# Laborator #12 : exerciții de laborator

**Sumar:**



---

## I) Exerciții de programare ce implementează comunicații între procese (prima parte, folosind canale de comunicație anonime):

*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. [Pipe commands #3]
   Generalizarea exercițiilor rezolvate [Pipe commands #1 & #2] din suportul online de laborator:
   Să se scrie un program C care "simulează" comanda înlănțuită:
   `UNIX> grep /bin/bash /etc/passwd | cut -d: -f1,3-5 | sort -t: -k 2,2 -n | cut -d, -f1`
   ***Cerință***: pentru executarea comenzilor simple din lanț se vor utiliza apeluri de sistem din familia `exec` și nu funcția `system`.
   (Indicație: rezolvare similară ca la exercițiile amintite mai sus, doar că va fi nevoie de 4 procese, câte unul pentru fiecare comandă simplă din comanda înlănțuită din enunț, și respectiv de 3 canale anonime pentru realizarea înlănțuirilor.)

2. ['Supervisor & cooperating workers' pattern #1 (v1, using anonymous pipes for IPC)]
   Să se scrie un program C ce creează două procese fii și care va avea comportamentul descris în continuare:
   - procesul părinte citește dintr-un fișier cu numele *date.txt* un șir de caractere, până la sfârșitul fișierului, și le trimite printr-un canal anonim primului fiu;
   - primul proces fiu primește caracterele de la procesul părinte, le selectează doar pe acelea care sunt litere mici și le trimite printr-un alt canal anonim către cel de-al doilea fiu;
   - al doilea proces fiu creează un fișier numit *statistica.txt* în care va scrie, pe câte o linie, fiecare literă distinctă și numărul de apariții ale acesteia în fluxul de date primit. În plus, la final va trimite părintelui, printr-un canal anonim suplimentar, numărul de litere distincte întâlnite în fluxul primit;
   - procesul părinte afișează pe ecran rezultatul primit de la al doilea proces fiu.
   *Cerință*: pentru comunicarea între procese se vor folosi canale anonime.

   Show / Hide some suggestions for solving the problem

```
Ideea de rezolvare -- următoarea schiță descrie pașii ce trebuie implementați:

- Procesul inițial P0 va crea mai întâi trei canale anonime, canal_P0toP1, canal_P1toP2 și canal_P2toP0.

- Apoi P0 va crea cele două procese fii, P1 și P2 (astfel, acestea vor 'moșteni' descriptorii pentru cele trei canale);
  Notă: evident, al doilea apel fork() pentru crearea fiului P2 se va pune pe ramificația tatălui de la primul apel fork().

- După crearea fiilor (moment începând de la care toate cele trei procese se vor executa în paralel !!!),
  procesul P0 va continua astfel:
  1. citește conținutul fișierului date.txt și-l scrie în canal_P0toP1;
  2. apoi (practic, la final), P0 va aștepta, prin apelul read(), să citească numărul transmis pe canal_P2toP0 și-l va afișa.

- După creare, fiul P1 va executa următoarea buclă:
  1. citește câte 1 octet din canal_P0toP1 și ...;
  2. ..., numai dacă este literă mică, scrie acel octet în canal_P1toP2.
  Notă: terminarea buclei se va face când P1 "citește" EOF din canal_P0toP1.

- După creare, fiul P2 va executa următoarea buclă:
  1. citește 1 octet (o literă mică) din canal_P1toP2;
  2. actualizează vectorul de apariții și totalul specificat în enunț, conform cu litera citită;
  Notă: terminarea buclei se va face când P2 "citește" EOF din canal_P1toP2.
  3. la finalul buclei, scrie totalul (un număr întreg) în canal_P2toP0.

  Notă: terminarea buclelor din P1 și P2 implică faptul ca P1, respectiv P2, să "citească" EOF din canalul corespunzător,
  i.e. faptul că nu mai există "scriitori" pentru canalul respectiv.
  Așadar, asigurați-vă de acest lucru!
  Cea mai simplă modalitate de a realiza acest lucru: fiecare din cele 3 procese să închida fiecare din cele 6 capete ale celor 3 canale,
  cât mai devreme posibil (i.e., din momentul când nu va mai avea nevoie în viitor de acel capăt).
```

