**UNIVERSITATEA DE STAT DIN MOLDOVA**

**FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ**

**DEPARTAMENTUL DE INFORMATICĂ**

**GRĂDINARU MĂDĂLINA**

**Cloud Computing**

**Lucrare de Laborator nr.6:**

Profesor: Aurelia Prepeliță, dr., conf.univ.

**CHIȘINĂU – 2024**

# ****1. Introducere:****

**Un proces în Linux reprezintă o instanță a unui program în execuție.** Este ca și cum am porni o aplicație pe telefon: fiecare aplicație deschisă devine un proces separat. Aceste procese sunt gestionate de sistemul de operare Linux și fiecare are propriul spațiu de memorie, resurse alocate și un identificator unic (PID).

**Caracteristicile unui proces:**

 **PID:** Fiecare proces are un număr unic de identificare (PID), care îl deosebește de celelalte.

 **Stare:** Un proces poate fi în diferite stări:

**R (Running):**  Procesul este în execuție sau gata să fie rulat.

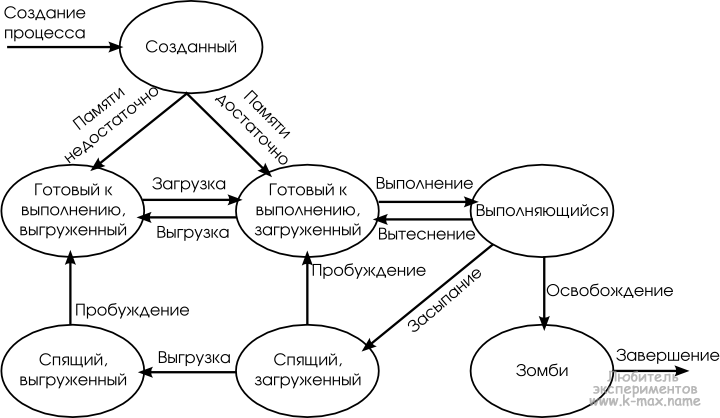
**S (Sleeping):** Procesul așteaptă un eveniment sau o resursă.

**Z (Zombie):** Procesul și-a terminat execuția, dar informațiile despre el nu au fost eliminate.

**T (Stopped):** Procesul este oprit manual sau din cauza unui semnal.

 **Ierarhie:** Procesele sunt organizate într-o structură arborescentă, cu un proces inițial (init) la rădăcină.

 **Resurse:** Fiecare proces are asociate resurse precum memorie, timp de procesor, fișiere deschise și conexiuni de rețea.



Imaginea prezintă diagrama stărilor proceselor în Linux:

1. **Созданный (Creat):** Procesul este creat, dar nu este încă pregătit să ruleze. Resursele necesare sunt verificate.
2. **Готовый к выполнению, выгруженный/загруженный (Pregătit pentru execuție, descărcat/încărcat):** Procesul așteaptă să fie planificat pentru execuție. Poate fi în memorie (încărcat) sau mutat pe disc (descărcat) dacă memoria este insuficientă.
3. **Выполняющийся (În execuție):** Procesul rulează pe procesor.
4. **Спящий, выгруженный/загруженный (În stare de așteptare, descărcat/încărcat):** Procesul așteaptă un eveniment pentru a continua execuția. Similar, poate fi în memorie sau descărcat.
5. **Зомби (Zombie):** Procesul și-a terminat execuția, dar încă există pentru a transmite informații procesului părinte.
6. **Завершение (Finalizat):** Procesul este complet șters din sistem după transmiterea informațiilor.

**Funcții utilizate pentru gestionarea proceselor:**

* **fork():**

Funcția fork() creează un nou proces, care este o copie aproape identică a procesului părinte. După apelul fork(), atât procesul părinte, cât și procesul copil continuă să execute codul imediat după apelul fork(). Aceasta returnează:

* 0 în procesul copil.
* PID-ul procesului copil în procesul părinte.
* -1 în caz de eroare, dacă procesul nu a putut fi creat.

