Laboratorul #11 : exerciții de laborator

Sumar:

I) Exerciții de programare cu apeluri exec (execuția programatică a unor comenzi uzuale) :

a) *Exerciții propuse spre rezolvare* :

Mai întâi, încercați să completați codul lipsă din exercițiul parțial rezolvat prezentat în suportul de laborator al acestei lecții practice..

Apoi, încercați să rezolvați singuri problema din setul de exerciții propuse spre rezolvare, pe care vi-l va indica profesorul de laborator, în timpul laboratorului:

Setul #1 Setul #2

Setul 1

1. [Exec command #4: cut]
   i) Să se scrie un program C care să execute comanda `cut -f1,3 -d: --output-delimiter=" - " /etc/group` , iar la sfârșitul execuției comenzii să afișeze textul: "`Comanda cut a fost executata ...`", dar numai în situația când într-adevăr s-a executat comanda `cut`, altfel să fie afișat textul: "`Comanda cut nu a putut fi executata...`".
   Cerință: se va folosi `execvp` și nu funcția `system`, și nici `execlp` !
   ii) Să se rescrie programul de la punctul i), utilizând primitiva `execv` în locul primitivei `execvp`.

   `//TODO` (*Sarcină pentru dvs.*): testați corectitudinea programului scris, în sensul că outputul afișat de acesta pe ecran să fie IDENTIC cu outputul afișat de comanda specificată în enunț!

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- aplicați aceeași idee prezentată în exercițiile rezolvate [Exec command #1: ls] și [Exec command #2: last].

   Atenție: comanda specificată în enunț reprezintă forma sintactică de introducere a comenzii respective la prompterul interpretorului de comenzi. Ceea ce se execută de către interpretorul de comenzi, este rezultatul interpretării acelei forme. Deci trebuie să va gândiți care este rezultatul interpretării, de către interpretorul `bash`, a liniei de comandă `cut -f1,3 -d: --output-delimiter=" - " /etc/group` pentru a putea să deduceți ce valori trebuie să specificați ca argumente în apelul `execvp`.
   Spre exemplu, cuvântul `--output-delimiter=" - "` din linia de comandă este interpretat prin `--output-delimiter= -` (caracterele ghilimele au rolul de a "proteja" cele două spații din *string*-ul ce specifică noul delimitator folosit pentru înlocuirea celui vechi de către comanda `cut`, i.e. acele ghilimele au rolul de a inhiba semnificația specială a spațiului în limbajul de comandă al interpretorului `bash`). Prin urmare, ca argument pe poziția corespunzătoare în vectorul transmis în apelul `execvp`, va trebui să specificați *string*-ul `"--output-delimiter= - "` și nu `"--output-delimiter=\" - \""` (deci nu trebuie transmise și acele ghilimele de "protejare" a celor două spații din noul delimitator).

Setul 2

1. [Exec command #8: find]
   i) Să se scrie un program C care să execute comanda `find ~ -name '*.c' -printf "%p : %k KB\n"` , iar la sfârșitul execuției comenzii să afișeze textul: "`Comanda find a fost executata ...`", dar numai în situația când într-adevăr s-a executat comanda `find`, altfel să fie afișat textul: "`Comanda find nu a putut fi executata...`".
   Cerință: se va folosi `execvp` și nu funcția `system`, și nici `execlp` !
   ii) Să se rescrie programul de la punctul i), utilizând primitiva `execv` în locul primitivei `execvp`.

   `//TODO` (*Sarcină pentru dvs.*): testați corectitudinea programului scris, în sensul că outputul afișat de acesta pe ecran să fie IDENTIC cu outputul afișat de comanda specificată în enunț!

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- aplicați aceeași idee prezentată în exercițiile rezolvate [Exec command #1: ls] și [Exec command #2: last].

