Lucrarea de laborator Nr. 02

Procese. Stările unui proces. Execuția unui proces. Programarea concurentă, paralelă și distribuită. Crearea și Identificarea proceselor: funcţiile fork(), wait(), waitpid(), getpid(), getppid(), getuid(), geteuid(), getgid() şi getegid(). *Netcat* (utilitar de reţea) utilizat în comunicarea prin socket într-o arhitectură client-server.

Aplicații demonstrative:
1. Crearea, terminarea și identificarea unui proces.
2. Crearea de procese în lanț (process chains).
3. Crearea de procese în pieptene (process fans).
4. Netcat (utilitar de reţea) utilizat în comunicarea prin socket într-o arhitectură client-server..

Despre procese Unix şi legătura lor cu funcţia *fork()*

Un *program* este o secvență de cod stocată pe un suport.
O secvență de cod scrisă pe hârtie este un program. O secvență de cod stocată într-un fișier disc este tot un program. Secvența de cod este scrisă într-un limbaj de programare. În momentul când o secvență de cod este încărcată în memorie pentru a fi executată ea devine proces.

Un *proces* este o *instanță* a unui *program*
caracterizată în pricipal printr-un **număr unic de identificare al procesului** - PID (**P**rocess **ID**endification number), un **număr de identificare al procesului părinte**- PPID (**P**arent **P**rocess **ID**endification number) şi o **stare a procesului** - PS (**P**rocess **S**tate). De asemenea un proces are o anumită **prioritate PRI** în raport cu alte procese în dobândirea resurselor sistemului de calcul cu precădere procesorul (CPU)**.**

Un *proces* Unix este materializat într-o entitate de memorie cuprinzând:
- o zonă de cod executabil;
- o zonă de date;
- un context (variabile de mediu, pointeri la blocurile de date, etc).

} (un..... .....program)

Procesele sunt de două tipuri:
- *procese sistem* - create (lansate în execuţie) de către *sistemul de operare*
  Exemple de *procese sistem*:
  - *dispatches*
  - *init*
  - *daemons*
  - *cron*
  - ...etc...
- *procese utilizator* - create (lansate în execuţie) de către un *utilizator* în cadrul unei *sesiuni de lucru*. Când un *utilizator* intră în *sesiune de lucru* (*log in*), *nucleul sistemului de operare* startează un *shell* (*Bourne Shell*, *C Shell*, *Korn Shell* etc.) şi conectează *shell*-ul la terminalul utilizatorului (terminalul de la care s-a făcut *log in*-ul). După intrarea în *sesiune de lucru, shell-ul* crează un *proces copil* pentru orice comandă introdusă de *utilizator*.
  Exemple de *procese utilizator*:
  - comenzi *shell* (cel puţin un *proces* pe comandă);

- programe *utilizator* sau *utilitare*

Fiecare *proces,* la creare*,* primeşte din partea *sistemului de operare Unix* un *număr* unic *de identificare*: ***PID*** (***P****rocess* ***ID****entification number*). La terminarea *procesului* acest număr este eliberat, el putând fi alocat altui *proces*.

Fiecare *proces* aparţine unui *proprietar*: ***UID*** (***U****ser* ***ID****entification number*); *proprietarul* unui *proces* este cel care l-a creat, adică *utilizatorul*.

UID este un număr unic alocat de către *sistemul de operare UNIX* fiecărui *utilizator* în momentul definirii acestuia de către *administratorul de sistem*. *UID* este stocat în fişierul ***/etc/passwd***, gestionat de către *administrator*. Comanda **id** afişează acest număr:

$id ↵
uid=2114(student1) gid=1(other)

Procesele se află, unele cu altele, în relaţii de tip *părinte-copil.*

În afară de ***PID*** şi ***UID*** un *proces* mai posedă:

- *PPID -Parent PID- numărul procesului tată*;
- *TTY –TeleTYpe-* numele ecranului asociat;
- *Status* - starea *procesului* în memorie; poate lua valorile:
  - O Procesul este executat de către procesor.
  - S "Sleeping": procesul este în aşteptare unui eveniment.
  - R "Runnable": procesul este gata de execuţie, într-o coadă de aşteptare
  - I"Idle": procesul a fost creat
  - Z "Zombie state": procesul este terminat şi procesul părinte nu e în aşteaptare
  - T "Traced": procesul a fost stopat printr-un semnal
  - X "SXBRK state": procesul este în aşteptare, neavând suficientă memorie alocată
- *PRI –PRIority* - prioritatea *procesului*; este un număr întreg utilizat de sistem pentru acordarea de resurse *procesului* respectiv, proces aflat în concurenţă cu alte *procese* (valoare mare semnifică prioritate mică); o resursă esenţială pentru un *proces* este acordarea de *timp CPU* - timp de ocupare a procesorului;
- *STIME -StartTIME-* timpul de start al *procesului*;
- *SZ -SiZe-* mărimea în blocuri a *procesului;*
- *TIME* - timpul total real de execuţie al *procesului* (timpul de ocupare a procesorului);
- ...etc...