Este utilizată pentru a crea procese copii care rulează în paralel cu procesul părinte.

* **getpid():**

Returnează PID-ul (identificatorul de proces) al procesului curent. Este folosit pentru a identifica procesul curent și pentru a-l afișa în scopuri de debug sau informații.

* **getppid():**

Returnează PID-ul procesului părinte. Este folosit pentru a determina procesul părinte al procesului curent.

* **exit():**

Funcția exit() este utilizată pentru a termina execuția unui proces și pentru a returna un cod de stare. Dacă un proces întâmpină o eroare, exit() poate fi folosit pentru a încheia execuția cu un cod de eroare (de obicei exit(1) pentru erori). De asemenea, aceasta este folosită pentru a închide procesul copil atunci când termină execuția.

* **waitpid():**

Funcția waitpid() este utilizată de procesul părinte pentru a aștepta terminarea unui proces copil specificat prin PID. Aceasta permite părintelui să colecteze informațiile despre terminarea copilului și previne apariția proceselor zombie, care rămân în tabelul de procese după ce s-au terminat, dar nu au fost curățate de părintele lor.

* **sleep():**

Funcția sleep() introduce o întârziere în execuția unui proces pentru un număr specificat de secunde. Este adesea folosită pentru a da timp unui proces copil să termine înainte ca procesul părinte să înceapă o acțiune sau pentru a preveni terminarea prematură a unui proces.

* **pipe():**

Funcția pipe() creează un canal de comunicare între procese. Acesta permite unui proces să scrie date într-un capăt al pipe-ului și altui proces să citească datele de la celălalt capăt. Este folosită pentru a transmite informații între procese, mai ales între procesul părinte și procesele copii.

Program 1:

**demonstrează crearea unui proces copil folosind fork() și ilustrează comportamentul proceselor în Linux, inclusiv modul în care un proces copil poate deveni orfan sau zombie."**

**#include <stdio.h>**

**#include <unistd.h> // pentru funcțiile getpid și fork**

**#include <stdlib.h> // pentru funcția exit**

**int main()**

**{**

**int pid;**

**// Obține ID-ul procesului curent**

**pid = getpid();**

**printf("Current Process ID is: %d\n", pid);**

**printf("[ Forking Child Process ... ]\n");**

**// Creează un proces copil și returnează PID-ul acestuia**

**pid = fork();**

**// Verifică dacă procesul copil nu a putut fi creat**

**if (pid < 0)**

**{**

**printf("Process creation failed!\n");**

**exit(-1);**

**}**

**// Dacă suntem în procesul copil**

**else if (pid == 0)**

**{**

**printf("Child Process Started...\n");**

**printf("Child Process Completed...\n");**

**}**

**// Contrar, dacă suntem în procesul părinte**

**else**

**{**

**sleep(10); // Procesul părinte așteaptă 10 secunde pentru a evita terminarea prematură**

**printf("Parent Process Running...\n");**

**printf("I am in Zombie State...\n");**

**while (1)**

**{**

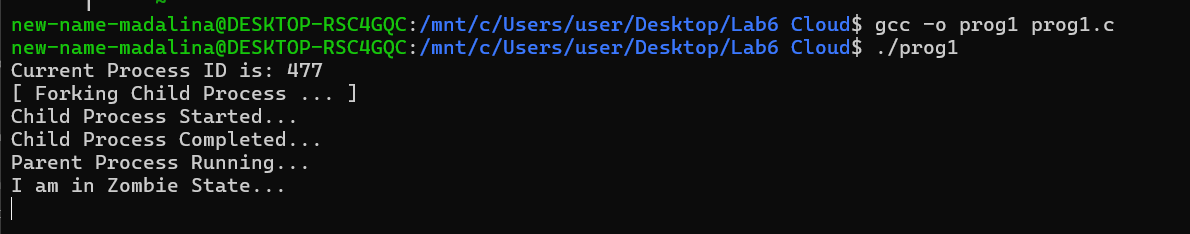
**// Buclă infinită pentru a menține procesul părinte activ**