   Atenție: comanda specificată în enunț reprezintă forma sintactică de introducere a comenzii respective la prompterul interpretorului de comenzi. Ceea ce se execută de către interpretorul de comenzi, este rezultatul interpretării acelei forme. Deci trebuie să vă gândiți care este rezultatul interpretării, de către interpretorul `bash`, a liniei de comandă `find ~ -name '*.c' -printf "%p : %k KB\n"` pentru a putea să deduceți ce valori trebuie să specificați ca argumente în apelul `execvp`.
   Spre exemplu, cuvântul `'*.c'` din linia de comandă este interpretat prin `*.c` (caracterele apostroafe au rolul de a "proteja" caracterul steluță * , i.e. de a inhiba semnificația specială a steluței în limbajul de comandă al interpretorului `bash`), deci ca argument, pe poziția corespunzătoare în vectorul transmis în apelul `execvp`, va trebui să specificați *string*-ul `"*.c"` și nu `"'*.c'"` (deci nu trebuie transmise și caracterele apostroafe de "protejare" a steluței în `bash`).
   Vă las ca exercițiu să vă gândiți singuri ce alte "transformări" ale parametrilor din linia de comandă trebuie să aplicați pentru a construi **corect** vectorul ce trebuie transmis în apelul `execvp`.

   Tip: dacă nu reușiți singuri să înțelegeți care este rezultatul interpretării acelei linii de comandă, de către interpretorul de comenzi `bash`, vă reamintesc despre comanda `set -x` prezentată în lecția practică "Interpretoare de comenzi Unix, partea a II-a -- scripting bash" la *slide*-ul #20, comandă ce activează opțiunea `xtrace` a interpretorului. Această opțiune este foarte utilă pentru depanare, având ca efect afișarea rezultatului interpretării liniei de comandă introduse de utilizator, înainte de a o executa. Concluzionând, activați optiunea respectivă și apoi executați comanda `find ~ -name '*.c' -printf "%p : %k KB\n"` , citind cu atenție prima linie de output afișată de aceasta (restul liniilor de output reprezintă rezultatul produs de execuția comenzii `find`).

   Observație: legat de cele spuse mai sus, vă mai reamintesc și faptul că simbolul `~` și, mai general, `~username` reprezintă căi relative la directorul *acasă* al utilizatorului curent, respectiv al utilizatorului specificat, care sunt interpretate doar de către interpretoarele de comenzi UNIX (`/bin/bash` și celelalte), deci ele NU sunt căi recunoscute la nivelul funcțiilor din API-ul POSIX (!). Practic, dacă le utilizați într-un apel de sistem dintr-un program C, vă va da eroare pe motivul "file not found".
   Prin urmare, va trebui să înlocuiți manual, în apelul `exec` utilizat, argumentul `~` cu calea reprezentată de acest simbol la nivelul interpretorului de comenzi. Cum puteți face acest lucru? 1) modalitatea ușoară, dar nu și eficientă: "hard-codați" în program calea absolută a directorului *acasă* al utilizatorului curent; 2) modalitatea recomandată: aflați programatic această cale -- cum anume? Fie consultați variabila de mediu `HOME`, primită de la părinte, fie puteți folosi funcția `getpwuid` pasându-i ca argument UID-ul propriu, pe care-l puteți afla cu funcția `getuid`.

b) *Exerciții suplimentare, propuse spre rezolvare pentru acasă* :

Iată alte câteva exerciții de programare cu apeluri exec, pe care să încercați să le rezolvați singuri în timpul liber, pentru a vă auto-evalua cunoștințele dobândite în urma acestui laborator:

1. [MyCall_System]
   Să se scrie un program C care să simuleze apelul funcției `system`, folosind doar apeluri `fork`, `wait` și `exec`.

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- trebuie să scrieți un program care, primind ca argumente în linia de comandă un nume de executabil (e.g., orice comandă validă în Linux/UNIX) și o serie de parametri pentru aceasta, să creeze un proces fiu care să se reacopere printr-un apel `exec` cu comanda respectivă, invocată împreună cu parametrii specificați. Iar procesul părinte va aștepta terminarea execuției comenzii respective și va prelucra statusul execuției comenzii respective (i.e., succes vs. eșec).
   Pentru a înțelege mai bine cum să implementați cele descrise mai sus, revedeți exercițiile rezolvate prezentate în suportul de laborator al acestei lecții practice.

2. [Run SPMD programs (versiunea în C)]
   Să se scrie un program C care să implementeze în C funcționalitatea scriptului `RunMySPMD.sh`, ce a fost descris în exercițiul rezolvat **[Run SPMD programs]** prezentat în suportul de laborator #5.

Cerință: trebuie să scrieți cod C echivalent cu acel script bash, adică NU vă este permis să apelați direct acel script, prin intermediul vreunei primitive `exec` sau a funcției `system` !

