**Ce este un proces ?**

În Linux, un **proces** reprezintă o instanță a unui program în execuție. Este o colecție de unul sau mai multe fire de execuție (threads) care execută codul programului, împreună cu resursele asociate cum ar fi memoria, descriptorii de fișiere și informațiile de securitate. Fiecare proces are un ID unic, cunoscut sub numele de PID (Process ID), prin care sistemul de operare îl identifică și gestionează.Un proces Unix are următoarele caracteristici:

1. Identificator de proces (PID): Fiecare proces are un PID unic care îl identifică în cadrul sistemului de operare.
2. Spațiu de adresă: Fiecare proces are propriul său spațiu de adresă virtual, care conține codul programului, datele și stiva sa de execuție.
3. Context de execuție: Un proces păstrează informații despre starea sa de execuție, cum ar fi valorile registrilor, counterul programului și alte informații necesare pentru a relua execuția din acel punct.
4. Prioritate: Fiecare proces are o prioritate asociată, care determină cât timp de procesor primește în comparație cu alte procese.
5. Controlul resurselor: Un proces poate accesa și utiliza resursele sistemului, cum ar fi fișiere, dispozitive de intrare/ieșire, memoria și alte resurse.
6. Comunicare interproces: Procesele pot comunica între ele utilizând diferite mecanisme, cum ar fi pipe-uri, cozi de mesaje, semafoare, memorie partajată sau socket-uri.
7. Ciclu de viață: Un proces poate fi creat, rulat, întrerupt, reluat, suspendat și în cele din urmă terminat. Ciclul de viață al unui proces poate fi influențat de evenimente interne sau externe.

Procesele Unix sunt esențiale în sistemul de operare, deoarece permit rularea și gestionarea simultană a mai multor programe și sarcini. Sistemul de operare acordă resursele necesare fiecărui proces și gestionează planificarea și sincronizarea lor pentru a asigura un mediu de execuție eficient și sigur.

Procesele pot fi legate între ele prin relații de tip părinte-copil, unde procesul părinte poate controla execuția și resursele proceselor sale copil. De asemenea, procesele pot comunica între ele prin diverse mecanisme de IPC (Inter-Process Communication), cum ar fi semnale, pipe-uri, socket-uri, etc.

**Cum vedem procesele in unix ?**

* **ps -e** : ne raporteaza procesele in unix **-e**  specificand ca ne arata toate procesele indiferent de user-ul care le-a initiat
  + PID (Identificatorul de proces): Numărul unic atribuit fiecărui proces.
  + TTY (Terminalul asociat procesului, dacă este cazul).

TIME (Timpul total de procesor utilizat de proces).

* + CMD (Comanda sau numele programului care a inițiat procesul).
* **ps -f -p 1** : ne da informatii despre procesul cu id =1. Parametrii reprezinta
  + "-f": Opțiunea "-f" specifică faptul că se va afișa un format extins al informațiilor despre procese, inclusiv informații detaliate despre procesul inițiat.
  + "-p 1": Opțiunea "-p" este folosită pentru a specifica PID-ul unui proces anume. În acest caz, PID-ul specificat este 1, care este PID-ul procesului init.

Informatii returnate

Astfel, atunci când rulezi comanda "ps -f -p 1" într-un terminal Unix, aceasta va afișa informații detaliate despre procesul init, care este primul proces creat în sistemul de operare și are un rol fundamental în pornirea și gestionarea altor procese.

* + UID (User ID): ID-ul utilizatorului asociat procesului.
  + PID (Identificatorul de proces): Numărul unic atribuit procesului init.
  + PPID (Parent PID): PID-ul procesului părinte al procesului init (de obicei, 0, deoarece init este procesul părinte al tuturor celorlalte procese).
  + C (CPU utilization): Utilizarea medie a procesorului de către procesul init.
  + STIME (Start time): Timpul de pornire al procesului init.
  + TTY (Terminalul asociat procesului init).
  + TIME (CPU time): Timpul total de procesor utilizat de procesul init.
  + CMD (Comanda sau numele programului care a inițiat procesul init).
* **ps -F -u stefan :** atunci când se ruleaza comanda "ps -F -u stefan" într-un terminal Unix, aceasta va afișa informații detaliate despre toate procesele care rulează și care sunt asociate utilizatorului "stefan".
  + UID (User ID): ID-ul utilizatorului asociat procesului.
  + PID (Identificatorul de proces): Numărul unic atribuit fiecărui proces.
  + PPID (Parent PID): PID-ul procesului părinte al procesului.
  + C (CPU utilization): Utilizarea medie a procesorului de către proces.
  + STIME (Start time): Timpul de pornire al procesului.
  + TTY (Terminalul asociat procesului).
  + TIME (CPU time): Timpul total de procesor utilizat de proces.
  + CMD (Comanda sau numele programului care a inițiat procesul).