**}**

**}**

**return 0;**

**}**

**Rezultat:  
**

 **Running**: Procesele părinte și copil rulează codul lor imediat după apelul fork().

 **Zombie**: După ce procesul copil termină execuția (afișând "Child Process Completed..."), intră în starea zombie, deoarece părintele nu colectează informațiile de terminare.

 **Infinite Loop (părinte)**: Procesul părinte intră într-o buclă infinită, menținându-se activ pentru a observa starea zombie a copilului.

**Exemplul 2:**

**demonstrează modul în care un proces copil devine orfan atunci când procesul părinte termină execuția înaintea copilului.**

**#include <stdio.h>**

**#include <unistd.h> // pentru funcțiile getpid, fork și sleep**

**#include <stdlib.h> // pentru funcția exit**

**int main()**

**{**

**int pid;**

**// Obține ID-ul procesului curent**

**pid = getpid();**

**printf("Current Process ID is : %d\n", pid);**

**// Mesaj că se începe crearea procesului copil**

**printf("[ Forking Child Process ... ] \n");**

**pid = fork();**

**if (pid < 0)**

**{**

**/\* Procesul nu a fost creat \*/**

**printf("Process creation failed!\n");**

**exit(-1);**

**}**

**else if (pid == 0)**

**{**

**/\* Procesul copil \*/**

**printf("Child Process is Sleeping ...\n");**

**// Procesul copil doarme pentru 5 secunde**

**sleep(5);**

**/\* După 5 secunde, procesul copil va verifica ID-ul părintelui său folosind getppid(). Deși procesul copil a fost creat de procesul părinte, după ce părintele moare, procesul copil va deveni orfan și va fi preluat de procesul init (PID 1). \*/**

**printf("Orphan Child's Parent ID : %d\n", getppid());**

**}**

**else**

**{**

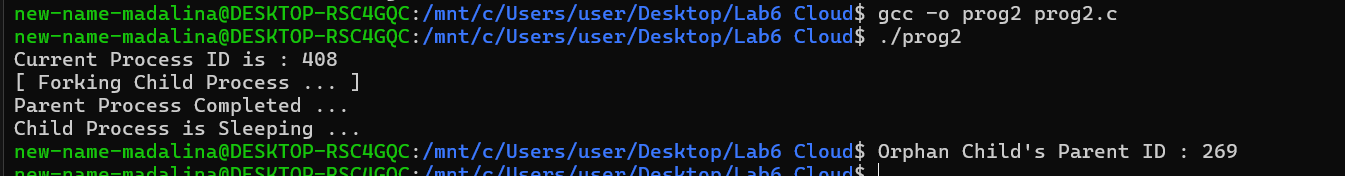
**printf("Parent Process Completed ...\n");**

**}**

**return 0;**

**}**

**Rezutat:**

****

 **Părinte:**

* **Running:** Procesul părinte începe execuția și creează copilul folosind fork().
* **Terminated:** Părintele se termină imediat după ce afișează mesajul "Parent Process Completed ...".

 **Copil:**

* **Running:** Copilul începe execuția și afișează mesajul "Child Process is Sleeping ...".
* **Sleeping:** Copilul intră în starea Sleeping pentru 5 secunde cu sleep(5).
* **Orphan:** După ce părintele se termină, copilul devine orfan. Starea sa este gestionată de procesul init.