Show / Hide some suggestions for solving the problem

Ideea de rezolvare -- puteți refolosi codul din exercițiul rezolvat **[N children]** prezentat în suportul de laborator #10, pentru a crea cele N instanțe ale comenzii specificate și pentru a aștepta terminarea execuției tuturor celor N instanțe, iar în procesele fii astfel create folosiți apeluri `exec` pentru a porni executabilul/comanda specificată, împreună cu argumentul specificat pentru fiecare instanță în parte.
Așadar, NU aveți voie să **apelați direct scriptul respectiv** (prin apelul funcției `system` sau printr-un apel `exec`), ci trebuie să implementați în C funcționalitatea acelui script, folosind apeluri `fork`, `wait` și `exec`.

II) **Exerciții de programare cu mai multe procese secvențiale (a treia parte - diverse probleme de sincronizare a unor procese cooperante)** :

a) *Exerciții propuse spre rezolvare* :

Intrați pe setul de exerciții propuse spre rezolvare, pe care vi-l va indica profesorul de laborator, în timpul laboratorului, și încercați să le rezolvați singuri:

Setul #1 Setul #2

Setul 1

1. ['Supervisor-workers' pattern #3: *Coordinated distributed WordCount (wc)*]
   Să se scrie două programe C, unul numit `master_wc.c`, iar celălalt `slave_wc.c`, care să realizeze coordonarea execuției simultane și concurente a `N` instanțe ale programului din exercițiul rezolvat **[MyWc]** prezentat în suportul de laborator #6, în felul următor:
   Supervisorul va "diviza" fișierul de intrare în `N` "părți" aprox. egale (unitatea de referință nefiind octetul/caracterul, ci "linia de text"), pe care le va transmite proceselor workeri. Fiecare worker, ce va rula o instanță adaptată a programului **[MyWc]**, va calcula statisticile solicitate DOAR pe "bucata" sa de fișier primit de la supervisor (deci nu pe întreg fișierul de intrare !) și le va transmite supervisorului. La final, supervisorul va "agrega" prin însumare statisticile primite de la workeri și va afișa rezultatele finale.
   Cerință: pentru comunicațiile necesare între supervisor și workeri, folosiți mapări cu nume (sau fișiere obișnuite).
   (Indicație: Mai precis, programul master pe care trebuie să-l scrieți va crea `N` procese fii, iar în fiecare fiu se va executa printr-un apel `exec` programul slave; pe acesta îl veți adapta din programul [MyWc], pentru prelucrarea unei "bucăți" de fișier).)

   `//TODO` (*Sarcină pentru dvs.*): testați corectitudinea programului pe care-l veți scrie, comparând outputul său cu outputul afișat de comanda `wc`. Repetați testele pentru diverse fișiere de intrare.

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- programul pe care trebuie să-l scrieți cu rol de proces supervisor, va trebui să implementeze următoarele funcționalități:
   - **1 -- faza de inițializare:**
     i) Supervisorul va primi la linia de comandă numărul de workeri `N`, numele fișierului de procesat și o serie de opțiuni, dintre cele 4 posibile: `-c`, `-w`, `-l`, `-L`. Pentru procesarea opțiunilor specificate în linia de comandă refolosiți codul corespunzător din programul **[MyWc]**.
     ii) Apoi va crea `N` mapări cu nume, de dimensiuni adecvate și va scrie opțiunile respective la începutul fiecăreia dintre cele `N` mapări cu nume, sau, alternativ, le poate transmite workerilor prin argumentele apelului `exec` (descris mai jos).
     iii) Apoi va citi linie cu linie fișierul de intrare și va "distribui" liniile citite, în manieră circulară, către workeri, scriindu-le în cele `N` mapări cu nume (create așadar înaintea workerilor și "asignate" câte una pentru fiecare worker).
   - **2 -- faza de calcul distribuit coordonat:**
     iv) Abia apoi procesul supervisor va crea `N` procese fii (a se revedea în acest sens exercițiul rezolvat **[N children]** prezentat în suportul de laborator #10), iar în fiecare proces fiu se va apela printr-un apel `exec` o instanță a programului worker, părintele așteptând apoi terminarea tuturor fiilor.
     v) Programul pe care trebuie să-l scrieți cu rol de proces worker, va citi opțiunile comenzii `wc` (fie din maparea cu nume "asignată" lui, fie din parametrii primiți în lnia de comandă prin apelul `exec`), precum și celelalte date de intrare necesare (i.e., poziția/lungimea, în cadrul mapării cu nume "asignate" lui, a "bucății" lui de fișier -- adică "colecția" liniilor de text din fișierul original ce i-au fost distribuite lui pentru procesare). Aceste informații sunt deja în memorie (sincronizarea necesară este realizată prin faptul că procesele fii sunt startate abia după scrierea în memorie a acestor date, de către supervisor).
     vi) Apoi workerul va calcula statisticile corespunzătoare opțiunilor respective. Pentru aceasta, puteți refolosi codul corespunzător din programul **[MyWc]**, pe care însă trebuie să-l adaptați pentru procesarea "fișierului" aflat în memorie, în maparea cu nume (*Notă*: dacă ați rezolvat cumva exercițiul suplimentar **[MyWc_mmap]** propus spre rezolvare în laboratorul #9, atunci efortul de adaptare al codului pe care trebuie să-l faceți va fi considerabil diminuat).
     vii) La final, fiecare proces worker va transmite statisticile calculate către părinte, scriindu-le în maparea cu nume "asignată" lui.
   - **3 -- faza de finalizare, i.e. de detecție a terminării calculului distribuit:**
     viii) Procesul supervisor va folosi apeluri `wait` pentru a aștepta terminarea tuturor proceselor workers, care vor indica prin codul de terminare dacă au finalizat calculul cu succes, sau nu. Dacă nu este vreo eroare la niciun worker, atunci se poate face "agregarea" rezultatelor parțiale pentru a obține rezultatele totale:

Pe măsură ce se termină fiecare worker, supervisorul va citi statisticile parțiale, raportate/scrise de acel worker în maparea cu nume asociată acestuia, și le va "agrega" în statisticile totale. "Agregarea" înseamnă operația de adunare pentru statisticile corespunzătoare opțiunilor `-c`, `-w` și `-l`, iar pentru opțiunea `-L` se va folosi operația de maxim pentru "agregare". Iar în cazul în care acel worker raportează că s-a terminat cu eroare, atunci supervizorul poate abandona așteptarea celorlalți workeri încă activi și "agregarea"rezultatelor de la aceștia, trecând la pasul final 4).

- **4 -- faza de afișare a rezultatelor:**
  Supervisorul va afișa pe ecran statisticile totale, dacă nu au fost erori la workeri, sau, în caz contrar, ne va informa asupra erorilor survenite.

Setul 2

1. ['Supervisor-workers' pattern #4: *Coordinated distributed alphanumeric characters count in a binary file*]
   Să se scrie două programe C, unul numit `master_wc-bin.c`, iar celălalt `slave_wc-bin.c`, care să realizeze următoarele: primul program, cu rolul de supervizor, va realiza coordonarea execuției simultane și concurente a `N` instanțe ale programului worker, iar acesta din urmă va avea rolul de a număra **caracterele alfanumerice** (i.e., doar cifrele și literele, majuscule și minuscule) aflate într-un anumit "segment" dintr-un fișier specificat.
   Supervisorul va primi de la linia de comandă un fișier binar de procesat și un număr ce va reprezenta numărul de instanțe ale programului worker ce vor fi rulate. Supervisorul va "diviza" fișierul de intrare în `N` "părți" aprox. egale (unitatea de referință fiind octetul/caracterul; astfel fișierul poate fi împărțit în funcție de dimensiunea fișierului respectiv).
   Fiecare "parte" astfel obținută prin divizare este practic o porțiune din acel fișier, caracterizată printr-un "punct de start" (i.e., *offset*-ul de început al porțiunii) și o dimensiune a ei, pe care supervisorul le va transmite către procesul worker ce va procesa acea porțiune, ca argumente în apelul `exec` folosit de fiul supervisorului ce se va reacoperi cu workerul respectiv.
   Așadar, fiecare worker va primi 3 argumente în linia de comandă (transmise printr-un apel `exec`). Primul argument va fi o cale către un fișier de procesat (i.e., aceiași cale primită de master și transmisă workerului), iar următoarele două argumente vor reprezenta *offset*-ul de început și dimensiunea porțiunii din acel fișier. Acel proces worker va procesa porțiunea de fișier primită de la supervisor, calculând numărul total de caractere alfanumerice din porțiunea respectivă de fișier (deci nu pe întreg fișierul de intrare !) și va transmite rezultatul supervisorului (prin scrierea acestuia fie într-o mapare cu nume, fie într-un fișier obișnuit).
   La final, supervisorul va "agrega" prin însumare rezultatele primite de la workeri și va afișa rezultatul final.
   Cerință: pentru comunicațiile necesare de la workeri spre supervisor, folosiți mapări cu nume (sau fișiere obișnuite).