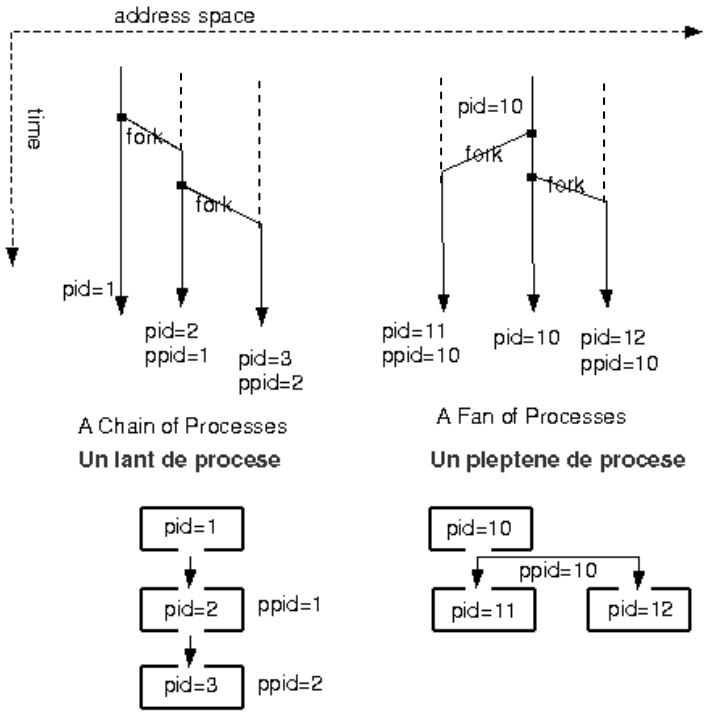
Un program poate fi instanţiat de mai multe ori, fiecare ***instanţă*** constituindu-se într-un *proces* separat. Fiecărui *proces* sistemul de operare îi alocă o zonă de memorie separată. Un *proces* poate "cere", prin apelul funcţiei sistem *fork*, crearea unui nou *proces.*

Un *fir de execuție* (*thread*) este *instanţa* unui *proces sau* părți ale acestuia,
procesul, după cum am văzut, fiind *instanţa* unui program (limbajul de programare Java este bine adaptat creării şi gestionării *thread*-urilor). *Thread*-urile se mai numesc şi *subprocese* sau *procese moi.*

Şi acum să revenim la *procese*. Cum am afirmat anterior, un *proces* poate crea un nou *proces*. *Procesul* care a creat un nou *proces* se numeşte *procesul părinte* (*parent*) sau pe scurt ***părinte sau tată***, iar un *proces* creat de un ***părinte*** se numeşte *proces copil* (*child*), sau pe scurt *copil*. Crearea şi gestionarea elementară a *proceselor* se poate face prin apelurile la funcţiile sistem ***fork***, ***wait, waitpid*** şi generic ***exec*** .

Un *proces* poate crea un nou *proces* printr-un apel al funcţiei sistem *fork*. *Procesul* creator se numeşte *părinte*, iar noul *proces* creat se numeşte *copil*. *Fork* –ul crează un *copil* care va executa o copie a *parintelui* aflată la o altă adresă decât cea a *părintelui.*

Figura de mai jos ilustrează două posibilităţi de utilizare a funcţiei *fork()*. Prima posibilitate construieşte un *lanţ de procese (a Chain of Processes)*, iar doua posibilitate construieşte un *pieptene de procese (a Fan of Processes)*.



Notă

(!)În UNIX în memoria calculatorului la un moment dat pot să coexiste mai multe *procese* ce constituite în fapt un sistem ierarhizat de *procese* bazat pe relaţii de tip *părinte-copil*. Aceste *procese* sunt într-o continuă competiţie unele cu altele pentru dobândirea resurselor sistemului (alocare memorie, timp procesor, dispozitive periferice, etc), cea mai importantă dintre resurse fiind timpul procesor (*proces* în curs de execuţie în procesor - asignarea procesorului). **În Windows** nu există conceptul de ierarhie de procese.

Din shell vizualizarea proceselor UNIX se poate face prin comanda *ps* (afişează informaţii despre *procese* şi ierarhia lor) ca în exemplele de mai jos:

.

$ps ↵ ⇒ afişează informaţii despre procesele startate de utilizator

```
  PID TTY          TIME CMD
 8089 pts/0    00:00:00 bash
 8094 pts/0    00:00:00 ps
```

$ps -l↵ ⇒ afişează o listă completă cu informaţii despre *procesele* startate de utilizator