**Program 3:**

**creează mai multe procese folosind apeluri succesive la funcția fork() și afișează PID-ul fiecărui proces creat.**

**#include <stdio.h>**

**#include <unistd.h>**

**#include <stdlib.h>**

**#include <string.h>**

**int main() {**

**char pid[255]; // Declara un șir de caractere pentru a stoca PID-ul**

**// Creează un proces copil**

**fork(); // După primul fork, vom avea 2 procese (părinte și copil)**

**fork(); // După al doilea fork, vom avea 4 procese**

**fork(); // După al treilea fork, vom avea 8 procese**

**// Utilizăm getpid() pentru a obține PID-ul procesului curent**

**sprintf(pid, "PID : %d\n", getpid());**

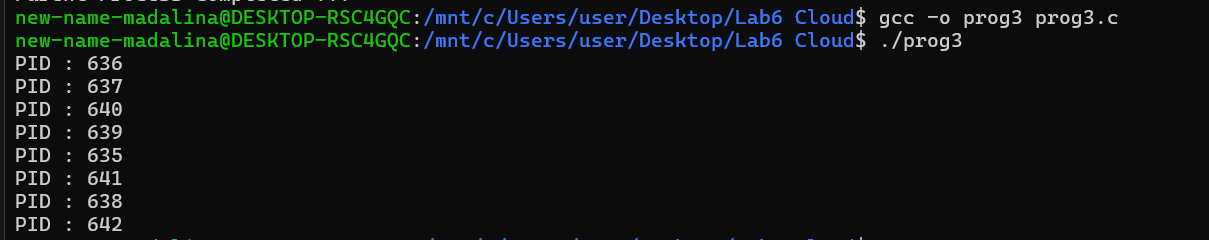
**// scriem PID-ul în standard output**

**write(STDOUT\_FILENO, pid, strlen(pid()));**

**// Termină procesul curent**

**exit(0);**

**}**

****

**Fiecare apel la fork() creează un nou proces. Numărul total de procese după n apeluri la fork() este 2n2^n2n. Acest program utilizează 3 apeluri la fork(), rezultând în 23=82^3 = 823=8 procese.**

**Program 4:**

**creează un proces copil folosind fork(), iar procesul părinte așteaptă terminarea copilului folosind waitpid().**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h**

**int main(void)**

**{**

**// Crează un proces copil**

**// Procesul părinte și procesul copil rulează simultan**

**pid\_t pid = fork();**

**if (pid == -1) {**

**// Dacă fork() returnează -1, înseamnă că a apărut o eroare în crearea procesului copil**

**perror("fork failed");**

**exit(EXIT\_FAILURE);**

**}**

**else if (pid == 0) {**

**// Dacă fork() returnează 0, suntem în procesul copil**

**printf("Hello from the child process!\n");**

**\_exit(EXIT\_SUCCESS);**

**}**

**else {**

**// Dacă fork() returnează un număr pozitiv, suntem în procesul părinte**

**// și PID-ul returnat este PID-ul procesului copil creat**

**int status;**

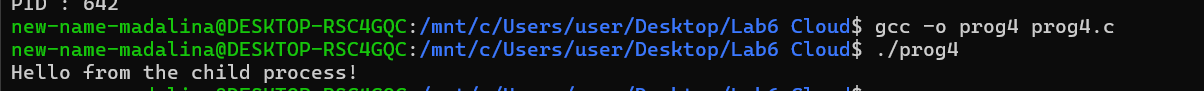
**// `status` va conține informații despre terminarea procesului copil**

**//Procesul părinte folosește waitpid() pentru a bloca execuția până când copilul își termină execuția, asigurând o gestionare corectă a resurselor și a terminării copilului.**

**(void)waitpid(pid, &status, 0); }**

**return EXIT\_SUCCESS;**

**}**

****

**Program 5:**

**creează doi copii, fiecare generând un număr aleator, trimite aceste numere procesului părinte prin pipe-uri, iar procesul părinte determină și afișează care dintre numere este mai mare.**