O sa cream un process si o sa vedem cum arata acest process. Vom crea un process parinte si un process copilcare vor rula 30 de secunde fiecare. Dupa aceea ne vom uita in procese si vom vedea ce se intampla

**Functia fork()**

Funcția fork() este o funcție din limbajul C care este utilizată pentru a crea un proces nou prin duplicarea procesului curent. Procesul rezultat este o copie identică a procesului părinte, cunoscut și ca procesul copil.

Atunci când se apelează funcția fork(), procesul curent este duplicat și se crează un nou proces copil. Acest nou proces copil are aceeași stare, cod și resurse ca și procesul părinte la momentul apelului fork(). Cu toate acestea, cele două procese rulează în spații de adrese de memorie separate.

După apelul funcției fork(), cele două procese (procesul părinte și procesul copil) rulează în paralel și au PID-uri diferite. Procesul părinte primește PID-ul procesului copil în urma apelului fork(), în timp ce procesul copil primește valoarea 0 ca rezultat al apelului fork().

Funcția fork() întoarce rezultatele următoare:

Dacă rezultatul este negativ (-1), înseamnă că apelul fork() a eșuat și nu a fost creat niciun proces copil.

Dacă rezultatul este 0, înseamnă că ne aflăm în procesul copil.

Dacă rezultatul este un număr pozitiv, reprezintă PID-ul procesului copil și ne aflăm în procesul părinte.

Funcția fork() este utilă în crearea proceselor paralele, în care procesul părinte poate crea un proces copil care să execute anumite sarcini sau operații separate, cum ar fi execuția unui program nou, manipularea resurselor sau realizarea de calcule intensive.

Primul exemplu:

Tipul de date **pid\_t** este utilizat în sistemele de operare Unix și Linux pentru a reprezenta identificatorii de procese (process identifiers, sau PID). este de obicei implementat ca un întreg (integer). În majoritatea sistemelor, acesta este de obicei un întreg pe 32 de biți, dar implementările pot varia în funcție de arhitectura sistemului și de specificațiile platformei.

Variabila **pid** la primul print are valoarea zero deoarece a fost doar declara si reprezentarea fiind un int pe 32 by default este zero

In momentul in care fac fork-ul pid va fi initializat iar in procesul parinte va pastra id-ul procesului copil iar in procesul copil va fi zero

Exemplul 2. Ne propunem sa facem un fork care sa dea failed

Pentru asta prima data vom reduce numarul maxim de procese ce pot fi folosite

Comanda **ulimit -u 20** ne ajuta sa facem asta

Pentru a vedea restrictiile care le avem putem folosi **ulimit -a**

Ulimit restrictioneaza resursele pentru sesiunea curenta de shell

**Explicatie create\_30s\_child\_process.c**

**Funcționalitate**

Codul creează un proces nou prin apelul funcției fork() și împarte execuția în două ramuri diferite, pentru procesul copil și procesul parinte.

Procesul copil rulează un cod specific în interiorul ramurii if (pid == 0), în timp ce procesul parinte rulează în ramura else.

Atât procesul copil, cât și procesul parinte rulează timp de 30 de secunde în fundal, utilizând funcția sleep().

**Procesul copil**

Procesul copil afișează mesajul "Procesul copil rulează timp de 30 de secunde.".

Utilizează sleep(30) pentru a aștepta 30 de secunde înainte de a afișa mesajul "Procesul copil s-a încheiat.".

După aceea, procesul copil se încheie cu return 0 implicit.

**Procesul parinte**

Procesul parinte afișează mesajul "Procesul parinte rulează timp de 30 de secunde.".

Utilizează fflush(stdout) pentru a elibera bufferul de ieșire și a asigura afișarea mesajului înainte de apelul funcției sleep().

Utilizează sleep(30) pentru a aștepta 30 de secunde înainte de a afișa mesajul "Procesul parinte s-a încheiat.".

După aceea, procesul parinte se încheie cu return 0 implicit.

La un moment de timp vom vedea un defazaj intre parinte si copil. Aceasta se datoreaza modului in care sistemul de operare folosesti resursele. **Sleep()** suspenda procesul pentru numarul de secunde specificat insa nu garanteaza ca procesul va fi wakeup imediat dup ace a trecut timpul

**Explicatie fork\_2.c**

Înainte de a intra în bucla switch, este creat un proces copil folosind funcția fork(). Rezultatul apelului fork() este stocat în variabila pid. Aceasta poate avea una din următoarele valori:

perror funcția **perror()** în C este utilizată pentru a afișa un mesaj descriptiv în stderr, care explică ultima eroare întâlnită de un apel al unei funcții a sistemului de operare. Acest mesaj este util pentru a înțelege de ce o anumită operațiune a eșuat, cum ar fi un eșec al apelului **open()**, **fork()**, **pipe()** sau oricare alte funcții care interacționează cu sistemul. . Pentru intelegere avem un exemplu simplu **perror\_sample.c**

-1 - indica eșecul apelului fork().