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 42024  8089  8087  0  80   0 - 14478 wait   pts/0    00:00:00 bash
0 R 42024  8095  8089  0  80   0 -  1591 -      pts/0    00:00:00 ps
```

$*ps –ef* ↵ ⇒ afişează o listă (***-f***) despre toate *procesele* active(***-e***)

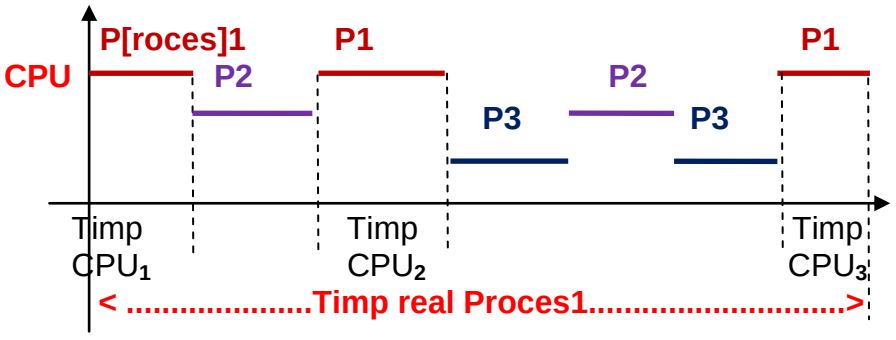
UID	***PID***	***PPID***	***C***	***STIME***	***TTY***	***COMD***
0	***0***	***0***			*?*	*swapper*
0	***1***	***0***			*?*	*init*
0	*2*	*0 .....*				
65536	***1893***	***1***			*pts/1*	*/bin/bash*
65536	***1888***	***1893***			***pts/1***	***ps –ef*** (linie de referinţă)

În memoria calculatorului, la un moment dat, pot să coexiste mai multe *procese,* ce constituite în fapt un *sistem ierarhizat de procese*. Aceste *procese* sunt într-o continuă competiţie unele cu altele pentru dobândirea resurselor sistemului (alocare memorie, timp procesor, dispozitive periferice, etc).

Sistemul de operare ***Unix*** gestionează un *sistem de procese* ierarhizat în conformitate cu o structură arborescentă inversată. Relaţiile dintre *procese* sunt relaţii de tip *părinte-copil.*

Execuţia unui proces Unix

Sistemul de operare UNIX este un sistem multitasking (multiprogramare). Multitakingul constă în posibilitatea unui sistem de calcul ca în loc ca *procesele* să se execute secvențial ele să se execute în același timp (în arhitecturi multiprocesor) sau aparent în același timp (în arhitectui monoprocesor). *Procesele* noi încep și întrerup cele procesele deja pornite înainte ca acestea să se încheie. Ca rezultat, un calculator execută segmente ale mai multor procese într-o manieră intercalată, accesând la un moment dat resursele comune de procesare cum ar fi procesorul (CPU) și memoria principală. Multitaskingul nu înseamnă neapărat că se execută mai multe procese în același timp (simultan). Cu alte cuvinte, multitaskingul nu presupune neapărat executarea paralelă pe mai multe procesoare astfel încât chiar și pe calculatoarele multiprocesor sau multicore, care au mai multe procesoare / nuclee, multitakingul permite executarea mai multor procese decât numărul de CPU-uri.

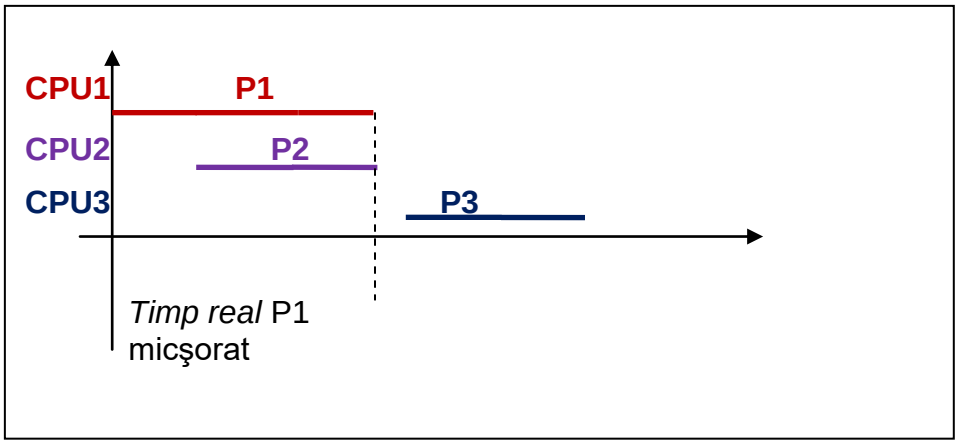


Timp CPU P1 = Timp virtual= Timp CPU_1 + Timp CPU_2 + Timp CPU_3

Multitasking- rulează mai multe procese. 3 în cazul de mai sus

Notă:

În cazul în care cele trei procese s-ar rula pe procesoare diferite sau pe calculatoare diferite legate în reţea atunci cele trei procese s-ar rula concurent, respectiv paralel sau distribuit şi per ansamblu s-ar produce a diminuare a *Timpului real.*



În **Programarea concurentă sau distribuită e**lementul esenţial care o deosebește de programarea **paralelă** este faptul ca *procesele* cooperează între ele în timpul execuţiei.

În **Programare paralelă** procesele paralele nu sunt condiţionate unul de celălalt, nu colaborează între ele, execuţia unuia nu este în nici un moment dependentă de rezultatele parţiale ale celuilalt.

Spunem că avem de-a face cu **programare concurentă respectiv distribuită** atunci atunci când procesele paralele se intercondiţionează reciproc. Termenii "programare paralelă" și " programare distribuită" referă în general un calculator multiprocesor, respectiv o rețea de calculatoare mono sau multiprocesor și au un grad mare de suprapunere. Același stil de programare poate fi caracterizat atât ca "paralel" cât și "distribuit"; iar procesoarele într-un sistem tipic distribuit pot rula simultan şi în paralel.

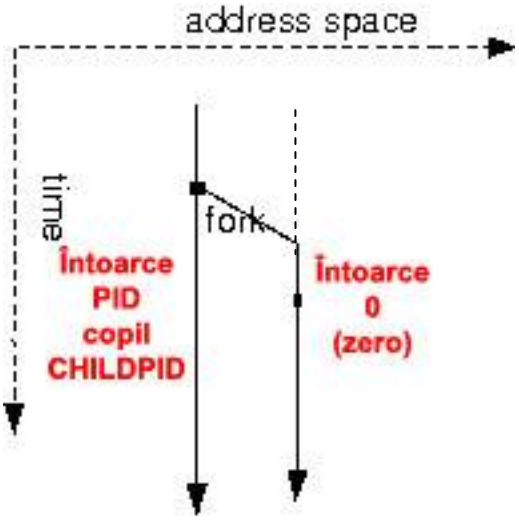
Creare proces: Funcţia *fork()*

Funcția fork() crează un nou proces, identic cu procesul apelant, dar în alt spațiu de memorie. Noul proces este referit sub numele de *copil*, iar procesul apelant este referit sub numele de *părinte*. Cele două procese, *părinte* și *copil*, rulează în spații de memorie diferite. Procesul *copil* rulază concurențial cu procesul *părinte*. După ce noul proces a fost creat, ambele procese vor executa următoare instrucțiuni care urmează apelului *fork()*. Procesul *copil* utilizează același *pc* (*program counter*), aceleiași registri CPU și aceleași fișiere deschise care sunt utilizate de procesul *părinte*.

Secvenţa de bază UNIX de apel al funcţiei sistem `fork()` este dată mai jos:

