` în procesul fiu, în caz de succes;

  - `(-1)` în caz de eroare, iar **errno** indică eroarea apărută.

- Eşecul apelului *fork* poate să apară dacă:

  - nu există memorie suficientă pentru efectuarea copiei imaginii procesului părinte;

  - numărul total de procese pentru acel ID utilizator real depăşeşte limita stabilită de sistem;

  - există deja prea multe procese în sistem.
- Interfaţa funcţiei **fork** este:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- Un proces poate crea mai mulţi fii. Deoarece nu există nici o funcţie care să permită a determina *pid* – ul proceselor fiu, funcţia **fork** returnează procesului părinte *pid* – ul procesului fiu.
- Funcţia **fork** returnează `0` procesului fiu, deoarece un proces poate avea doar un singur proces părinte, care se poate afla prin apelul funcţiei **getppid()** (*pid* `0` este folosit de procesul *swapper*, deci nu poate fi *pid* – ul unui proces fiu!).

- Prin testarea valorii returnate de **fork,** se poate partiţiona codul programului în cele două procese, fiecare proces executând codul corespunzător lui:

```
swich  ( fork () )
{
    case – 1:
        perror ( “fork” );
        exit (1);
    case 0: /* proces fiu */
        //instrucţiuni
    default: /* proces părinte */
        //instrucţiuni
}
```

- **Terminarea unui proces.** Terminarea normală a unui proces de către el însuşi se realizează prin apelul funcţiei de sistem **exit** sau **_exit**. Interfaţa celor două funcţii de sistem este:

  ```
  void exit ( int  cod_exit );
  void _exit ( int cod_exit );
  ```

  unde: *cod_exit* este o valoare întreagă ce se returnează părintelui în vederea analizei.
- În urma execuţiei funcţiei de sistem **exit**, procesul respectiv intră în starea terminat. Funcţia **_exit** este apelată intern de către nucleu pentru terminarea unui proces când recepţionează semnale ce nu sunt tratate. În acest caz, **_exit** returnează un cuvânt de stare ce include şi numărul de identificare al semnalului.
- Sunt esenţiale trei cazuri:

  - procesul părinte se termină înaintea procesului fiu;

  - procesul fiu se termină înaintea procesului părinte;

  - procesul fiu, moştenit de procesul **init,** se termină.
- Procesul **init** devine părintele oricărui proces pentru care procesul părinte s-a terminat.

- Când un proces se termină, nucleul parcurge toate procesele active, pentru a vedea dacă printre ele există un proces care are ca părinte procesul terminat.
- Dacă există un astfel de proces, *pid*- ul procesului părinte devine 1 (*pid*- ul lui **init**). Nucleul garantează astfel că fiecare proces are un părinte.
- Dacă procesul fiu se termină înaintea procesului părinte, nucleul trebuie să păstreze anumite informaţii (*pid*, starea de terminare, timp de utilizare CPU) despre modul în care fiul s-a terminat.
- Aceste informaţii sunt accesibile părintelui prin apelul **wait** sau **waitpid**.
- În terminologie **Linux** un proces care s-a terminat şi pentru care procesul părinte nu a executat **wait** se numeşte "*zombie*".
- În această stare, procesul nu are nici un fel de resurse alocate, ci doar intrarea sa în tabela proceselor.
- Nucleul poate descărca toată memoria folosită de proces şi închide fişierele deschise.
- Un proces "*zombie*" se poate observa prin comanda **Linux** `ps`**,** care afişează la starea procesului litera 'Z'.

- **Aşteptarea unui proces (funcţiile wait, waitpid).**
- Când un proces se termină (normal sau anormal), procesul părinte este atenţionat de nucleu prin transmiterea semnalului SIGCLD. Acţiunea implicită ataşată acestui semnal este ignorarea sa.
- Un proces ce apelează **wait** sau **waitpid** poate:

  - să se blocheze (dacă toţi fiii săi sunt în execuţie);

  - să primească starea de terminare a fiului (dacă unul dintre fii s-a terminat);