0 - indica faptul că suntem în procesul copil.

un număr pozitiv - reprezintă PID-ul procesului copil și suntem în procesul părinte.

Dacă rezultatul apelului fork() este -1, acesta indică că apelul a eșuat și se afișează un mesaj de eroare folosind funcția perror(). Apoi, procesul curent se încheie cu exit(EXIT\_FAILURE).

Dacă rezultatul apelului fork() este 0, acesta indică că suntem în interiorul procesului copil. În acest caz, se afișează un mesaj specific procesului copil, inclusiv PID-ul acestuia (getpid()) și PID-ul procesului părinte (getppid()). Apoi, procesul copil se încheie cu exit(EXIT\_SUCCESS).

Dacă rezultatul apelului fork() este un număr pozitiv, acesta indică că suntem în interiorul procesului părinte. În acest caz, se afișează un mesaj specific procesului părinte, inclusiv PID-ul acestuia (getpid()) și PID-ul copilului (pid).

La final, procesul curent (atât părintele cât și copilul) se încheie cu return 0

**Diferenta intre return 0 si exit(EXIT\_SUCCESS)**

**exit(EXIT\_SUCCESS):**

exit() este o funcție din biblioteca standard C (stdlib.h) care este utilizată pentru a încheia execuția programului și a returna un cod de ieșire specificat.

EXIT\_SUCCESS este o macrodefiniție definită în stdlib.h și, de obicei, are valoarea 0, care reprezintă codul de ieșire asociat cu finalizarea cu succes a unui program.

Apelul exit(EXIT\_SUCCESS) înseamnă că programul se încheie imediat și se returnează codul de ieșire 0, indicând o finalizare cu succes.

De obicei, exit(EXIT\_SUCCESS) este utilizat atunci când programul dorește să se încheie explicit și să indice că a ajuns la un punct de finalizare dorit.

**return 0:**

return este o instrucțiune utilizată într-o funcție pentru a returna o valoare la locul de apel.

0 este o valoare întoarsă de return și reprezintă o finalizare cu succes a funcției respective.

Utilizarea return 0 în funcția main() indică că programul a fost finalizat cu succes și funcția main() se încheie, ceea ce duce implicit la încheierea programului.

De obicei, return 0 este utilizat în main() pentru a indica o finalizare cu succes și încheierea programului într-un mod normal.

În practică, ambele moduri (exit(EXIT\_SUCCESS) și return 0) indică faptul că programul a finalizat cu succes și că codul de ieșire este 0. Diferența principală este că exit() se folosește pentru a încheia programul din orice funcție, în timp ce return este utilizat pentru a returna o valoare specifică în funcția main().

**Explicatie fork\_3**

Codul de mai sus demonstrează crearea unui proces copil care devine un proces ORPHAN (orfan).

Înainte de a intra în bucla if, este creat un proces copil folosind funcția fork(). Rezultatul apelului fork() este stocat în variabila pid. Aceasta poate avea una din următoarele valori:

-1 - indica eșecul apelului fork().

0 - indica faptul că suntem în procesul copil.

un număr pozitiv - reprezintă PID-ul procesului copil și suntem în procesul părinte.

Dacă rezultatul apelului fork() este -1, acesta indică că apelul a eșuat și se afișează un mesaj de eroare folosind funcția perror(). Apoi, procesul curent se încheie cu exit(EXIT\_FAILURE).

Dacă rezultatul apelului fork() este 0, acesta indică că suntem în interiorul procesului copil. În acest caz, se afișează mesaje specifice procesului copil, inclusiv PID-ul acestuia (getpid()) și PID-ul procesului părinte (getppid()).

În procesul copil, se așteaptă timp de 5 secunde utilizând sleep(5). În acest timp, procesul părinte poate să se încheie.

După cele 5 secunde de așteptare, procesul copil afișează mesaje pentru a indica că a devenit un proces ORPHAN (orfan). Asta înseamnă că procesul părinte nu mai există, fie pentru că s-a încheiat, fie pentru că a terminat executarea fără a aștepta terminarea copilului.

Procesul copil se încheie cu exit(EXIT\_SUCCESS) pentru a indica o final

**Functiile wait() waitpid()**

Funcția wait() este o funcție din biblioteca standard C (sys/wait.h) utilizată în procesele părinte pentru a aștepta terminarea unuia sau mai multor procese copil

Funcția waitpid() este o funcție din biblioteca standard C (sys/wait.h) utilizată pentru a aștepta terminarea unui anumit proces copil. Diferența principală între waitpid() și wait() constă în faptul că waitpid() permite specificarea unui proces copil anume pe care părintele îl așteaptă să se încheie, în timp ce wait() așteaptă orice proces copil să se încheie

**Explicatie wait\_sample.c**

În acest exemplu, procesul părinte creează un proces copil folosind fork(). Procesul copil afișează mesaje specifice și așteaptă timp de 3 secunde utilizând sleep(3) pentru a simula efectuarea unui lucru. După aceasta, procesul copil se încheie cu exit(EXIT\_SUCCESS).