```
#include<sys/types.h>
#include<unistd.h>
        /**
        pid_t  este un tip de data  integer definită în sys/type.h
        **/
pid_t fork(void);
```

Un apel al funcţiei *fork* este procesat atât de *părinte* cât şi de *copil*. Un apel al funcţiei *fork* returnează:



- **0** dacă *procesul* care procesează apelul este *copilul*.
- **un întreg pozitiv lung** reprezentând PID-ul *copilului* dacă *procesul* care procesează apelul este *părintele*.
- **-1** dacă apelul funcţiei sistem *fork* a eşuat (nu s-a creat *copilul*)

Așteptare terminare proces: Funcţiile *wait()* și *waitpid()*

NUME

wait, waitpid - așteptă terminarea unui procesul copil.

APEL

#include <sys / wait.h>

pid_t wait (int * stat_loc);
pid_t waitpid (pid_t pid, int * stat_loc, opțiuni int);

DESCRIERE

Funcțiile wait () și waitpid () vor obține informații de stare (stat_loc) referitoare la unul dintre procesele copil ale apelantului.

Funcția wait () așteaptă terminarea oricărui proces copil și obține informații de stare a acestuia.

Funcția waitpid () așteaptă terminarea unui anumit proces copil (pid) și obține informații de stare a acestuia

Funcția wait () / waitpid () va face ca procesul de apelant să fie blocat până când informațiile de stare generate de terminarea procesului copil sunt puse la dispoziția procesului apelant, sau până la livrarea unui semnal a cărui acțiune este fie executarea unei funcții de captare a semnalului, fie încheierea procesului , sau apare o eroare. Dacă informațiile privind starea terminării sunt disponibile înainte de apelul de așteptare wait(), adică copilul s-a terminat, returnarea va fi imediată. Dacă informațiile privind starea terminării sunt disponibile pentru două sau mai multe procese secundare, ordinea în care este raportată starea lor nu este specificată.

VALOARE RETURNATĂ

Dacă wait () sau waitpid () returnează deoarece starea unui proces *copil* este disponibilă, adică procesul copil s-a terminat, aceste funcții vor returna o valoare egală cu PID-ul procesului procesului *copil* pentru care este raportată starea. Dacă wait () sau waitpid () returnează din cauza *livrării* unui semnal către procesul de apelare, -1 va fi returnat și *errno* setat la [EINTR]. Dacă waitpid () a fost invocat cu WNOHANG setat în opțiuni, are cel puțin un proces copil specificat de PID pentru care starea nu este disponibilă, sau starea nu este disponibilă pentru niciun proces specificat de PID, 0 este returnat. În caz contrar, -1 va fi returnat și *errno* setat pentru a indica eroarea.

Identificare proces: Funcţiile *getpid(), getppid*(), *getuid(), geteuid(),* getgid() şi getegid()

Fiecărui *proces*, la creare, sistemul de operare îi alocă un *PID*, adică un număr de identificare. *PID*-ul este un număr întreg pozitiv de tip int, care identifică în mod unic un *proces* (*proces* = *instanţa* unui program) în cadrul sistemului de operare. La terminarea *procesului PID*-ul este eliberat, el putând fi alocat altui *proces*. Există două apeluri sistem (*system calls*) prin care putem afla *PID*-ul *procesului părinte* (*PPID* - ParentPID), respectiv *PID*-ul *procesului* apelant (*PID* - ProcessID):