**#include <stdlib.h>**

**#include <stdio.h**

**#include <string.h>**

**#include <unistd.h>**

**// Funcția pentru generarea unui număr aleator între 0 și 19**

**int randomNumber(int j) {**

**// Setează seed-ul pentru generatorul de numere aleatoare**

**srand(j);**

**return rand() % 20;**

**}**

**int main() {**

**int fd[2]; // Pipe-ul pentru comunicarea părinte-copil1**

**int df[2]; // Pipe-ul pentru comunicarea părinte-copil2**

**int child1, child2; // Variabile pentru identificarea proceselor copil**

**int a, b; // Numerele generate de copii**

**int a\_rec = 0; // Numărul primit de la copil1**

**int b\_rec = 0; // Numărul primit de la copil2**

**char buffer1[255]; // Buffer pentru scrierea numărului de copil1**

**char buffer2[255]; // Buffer pentru citirea numărului de copil1**

**char buffer3[255]; // Buffer pentru scrierea numărului de copil2**

**char buffer4[255]; // Buffer pentru citirea numărului de copil2**

**// Creează primul pipe**

**if (pipe(fd) == -1) {**

**perror("Pipe failed");**

**exit(1); }**

**// Creează al doilea pipe**

**if (pipe(df) == -1) {**

**perror("Pipe failed");**

**exit(1);**

**}**

**// Creează primul proces copil**

**child1 = fork();**

**if (child1 == -1) {**

**perror("fork");**

**exit(1);**

**}**

**if (child1 == 0) {**

**// Cod executat în procesul copil1**

**close(fd[0]); // Închide capătul de citire al pipe-ului fd**

**a = randomNumber(1); // Generează un număr aleator pentru copil1**

**sprintf(buffer1, "%i\n", a); // Convertește numărul într-un șir de caractere**

**printf("Child1 sends a number: %s\n", buffer1); // Afișează nr generat**

**write(fd[1], buffer1, strlen(buffer1)); // Scrie numărul în pipe-ul fd**

**\_exit(0);**

**} else {**

**// Cod executat în procesul părin**

**close(fd[1]); // Închide capătul de scriere al pipe-ului fd**

**// Citește numărul trimis de copil1**

**read(fd[0], buffer2, sizeof(buffer2));**

**a\_rec = atoi(buffer2); // Convertește șirul în număr întreg**

**printf("Father received from child1 a number: %d\n", a\_rec); // Afișează numărul primit**

**// Creează al doilea proces copil**

**child2 = fork();**

**if (child2 == -1) {**

**perror("fork");**

**exit(1);**

**}**

**if (child2 == 0) {**

**// Cod executat în procesul copil2**

**close(df[0]); // Închide capătul de citire al pipe-ului df**

**b = randomNumber(2); // Generează un număr aleator pentru copil2**

**sprintf(buffer3, "%i\n", b); // Convertește numărul într-un șir de caractere**

**printf("Child2 sends a number: %s\n", buffer3); // Afișează numărul generat**

**write(df[1], buffer3, strlen(buffer3)); // Scrie numărul în pipe-ul df**

**\_exit(0); // Iese din procesul copil2**

**} else {**