În procesul părinte, se afișează mesaje specifice și se utilizează funcția wait(NULL) pentru a aștepta terminarea oricărui proces copil. Acest apel de funcție blochează procesul părinte până când procesul copil se încheie. După terminarea procesului copil, se afișează un mesaj pentru a indica acest lucru.

**Explicatie waitpid\_sample.c**

În acest exemplu, procesul părinte creează un proces copil folosind fork(). Procesul copil afișează mesaje specifice și așteaptă timp de 3 secunde utilizând sleep(3) pentru a simula efectuarea unui lucru. După aceasta, procesul copil se încheie cu exit(EXIT\_SUCCESS).

În procesul părinte, se afișează mesaje specifice și se utilizează funcția waitpid() pentru a aștepta terminarea procesului copil cu PID-ul specificat (pid). Acest apel de funcție blochează procesul părinte până când procesul copil cu PID-ul specificat se încheie. După terminarea procesului copil, se afișează un mesaj pentru a indica acest lucru.

**Functia signal()**

Funcția signal() este o funcție din biblioteca standard C (<signal.h>) utilizată pentru a atribui un handler (un tratator) unei anumite semnale în cadrul unui program. Semnalele sunt mecanisme prin care sistemul de operare sau alte procese pot comunica cu un program în execuție.

Iată o explicație detaliată a funcționării funcției signal():

Prototipul funcției signal() este void (\*signal(int signum, void (\*handler)(int)))(int), unde:

signum reprezintă numărul semnalului pentru care se dorește atribuirea unui handler. Aceasta poate fi o constantă predefinită pentru semnale, cum ar fi SIGINT, SIGTERM, SIGUSR1, etc., sau poate fi un număr întreg reprezentând un semnal specific.

handler reprezintă pointer-ul către funcția care va fi apelată (handler-ul) atunci când semnalul specificat este recepționat. Handler-ul trebuie să aibă un singur argument de tip int, care reprezintă numărul semnalului.

Funcția signal() returnează un pointer către handler-ul anterior asociat semnalului specificat sau SIG\_ERR în caz de eroare.

Atunci când semnalul specificat este recepționat de program, handler-ul atribuit prin signal() este apelat și poate efectua anumite acțiuni în funcție de necesități. Aceste acțiuni pot include întreruperea execuției programului, ignorarea semnalului sau tratarea acestuia prin intermediul unor coduri specifice.

Există trei tipuri de acțiuni pe care le poate lua handler-ul atribuit unui semnal:

SIG\_DFL (default): Acțiunea implicită pentru semnal. De exemplu, pentru SIGINT, acțiunea implicită este să întrerupă execuția programului.

SIG\_IGN (ignore): Semnalul este ignorat și nu are nicio acțiune asociată.

Un pointer către o funcție personalizată (handler-ul): O funcție definită de utilizator care va fi apelată atunci când semnalul este recepționat.

Un aspect important al utilizării funcției signal() este faptul că handler-urile atribuite prin signal() nu garantează o anumită ordine de apel sau execuție a semnalelor. Mai mult, în unele cazuri, anumite semnale pot fi blocate sau nu pot fi atribuite handler-uri personalizate.

Funcția signal() este utilă pentru gestionarea semnalelor într-un program și permite programatorului să controleze comportamentul programului în funcție de semnalele primite de la sistemul de operare sau alte procese.

**Explicatie exemplu SIGTERM**

În acest exemplu, am definit funcția sigtermHandler() care servește ca handler personalizat pentru semnalul SIGTERM. Handler-ul este apelat atunci când semnalul SIGTERM este recepționat.

În interiorul handler-ului sigtermHandler(), se afișează un mesaj specific că semnalul SIGTERM a fost primit și programul se încheie în mod controlat prin apelul funcției exit(EXIT\_SUCCESS).

În funcția main(), se atribuie handler-ul personalizat sigtermHandler() semnalului SIGTERM utilizând funcția signal(). Apoi, se afișează un mesaj pentru a indica că programul rulează și se așteaptă semnalul SIGTERM pentru a încheia în mod controlat.

Programul rulează într-o buclă infinită, afișând mesaje la fiecare secundă pentru a simula operații. Programul continuă să ruleze până când semnalul SIGTERM este recepționat și handler-ul personalizat este apelat pentru a încheia programul în mod controlat prin apelul funcției exit()

Procesul se inchide cu **kill pid**