```
#include<sys/type.h>
#inlcude<unistd.h>
      /**
      pid_t  este o dată de tip int definită în sys/type.h
      **/
pid_t getppid(void);          // raspunde cu PPID
pid_t getpid(void);           // raspunde cu PID
```

UNIX asociază un *utilizator* (*user*) sau *proprietar* (*owner*) fiecărui *proces*. *Utilizatorul* sau *proprietarul* au anumite privilegii legate de gestiunea *procesului*. *Utilizatorul* este unic determint în sistem prin *UID* (User IDentification number), număr întreg pozitiv ce i-a fost alocat de către sistemul de operare în momentul creării sale de către administrator. Prin acest *UID* se face asocierea între *utilizator* şi *proces*. Exită de asemenea şi un *EUID* (Effective User IDentification number), număr întreg pozitiv prin care se identifică privilegiile *proceselor* în accesarea resurselor, ca de exemplu acces fişiere, partajare memorie şi semafoare. *UID* şi *EUID* pot fi accesate prin următoarele apelurile sistem (*system calls*): *getuid(),* respectiv *geteuid*():

```
#include<sys/types.h>
#include<unistd.h>
      /**
      uid_t  este o dată de tip unsigned int definită în sys/type.h
      **/
uid_t getuid(void);           // raspunde cu UID (real)
uid_t geteuid(void);          // raspunde cu EUID (efectiv)
```

În UNIX un *utilizator* (*user*) face parte dintrun *grup de utilizatori*. *Utilizatorii* care fac *parte din grupul de utilizator*i ai *proprietarului* procesului au anumite privilegii legate de gestiunea *procesului*. Grupul de u*tilizatori* este unic determinat în sistem prin *GID* (Group IDentification number), număr întreg pozitiv ce i-a fost alocat de către sistemul de operare în momentul creării grupului de către administrator. Se mai numește și GID real.

Există de asemenea şi un *EGID* (Effective Group IDentification number), număr întreg pozitiv prin care se identifică privilegiile *proceselor* în accesarea resurselor, ca de exemplu acces fişiere, partajare memorie şi semafoare. *GID* real și *EGID* pot fi accesate prin apelurile sistem (*system calls*) *getgid()* și *getegid()*:

```
#include<sys/types.h>
#include<unistd.h>
      /**
      gid_t  este o dată de tip unsignet int definită în sys/type.h
      **/
gid_t getgid(void);           // raspunde cu GID (real)
gid_t getegid(void);          // raspunde cu EGID (efectiv)
```

Program *fork_.c* - creare, terminare şi identificare proces

- Exemplul de mai jos ilustreaza o utilizare simpla a functiei *fork()* prin care se crează o structură de două procese în lanţ

```
/*****************************************************************************
* FISIER: fork_.c
*****************************************************************************/
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
// MACRO tipvar()  de determinare tip variabila
// Cuvantul cheie  _Generic  in C este utilizat pentru a defini un MACRO pentru diferitele tipuri de data.
// Aceast nou cuvant cheie a fost adaugat in limbajul de programare C o data cu  lansarea standardului C11.
#define tipvar(X) _Generic((X), int:"int", long:"long",short:"short",\
                long long:"long long", unsigned int:"unsigned int", default: "necunoscut")
int main(int argc, char **argv)
{
 /**
    pid_t este o data tip  integer definita in sys/types.h
    uid_t este o data tip integer definita in sys/types.h
    gid_t este o data tip integer definita in sys/types.h
**/
 pid_t pidt;
 uid_t uidt;
 gid_t gidt;


 pid_t childpid;

 printf("\nTipurile de data pid_t, uid_t si gid_t sunt definite in sys/types.h\n");

  // printf("%d\n", _Generic( 1.0L, float:1, double:2,
  //                       long double: 3, default:0));  // exemplu utilizare _Generic()

 printf("\tpid_t ---  size %d \t type %s\n", sizeof(getpid()), tipvar(pidt));
 printf("\tuid_t ---  size %d \t type %s\n", sizeof(getuid()), tipvar(uidt));
 printf("\tgid_t ---  size %d \t type %s\n", sizeof(getgid()), tipvar(gidt));