  - să primească un mesaj de eroare (dacă nu are procese fiu).
- Interfaţa celor două funcţii **wait** şi **waitpid** este următoarea:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
pid_t wait (pid_t pid, int *status, int opt);
```

- *(ambele întorc pid, 0 (vezi **waitpid**) sau (-1) în caz de eroare)*

  status este un pointer spre locaţia din spaţiul de adrese al procesului unde sistemul depune informaţia de stare la terminarea unui proces fiu. Analizând informaţia de stare redată pe 16 biţi, se poate şti cum s-a terminat procesul fiu.

Fire de executie sub Linux

- Realizarea unui nou fir de executie se realizeaza prin:
  `pid=clone(function,stack_ptr,sharing_flags,arg);`
- Apelul creaza un nou fir de executie, fie in procesul curent, fie intr-un proces nou, depinzind de `sharing_flags.` Daca noul fir de executie este in procesul curent, el partajeaza spatiul de adrese cu firele de executie existente si orice scriere care urmeaza, a oricarui fir de executie, la orice octet din spatiul de adrese este vizibil imediat tuturor celorlalte fire de executie.
- Daca spatiul de adrese nu este partajat, atunci noul fir de executie primeste o copie exacta a spatiului de adrese, dar scrierile urmatoare nu sunt vizibile celor mai vechi. Aceste semantici sunt la fel cu cele din `fork`.
- In ambele cazuri noul fir isi incepe executia cu `function,` care este apelata cu parametrul unic `arg.` De asemenea, in ambele cazuri, noul fir de executie primeste noua proprie stiva privata, cu indicatorul de stiva initializat la `stack_ptr`.

- Parametrul `sharing_flags` este o harta de biti; sunt definiti cinci biti, fiecare bit controleaza un aspect al partajarii si fiecare dintre acestia poate fi setat independent.

  - `CLONE_VM` determina daca memoria virtuala (spatiul de adrese) este partajat cu firele de executie mai vechi sau este copiat. Daca este setat, se creaza noul fir de executie in cadrul procesului existent; in caz contrar, noul fir de executie primeste propriul spatiu de adrese (se creaza un nou proces).

  - `CLONE_FS` controleaza partajarea radacinii, a cataloagelor de lucru si a indicatorului `unmask.` Chiar daca noul fir de executie are propriul spatiu de adrese, daca acest bit este setat vechile fire de executie impart cataloagele de lucru cu cel nou. Sub Linux, un apel `chdir` facut de un fir de executie schimba catalogul curent pentru celelalte fire din procesul sau, dar niciodata pentru fire de executie din alte procese.

  - `CLONE_FILES` daca este setat, noul fir de executie va partaja descriptorii de fisier cu celelalte fire din acelasi proces.

  - `CLONE_PID` controleaza daca noul fir de executie primeste propriul PID sau partajeaza PID-ul parintelui. Aceasta optiune este necesara in timpul pornirii sistemului. Proceselor utilizator nu le este permis sa o activeaza.

Planificarea sub linux

- Sub Linux firele de executie se impart in 3 clase de prioritati:
- - FIFO in timp real;
- - rotatie(round-robin) in timp real;
- - cu partajarea timpului.
- Firele de executie FIFO in timp real au cea mai mare prioritate si nu pot fi fortate.
- Firele de executie in rotatie in timp real se executa dupa metoda round-robin.
- Fiecare fir de executie are o prioritate de executie. Valoarea implicita este 20, dar poate fi modificata folosind apelul de sistem `nice(valoare)` la valoarea `20-valoare.` Parametrul valoare este un numar intreg din intervalul `[-20 +19]`, deci prioritatile sunt in intervalul `1,40`.
- Intentia este de a face calitatea serviciilor proportionala cu prioritatea; firele de executie cu prioritate mai mare primesc un timp de raspuns mai rapid si un timp CPU mai mare.
- In plus fata de prioritate, fiecare fir de executie are o cuanta asociata. Cuanta reprezinta un numar de tacturi de ceas cit poate sa mai ruleze firul de executie.Ceasul merge implicit la 100 Hz, deci fiecare tact este de 10 msec., ceea ce se numeste **jiffy(moment).**

- Planificatorul foloseste prioritatea si cuanta dupa cum urmeaza.
- Intii calculeza cit de bun (goodness) este fiecare fir de executie pregatit, aplicind urmatoarele reguli:

```
if (class == real_time) goodness = 100 + priority;
if (class == timesharing && quantum > 0)
    goodness = quantum + priority;
if (class == timesharing && quantum == 0) g
    goodness = 0;
```

- Ambele clase de timp real se incadreaza in primul if. Fiecare astfel de proces primeste o valoare **goodness** mai mare decit a tuturor firelor de executie.
- Daca firului de executie care a rulat ultimul i-a mai ramas o parte din cuanta de timp, acesta primeste o valoare **goodness** mai mare, astfel incit la aceeasi prioritate, un astfel de proces sa fie executat, in conditiile in care se presupune ca procesul respectiv are paginile sale si blocurile din memoria ascunsa incarcate.

- Algoritmul de planificare se deruleaza astfel:

  1. se selecteaza firul cu valoarea **goodness** cea mai mare;

  2. in timpul executiei firului, cuanta sa este decrementata cu 1 la fiecare tact de ceas;

  3. CPU este luat firului de executie daca se intimpla oricare dintre conditiile urmatoare:

  -Cuanta sa ajunge la 0;

  -Firul de executie intra in starea de blocare;

  -Un fir de executie blocat anterior cu o valoare **goodness** mai mare devine pregatit.

  Cuantele proceselor tind sa devina 0. Cind firele de executie care intra intr-o operatie de I/O le mai ramine o parte din cuanta lor alocata, planificatorul reseteaza toate valorile **Quantum**, conform formulei:

  ```
  quantum = (quantum/2) + priority
  ```