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- este asemănătoare cu cea descrisă în sugestiile de rezolvare date la problema de la setul #1.

b) *Exerciții suplimentare, propuse spre rezolvare pentru acasă* :

Alte câteva exerciții de programare cu apelurile de sistem `fork`, `wait` și `exec`, a unor probleme de sincronizare diversificate:

1. [MyShell v1] Show / hide this problem

   Să se scrie un program C care să ofere funcționalitatea minimă, de bază, a unui *shell* pentru sistemele de operare UNIX/Linux: într-o buclă va afișa un prompter și va prelua de la tastatură numele unei comenzi simple, cu sau fără parametri, pe care o va executa, fie în *foreground* (caz în care va aștepta terminarea execuției ei înainte de a reafișa prompterul), fie în *background* (caz în care nu mai așteaptă terminarea ei), apoi va relua bucla de citire și execuție de comenzi simple, până la citirea comenzilor `exit` sau `logout`.
   Cerințe: se va folosi primitiva `execlp` și nu funcția `system` ! La așteptarea introducerii unei comenzi, programul va afișa ca prompter textul: `MyShell>` .

2. ['Supervisor-workers' pattern #5: *Coordinated parallel sorting*] Hide / show this problem

   Să se scrie un program C care să realizeze coordonarea execuției simultane și concurente a `N` instanțe ale programului de sortare cu lacăte din exercițiul rezolvat **[MyCritSec #2 : Parallel sorting]**) prezentat în suportul de laborator #7.
   (Indicație: Mai precis, programul pe care trebuie să-l scrieți va crea `N` procese fii, iar în fiecare fiu va executa programul de sortare cu lacăte.)

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- programul pe care trebuie să-l scrieți va trebui să implementeze următoarele funcționalități (cu rol de proces supervisor):

   - **1 -- faza de inițializare:** i) numărul de instanțe `N` se va prelua ca parametru în linia de comandă, sau se va citi de la tastatură, în caz contrar; ii) pregătirea secvenței de numere ce urmează a fi sortată, fie prin citirea ei de la tastatură, fie prin generarea de numere aleatoare, și scrierea ei, în format binar, într-un fișier numit `secventa.bin`, apelând în acest scop programul demonstrativ `txt2bin_write-file.c` prezentat în suportul de laborator #7 ; pentru simplificare puteți recurge la apelarea cu `system(comandă);` .

- **2 -- faza de sortare concurentă coordonată:** se vor porni cele `N` instanțe ale programului worker (i.e. programul `sortare_cu-lacate.c` din exercițiul rezolvat **[MyCritSec #2 : Parallel sorting]** prezentat în suportul de laborator #7), astfel: procesul supervisor va crea `N` procese fii (a se revedea în acest sens exercițiul rezolvat **[N children]** prezentat în suportul de laborator #10), iar în fiecare proces fiu se va apela printr-un apel `exec` o instanță a programului worker, numele fișierului de sortat (i.e., `secventa.bin`) fiindu-i transferat ca argument în linia de comandă.
- **3 -- faza de finalizare, i.e. de detecție a terminării calculului distribuit:** se vor folosi apeluri `wait` în procesul supervisor pentru a aștepta terminarea tuturor proceselor workers, care vor indica prin codul de terminare dacă au finalizat sortarea cu succes, sau nu (recitiți programul `sortare_cu-lacate.c` pentru a vedea care este codul de terminare folosit pentru a indica finalizarea cu succes a sortării).
  În cazul acestei probleme, în clipa în care procesul supervisor detectează "terminarea calculului distribuit" prin primirea **primului** cod de terminare ce indică finalizarea cu succes a sortării de către un worker oarecare, aceasta semnifică faptul că fișierul a devenit sortat, și prin urmare supervisorul poate opri forțat toate celelalte procese worker care mai sunt încă active, pentru a nu mai continua degeaba, căci oricum nu vor mai găsi nicio inversiune de efectuat (în acest scop, puteți folosi apelul de sistem `kill(pid_worker_activ,SIGTERM);`).
- **4 -- faza de afișare a rezultatelor:** se va afișa pe ecran conținutul sortat din fișierul `secventa.bin`, apelând în acest scop programul demonstrativ `bin2txt_read-file.c` prezentat în suportul de laborator #7 ; pentru simplificare puteți recurge la apelarea cu `system(comandă);` .