 if (( childpid = fork()) == 0 )    // creare proces copil
                                    // raspunsul functiei fork()
                                    // =0 proces copil
                                    // >0 proces parinte (PID proces copil creat)
                                   // =-1 fork() esuat
 {
  /* COD PROCES COPIL RULAT IN SPATIUL DE ADRESE COPIL */
                                  // \n = enter
                                  // \t = tab
                                 // \ = continuare instr. pe linia urmatoare
  printf("\nSunt copilul, \
        \n\tam ProcessID (PID) = %d, parintele meu are  ParentPID (PPID) =%d, \
        \n\tCopilul meu are ChildPID = %d, \
        \n\tProprietarul meu  (UID) este=%u, \
        \n\tUtilizator efectiv (EUID) este: %u, \
        \n\tGrupul din care fac parte (GID) este: %u, \
        \n\tGrup efectiv (EGID)= %u\n", \
```

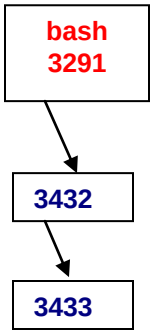
```
            (int) getpid(), (int) getppid(), (int)childpid, \
            (unsigned int) getuid(), (unsigned int) geteuid(), \
            (unsigned int) getgid(), (unsigned int) getegid());  // a se urmari tipul de data comparativ la parinte
    } else if (childpid > 0) {
      /* COD PROCES PARINTE RULAT IN SPATIUL DE ADRESE PARINTE */
      printf("\nSunt parintele, \
            \n\tam ProcessID (PID) = %d, parintele meu are ParentPID (PPID) = %d, \
            \n\tCopilul meu are  ChildPID=%d, \
            \n\tProprietarul meu (UID) este=%u, \
            \n\tUtilizator efectiv (EUID) este: %u, \
            \n\tGrupul din care fac parte (GID) este: %u, \
            \n\tGrup efectiv (EGID)= %u \n", \
           (pid_t) getpid(), (pid_t) getppid(),  childpid, \
           (uid_t) getuid(), (uid_t) geteuid(), \
           (gid_t) getgid(),(gid_t) getegid());   // a se urmari tipul de data comparativ la copil
    }
    /* cod comun proces parinte/copil*/
    /* rulat insa in spatii de memorie diferite: spatiul de adrese parinte respectiv copil */
    printf("\n\tCod rulat in parinte/copil.  PID= %d , PPID= %d  --- terminat\n", \
          (pid_t) getpid(), (pid_t) getppid());
    return 0;      // term proces parinte/copil
   }
```

Compilare linkeditare şi rulare program fork_.c

```
$gcc   -o   fork_   fork_.c

$./fork_
```

Mesaje afişate în urma rulării



Tipurile de data pid_t, uid_t si gid_t sunt definite in sys/types.h
pid_t --- size 4   type int
uid_t --- size 4   type unsigned int
gid_t --- size 4   type unsigned int

Sunt parintele,
am ProcessID (PID) = **3432**, parintele meu are ParentPID (PPID) = **3291**,
Copilul meu are ChildPID=**3433**,
Proprietarul meu (UID) este=**1000**,
Utilizator efectiv (EUID) este: **1000**,
Grupul din care fac parte (GID) este: **1000**,
Grup efectiv (EGID)= **1000**

Cod rulat in parinte/copil. PID= **3432** , PPID= **3291** --- terminat

Sunt copilul,
am ProcessID (PID) = **3433**, parintele meu are ParentPID (PPID) =**3432**,
Valoare transmisa ChildPID = **0**,
Proprietarul meu (UID) este=**1000**,
Utilizator efectiv (EUID) este: **1000**,
Grupul din care fac parte (GID) este: **1000**,
Grup efectiv (EGID)= **1000**

Cod rulat in parinte/copil. PID= **3433** , PPID= **3432** --- terminat

Observatii:

1. Răspunsul funcţiei fork apelată din procesul părinte este ChildPID=3433
2. Răspunsul funcţiei fork apelată din procesul copil este ChildPID=0
3. Părintele (ParentPID=3291 procesului părinte este shell-ul *(bash*). Se dă pentru confirmare comanda:

```
$ps
   PID TTY          TIME CMD
 3291 pts/0    00:00:00 bash
 3456 pts/0    00:00:00 ps
```

4. Proprietarul procesului este (UID= 1000) care este login utilizator şi poate fi aflat cu comanda:

```
$ id
uid=1000(streian) gid=1000(cadre) …
```

Program *process_chains_0.c* - Creare de *procese în lanţ*

- Exemplu de mai jos crează un “lanţ de 3 procese”. Înainte de terminare fiecare proces îşi afişează numărul de creare i, identificatorul de proces (Process ID) – PID şi identificatorul procesului părinte (parent process ID) – PPID.
- În acest exemplu nu există aşteptări asfel încât un proces părinte poate să se terminte (exit - return) înaintea procesului copil (creat de părinte). Acest lucru cauzează adoptarea procesului copil de către procesul init care are PID-ul =1. Cu alte cuvinte în unele cazuri se va constata că tatăl unui proces copil are PPID=1.

```
/*****************************
* FISIER: process_chains_0.c
*****************************/
    #include<stdio.h>
    #include<sys/types.h>
    #include<unistd.h>

    int main()
    {
            int i=0;            // i= numar proces creat
            int n=4;            // n-1= numar procese din lant
            pid_t childpid;
            printf("\n%i Acesta este procesul initial cu PID=%ld care are parintele PPID=%ld\n", i, (long) getpid(),
            (long)getppid());

            for(i=1; i<n; ++i) if (childpid = fork()) break; // in parinte terminare fortata bucla for
            else{
                    /* secventa proces copil */
                    if(i == 1)sleep(1);   // !!! intarziere pt. ca parintele sa se termine inaintea copilului
                    printf("\n%i Acesta este procesul creat cu PID=%ld care are parintele PPID=%ld\n", i, (long)
            getpid(), (long)getppid());
```