**// Cod executat în procesul părinte după crearea copilului 2**

**close(df[1]); // Închide capătul de scriere al pipe-ului df**

**// Citește numărul trimis de copil2**

**read(df[0], buffer4, sizeof(buffer4));**

**b\_rec = atoi(buffer4); // Convertește șirul în număr întreg**

**printf("Father received from child2 a number: %d\n", b\_rec); // Afișează numărul primit**

**// Compară numerele primite de la copii și afișează care este mai mare**

**if (a\_rec > b\_rec)**

**printf("Child1's number is the biggest\n");**

**else**

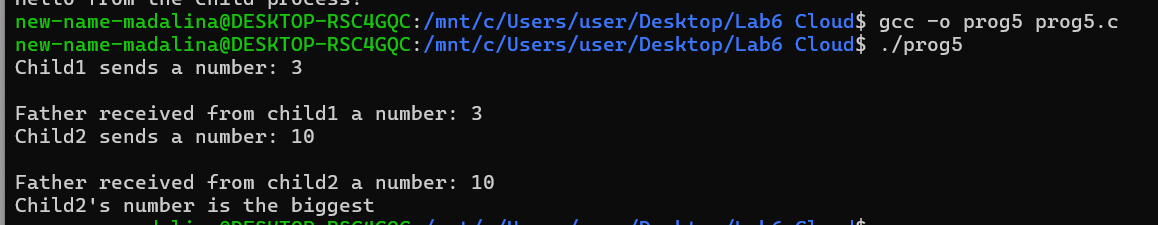
**printf("Child2's number is the biggest\n");**

**}**

**}**

**return 0;**

**}**

****

**Program6:  
creează un proces copil folosind fork() și demonstrează izolarea memoriei între procese.**

**#include <stdio.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include <unistd.h>**

**#include <stdlib.h>**

**int main() {**

**pid\_t pid; // Variabilă pentru identificarea procesului**

**int a = 0; // Variabilă partajată logic între procese, dar separată fizic**

**// Crearea unui proces copil**

**if ((pid = fork()) == -1) {**

**perror("Error at fork");**

**exit(1);**

**}**

**if (pid == 0) {**

**// Cod executat de procesul copil**

**printf("Child process id=%d, with parent: id=%d\n", getpid(), getppid());**

**printf("Child process: after fork, variable a=%d\n", a);**

**// Modifică variabila `a` doar în copil**

**a = 5;**

**printf("Child process: after modifying, variable a=%d\n", a);**

**} else {**

**// Cod executat de procesul părinte**

**printf("Parent process id=%d, with its parent: id=%d and child: id=%d\n",**

**getpid(), getppid(), pid);**

**// Așteaptă puțin pentru a permite copilului să afișeze mai întâi**

**sleep(2);**

**printf("Parent process: variable a=%d\n", a);**

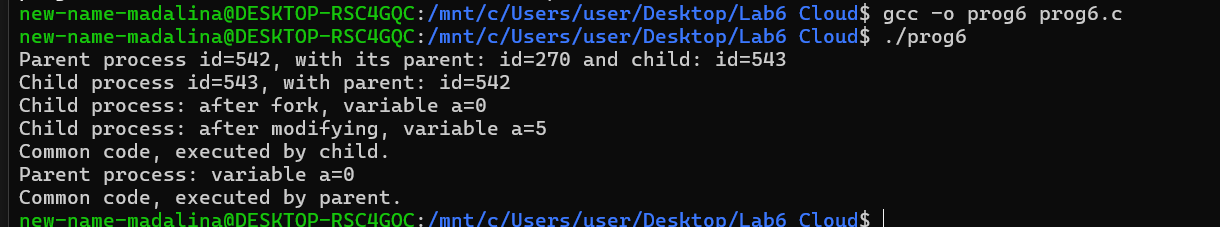
**}**

**// Cod comun executat de ambele procese**

**printf("Common code, executed by %s.\n", pid == 0 ? "child" : "parent");**

**return 0;**

**}**

****

* + Afișează PID-ul procesului copil PID-ul părintelui său , obținute cu funcțiile getpid() și getppid().
  + După apelul fork(), procesul copil vede că valoarea variabilei a este 0 (aceasta este valoarea la momentul creării copilului, deoarece fiecare proces are propria copie a variabilei)
  + Copilul modifică variabila a la 5. Această modificare nu va afecta procesul părinte, deoarece fiecare proces are o copie separată a variabilei a.
  + Procesul părinte vede că valoarea variabilei a este încă 0, chiar dacă copilul a modificat-o. Aceasta se datorează faptului că, după apelul fork(), fiecare proces are o copie separată a variabilei a. Modificările făcute de un proces nu afectează variabila în celălalt proces.