3. ['Supervisor-workers' pattern #6: *Coordinated distributed "end-of-year" report*] Show / hide this problem

4. ['Client/Server' pattern #1: *A supervisor which spawns a new worker for every task received*] Show / hide this problem

5. ['Client/Server' pattern #1bis: *A supervisor which spawns a new worker for every task received*] Show / hide this problem

Referitor la programul pe care l-ați scris pentru a rezolva exercițiul **['Client/Server' pattern #1:** ***A supervisor which spawns a new worker for every task received*****]** propus mai sus și care reprezenta un exemplu de implementare a șablonului 'Client/Server de tip secvențial' (i.e., serverul poate servi un singur client la un moment dat).
Se cere să se modifice programul respectiv, astfel încât să se implementeze șablonul 'Client/Server de tip paralel/concurent' (i.e., serverul poate servi mai mulți clienți în același timp).
(Indicație: renunțați la așteptarea terminării workerului curent și, în schimb, folosiți o buclă de așteptare ocupată, în care să verificați încontinuu apariția oricăruia dintre cele două tipuri de evenimente care ne interesează: o nouă cerere de calcul de la tastatură, respectiv terminarea unuia dintre workerii activi. Pentru a "aștepta ocupat" (i.e., prin testare repetitivă), în bucla de care pomeneam, apariția acestor două tipuri de evenimente, folosiți apeluri asincrone/neblocante, în loc de apelurile uzuale, `read`/`scanf` și respectiv `wait`, care sunt blocante.)

Show / Hide some suggestions for solving the problem

6. ['Client/Server' pattern #2: *A supervisor which spawns a new worker for every task received*] Show / hide this problem

7. ['Client/Server' pattern #2bis: *A supervisor which spawns a new worker for every task received*] Show / hide this problem

Referitor la programul pe care l-ați scris pentru a rezolva exercițiul **['Client/Server' pattern #2:** ***A supervisor which spawns a new worker for every task received*****]** propus mai sus și care reprezenta un exemplu de implementare a șablonului 'Client/Server de tip secvențial' (i.e., serverul poate servi un singur client la un moment dat).
Se cere să se modifice programul respectiv, astfel încât să se implementeze șablonul 'Client/Server de tip paralel/concurent' (i.e., serverul poate servi mai mulți clienți în același timp).

Show / Hide some suggestions for solving the problem

Ideea de rezolvare -- este similară cu cea descrisă în sugestiile de rezolvare date la problema **['Client/Server' pattern #1bis: ...]**.

III) **Exerciții ce presupun corectarea unor greșeli 'strecurate' într-un program C dat** :

a) *Exerciții propuse spre rezolvare pentru acasă* :

1. [Exemplul #2 cu erori sintactice și semantice]
   Se dă programul C de mai jos, ce conține (măcar) trei erori sintactice și o greșeală logică:

   Show / Hide the program

```c
/*
  Filename: p2.c

  Programul de mai jos ar trebui să ofere următoarea funcționalitate:

    Procesul principal creează un fiu.
    Procesul fiu se reacoperă cu comanda stat, executată cu opțiunea de a afisa tipul fișierului pentru fișierul "p2.c"
    și cu redirectarea output-ului către fișierul "fisier.txt".
    În acest timp, procesul părinte așteaptă să se termine fiul, apoi citește din "fisier.txt" primele 10 caractere și le afișează.

*/

#include <unistd.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
        int pid1;

        pid1 = fork();

        if(pid1 == -1)
        {
                perror(failed to create child process);
                return -1;
        }

        if(pid1==0)
        {
                int fd = open("fisier.txt", O_WRONLY|O_CREAT, 0600);
                dup2(fd, 1);
                close(fd);
                execlp("stat","stat","p2.c","--printf","%F" ,NULL);
                return 0;
        }
        else
        {
                wait(NULL);
                int fd = open("fisier.txt", O_RDONLY);
                char a;
                read(fd, &a, siezof(char));
                close(fd);
                print("Primele 10 caractere din fisier.txt sunt: %c\n", a);
        }

        return 0;
}
```

i) Explicați ce se dorește a se afișa pe ecran în urma execuției programului.
ii) Corectați eventualele erori existente astfel încât programul să poată fi executat și să ofere la execuție EXACT funcționalitatea descrisă în comentariul de la începutul programului.