```
        }
        /* secventa comuna proces parinte/copil */
        return 0;
}
```

Compilare linkeditare şi rulare program process_chains_0.c

$gcc -o process_chains_0 process_chains_0.c

$./process_chains_0¶

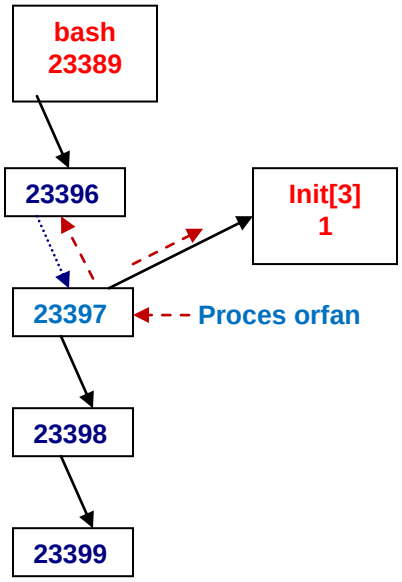
Mesaje afişate în urma rulării

0 Acesta este procesul initial cu PID=**23396** care are parintele PPID=**23389**

1 Acesta este procesul creat cu PID=**23397** care are parintele PPID=**1**

2 Acesta este procesul creat cu PID=**23398** care are parintele PPID=23397

3 Acesta este procesul creat cu PID=**23399** care are parintele PPID=23398



Observatii:

1. Procesul cu PID=**23397** este orfan (PPID=1). Dacă procesul nu ar fi rămas orfan, atunci ar fi fost PPID=**23396**
2. Procesul *init* are PID=1. Se dă pentru confirmare comanda (*more* este pentru oprirea afişării dupa 24 de rânduri):

```
$ps¶
  PID TTY          TIME CMD
27281 pts/1    00:00:00 ps
23389 pts/1    00:00:00 bash

$ps –ef|more¶
UID        PID  PPID  C    STIME        TTY          TIME         CMD
root       1    0     2    08:48        ?            00:00:03     init[3]
...etc...
```

3. Linia *if(i == 1)sleep(1);* întârzie în mod deliberat procesul nr.1 creat astfel încât el să devină orfan (procesul părinte se termină sigur înaitea procesului copil). Explicaţie: procesul a devenit orfan deoarece nu este prevăzută aşteptarea părintelui până la terminarea copilului. Se scoate această linie şi atunci mesajele afişate în urma rulării ar putea arăta astfel (posibil să apară şi în această situaţie procese orfane- nu există aşteptări):

0 Acesta este procesul initialcu PID=**23396** care are parintele PPID=**23389**

1 Acesta este procesul creat cu PID=**23397** care are parintele PPID=23396

2 Acesta este procesul creat cu PID=**23398** care are parintele PPID=23397

3 Acesta este procesul creat cu PID=**23399** care are parintele PPID=23398

Program *process_fans.c* - Creare de *procese în pieptene*

- Exemplu de mai jos crează un "pieptene de n procese", unde n este un argument introdus pe linia de comandă. Înainte de terminare fiecare proces copil îşi afişează numărul de creare i, identificatorul de proces (Process ID) – PID şi identificatorul procesului părinte (parent process ID) – PPID.
- În acest exemplu există aşteptări asfel încât procesul părinte nu poate să se terminte (exit - return) înaintea procesului copil, deci nu apare cazul în care procesul copil va fi adoptat către procesul init care are PID-ul =1. Cu alte cuvinte nu se va constata că tatăl unui proces copil are PPID=1.

```
// rulat in Oracle Solaris 11

/* process_fans.c
Programul creaza procese in evantai/pieptene (fans process)
*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>

        int main(int argc, char **argv)
        {

        int i;                          // i= numar proces creat
        int n;                          // n= numar procese din pieptene
        pid_t childpid;    // memoreaza PID proces copil creat de un parinte
        int status;                     // memoreaza starea unui proces; se utilizeaza la wait()

        if(argc != 2){                  // test daca s-a introdus pe linia de comanda nr. de procese de creat
                fprintf(stderr, "Utilizare: %s nr_procese \n", argv[0]);
                exit(1);
        }

        n=atoi(argv[1]);  // conversie ascii – intreg pt. nr. de procese
        childpid=7777;   // ceva pozitiv, nu are importanta ce

        fprintf(stderr, "acesta este procesul PARINTE %ld \n", (long)getpid());
        fflush(stderr);


/* for i=1, ...       se executa numai in parinte.
In copil break - exit
*/
        for(i=1;i<=n;++i)
        {
                if((childpid=fork())==0)
                {
                /* secventa proces copil */
                /* daca este proces copil sau esec la functia fork() --> break si exit(0) */
```

```
                fprintf(stderr, "acesta este procesul COPIL[%d]  %ld cu parintele %ld\n", i, (long)getpid(), (long)getppid());
                fflush(stderr);

                break;          // ajunge la exit = terminare proces copil
        }
        else
        {
                /* secventa proces parinte */
                        // se pune linia de mai jos pe comentariu si vom vedea ca
                        // apar copii orfani care au PID parinte egal cu 1
                while (wait(&status) != childpid){;};       /* empty: se asteapta terminarea copilului */
                                                            // for-ul va fi reluat pt. i=i+1 in parinte si se va crea un nou proces
                                                            copil
        };      // end if
        };      // end for, i = i+1 ---> urmatorul fork
        exit(0);
}
```

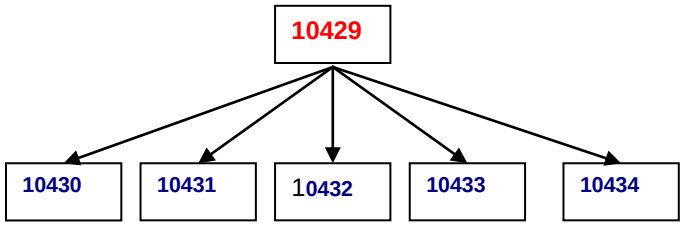
Compilare linkeditare şi rulare program process_fans.c

$gcc -o process_fans process_fans.c¶

$./process_fans **5**¶

Mesaje afişate în urma rulării

acesta este procesul PARINTE **10429**
acesta este procesul COPIL[1] **10430** cu parintele 10429
acesta este procesul COPIL[2] **10431** cu parintele 10429
acesta este procesul COPIL[3] **10432** cu parintele 10429
acesta este procesul COPIL[4] **10433** cu parintele 10429
acesta este procesul COPIL[5] **10434** cu parintele 10429



• Utilitarul *netcat*

Netcat, sau *nc* în unele implementări UNIX, este un utilitar de rețea utilizat în depanarea și investigarea rețelei. *Netcat* citește și scrie date în conexiuni de rețea utilizând protocolul TCP/IP sau UDP. *Netcat* este conceput pentru a fi un instrument de *back-end* de încredere care poate fi utilizat direct și ușor de alte programe și scripturi. În același timp, este un instrument de depanare și explorare a rețelei, bogat în caracteristici, deoarece poate crea aproape orice fel de conexiune de care este nevoie și are câteva capabilități încorporate interesante.

Exemple de utilizare a utilitarului netcat
Se deschid două ferestre terminal numite astfel:

- Ferestra Server
- Ferestra Client:

Netcat utilizat în comunicarea prin socket într-o arhitectură server-client

Virgiliu Streian. Stările unui proces. Execuția unui proces. Programarea concurentă, paralelă și distribuită. Procese. crearea și Identificarea proceselor: funcţiile fork(), getpid(), getppid(), getuid(), geteuid(), getgid() şi getegid(). Netcat (utilitar de reţea) utilizat în comunicarea prin socket într-o arhitectură client-server.

Protocol TCP

Ferestra Server :

Netcat rulat în modul server TCP pe un port specificat (5500 în cazul de față) care asculta conexiunile de intrare (protocol TCP).

```
$ nmap localhost                ⇒ Constatarea că portul 5500 este liber
                                  porturile ocupate sunt: 22, 25, 111, 443
      Starting Nmap 4.53 ( http://insecure.org ) at 2021-30-10 10:39 EET
      Interesting ports on localhost (127.0.0.1):
      Not shown: 1710 closed ports
      PORT    STATE SERVICE
      22/tcp  open  ssh
      25/tcp  open  smtp
      111/tcp open  rpcbind
      443/tcp open  https

$ netstat -an | grep LISTEN     ⇒ afișeaza detalii despre conexiunile active

      tcp     0      0 0.0.0.0:44194         0.0.0.0:*           LISTEN
      tcp     0      0 0.0.0.0:111           0.0.0.0:*           LISTEN
      tcp     0      0 0.0.0.0:4949          0.0.0.0:*           LISTEN
      tcp     0      0 0.0.0.0:22            0.0.0.0:*           LISTEN
      tcp     0      0 127.0.0.1:25          0.0.0.0:*           LISTEN
      tcp     0      0 0.0.0.0:443           0.0.0.0:*           LISTEN
      tcp6    0      0 :::22                 :::*                LISTEN
      tcp6    0      0 :::443                :::*                LISTEN
```

Ferestra Client :

Netcat rulat în modul client TCP conectat la un server pe un port specificat (localhost și 5500 în cazul de față).

```
$ netcat   localhost   5500     ⇒ client TCP, port 5500
```

Acum dacă scriem text în Ferestra Client el va aparea în Fereastra Server și viceversa ca mai jos:

Ferestra Client :

```
$ netcat   localhost   5500     ⇒ comandă dată anterior în fereastră
12345
qwerty
<ctrl>/c                        ⇒ terminare
$
```

Ferestra Server :

```
$ netcat   -lp   5500     ⇒ comandă dată anterior în fereastră

12345
qwerty
$
```

Protocol UDP

Ferestra Server :

Netcat rulat în modul server UDP pe un port specificat (6000 în cazul de față) (protocol UDP).

```
$ netcat   -lup   6000
```

Ferestra Client :

Netcat rulat în modul client UDP conectat la un server pe un port specificat (localhost și 6000 în cazul de față).

```
$ netcat   -u  localhost   6000
```

Acum dacă scriem text în Ferestra Client el va aparea în Fereastra Server și viceversa.