Show / Hide some suggestions for solving the problem

i) Compilați programul, identificați eventualele mesaje de eroare (sau avertismente) ce apar pe ecran și încercați să identificați cauzele care le produc, corectând în mod adecvat programul! Repetați acești pași până când nu se mai afișează nicio eroare (sau avertisment) la compilare.
Tip: prin această manieră ar trebui să găsiți și să corectați (măcar) trei erori sintactice.

ii) Apoi încercați să identificați și eventualele erori semantice (i.e., *bug*-uri în cod), adică greșeli logice strecurate în program care nu cauzează mesaje de eroare la compilarea lui, dar datorită cărora programul nu face exact ceea ce este specificat în enunț că ar trebui să facă!

2. [Exemplul #3 cu erori sintactice și semantice]
Se dă programul C de mai jos, ce conține (măcar) trei erori sintactice și o greșeală logică:

Show / Hide the program

```
/*
  Filename: p3.c

  Programul de mai jos ar trebui să ofere următoarea funcționalitate:

    Procesul principal creează un fiu.
    Procesul fiu se reacoperă cu comanda ps, executată cu opțiunea -o pid,user,args.
    În acest timp, procesul părinte așteaptă să se termine fiul, apoi afișează un mesaj.

*/

#include <unistd.h>
#include <stdio.h>

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

        int pid3;

        pid3 = fork();

        if(pid3 == -1)
        {
                perror("failed to create  child process");
                return -1;
        }

        if(pid != 0)
        {
                wait(NULL);
                print("Procesul fiu a executat comanda ps.\n");;
        }
        else
        {
                char* parametru[] = {"ps","-o","pid","user","args",NULL};
                execvp("ps", parametru);
                abort();
        }

        return perror;
}
```

i) Explicați ce se dorește a se afișa pe ecran în urma execuției programului.
ii) Corectați eventualele erori existente astfel încât programul să poată fi executat și să ofere la execuție EXACT funcționalitatea descrisă în comentariul de la începutul programului.

Show / Hide some suggestions for solving the problem

i) Compilați programul, identificați eventualele mesaje de eroare (sau avertismente) ce apar pe ecran și încercați să identificați cauzele care le produc, corectând în mod adecvat programul! Repetați acești pași până când nu se mai afișează nicio eroare (sau avertisment) la compilare.
Tip: prin această manieră ar trebui să găsiți și să corectați (măcar) trei erori sintactice.

ii) Apoi încercați să identificați și eventualele erori semantice (i.e., *bug*-uri în cod), adică greșeli logice strecurate în program care nu cauzează mesaje de eroare la compilarea lui, dar datorită cărora programul nu face exact ceea ce este specificat în enunț că ar trebui să facă!

3. [Exemplul #4 cu erori sintactice și semantice]
   Se dă programul C de mai jos, ce conține (măcar) trei erori sintactice și o greșeală logică:

   Show / Hide the program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>

int main()
{
        char w=0, *text;
        int p, q[2];
        pipe(q[2]);
        p=fork(2);
        if(p==-1) exit(2);
        if(!p)
        {
                dup2(q[0],0);
                close(q[1]);
                while( read(0,&w,1) != 0)
                        printf("%c",w);
                wait(NULL);
        }
        elif
        {
                text="Salutari!\n";
                write(q[0],text,strlen(text));
        }
        return 0;
}
```

   i) Explicați ce se dorește a se afișa pe ecran în urma execuției programului.
   ii) Corectați eventualele erori existente astfel încât programul să poată fi executat și să producă la execuție afișarea pe ecran de către procesul fiu a textului primit de la părinte.
   (Recomandare: abordați acest exercițiu după ce învățați lecția despre canale de comunicație anonime, în laboratorul următor.)

   Show / Hide some suggestions for solving the problem

   i) Compilați programul, identificați eventualele mesaje de eroare (sau avertismente) ce apar pe ecran și încercați să identificați cauzele care le produc, corectând în mod adecvat programul! Repetați acești pași până când nu se mai afișează nicio eroare (sau avertisment) la compilare.
   Tip: prin această manieră ar trebui să găsiți și să corectați (măcar) trei erori sintactice.

   ii) Apoi încercați să identificați și eventualele erori semantice (i.e., *bug*-uri în cod), adică greșeli logice strecurate în program care nu cauzează mesaje de eroare la compilarea lui, dar datorită cărora programul nu face exact ceea ce este specificat în enunț că ar trebui să facă!