Setul 2

1. [Pipe commands #4]
   Generalizarea exercițiilor rezolvate [Pipe commands #1 & #2] din suportul online de laborator:
   Să se scrie un program C care "simulează" comanda înlănțuită:
   `UNIX> w -h | tr -s " " | cut -d" " -f1,8 | sort -t " " -k 1`
   ***Cerință***: pentru executarea comenzilor simple din lanț se vor utiliza apeluri de sistem din familia `exec` și nu funcția `system`.
   (Indicație: rezolvare similară ca la exercițiile amintite mai sus, doar că va fi nevoie de 4 procese, câte unul pentru fiecare comandă simplă din comanda înlănțuită din enunț, și respectiv de 3 canale anonime pentru realizarea înlănțuirilor.)

2. ['Supervisor & cooperating workers' pattern #2 (using anonymous pipes for IPC)]
   Să se scrie un program C ce creează două procese fii și care va implementa următoarea funcționalitate:
   - procesul părinte citește de la tastatură o propoziție, pe care o va transmite primului fiu;
   - primul fiu va elimina toate vocalele din textul primit și va transmite textul astfel obținut către fiul al doilea;
   - al doilea fiu va procesa textul primit transformând toate literele mici în litere mari, iar rezultatul îl va trimite către tată;
   - la final, tatăl va afișa pe ecran textul prelucrat primit de la al doilea fiu.
   *Cerință*: pentru comunicarea între procese se vor folosi canale anonime.

   Show / Hide some suggestions for solving the problem

   ***Ideea de rezolvare*:** este asemnănătoare cu cea prezentată la problema a doua din setul #1. Consultați indicațiile date acolo și adaptați-le pentru această problemă.

Iată și o sugestie generală de rezolvare a primei probleme propuse în seturile de mai sus: Show / Hide some suggestions for solving these problems

***Ideea generală de rezolvare*** se poate desprinde ușor recitind, cu atenție, fie observația referitoare la a treia soluție prezentată în exercițiul rezolvat [Pipe commands #2] din suportul online de laborator, fie observația referitoare la a doua soluție prezentată în același exercițiu.

*Exerciții suplimentare, propuse spre rezolvare pentru acasă:*

Iată alte câteva exerciții de programare ce utilizează comunicații prin canale anonime, pe care să încercați să le rezolvați singuri în timpul liber, pentru a vă auto-evalua cunoștințele dobândite în urma acestui laborator:

1. [MyShell v2, with pipe commands]
   Incorporați în programul realizat la exercițiul [MyShell v1] din laboratorul precedent, ideea generalizată de la exercițiile **[Pipe commands #1, #2 și #3]** din laboratorul curent:

   Să se scrie un program C care, într-o buclă, preia de la tastatură numele a `m` comenzi simple înlănțuite prin simbolul `pipe`, i.e. o linie de forma următoare:
   `UNIX>` $comanda_1\ arg_{1,1}\ ...\ arg_{1,N1}$ `|` $comanda_2\ arg_{2,1}\ ...\ arg_{2,N2}$ `|` `...` `...` `|` $comanda_m\ arg_{m,1}\ ...\ arg_{m,Nm}$
   și le execută în manieră înlănțuită, similar interpretoarelor clasice de comenzi din UNIX/Linux.
   La așteptarea introducerii unei noi comenzi înlănțuite, programul va afișa prompterul "`UNIX>`". Pentru executarea comenzilor se vor utiliza apeluri de sistem din familia `exec` și nu funcția `system`.

   Show / Hide some suggestions for solving this problem

   ***Ideea de rezolvare*** se poate desprinde ușor recitind observația referitoare la a doua soluție prezentată în exercițiul rezolvat [Pipe commands #2] din suportul online de laborator.

2. ['Supervisor-workers' pattern #6: *A coordinated distributed sum #1N* (v2, using anonymous pipes for IPC)]
   Să se modifice programul din exercițiul rezolvat ['Supervisor-workers' pattern #1N: *A coordinated distributed sum #1N* (v1, using regular files for IPC)] prezentat în laboratorul precedent, astfel încât comunicațiile între procese să se realizeze prin canale anonime, în loc de fișiere obișnuite.
   În plus, citirea numerelor de la tastatură se va muta în cod după apelurile fork(), lucru posibil deoarece sincronizările ce erau necesare în varianta cu fișiere obișnuite, se vor realiza acum în mod automat (prin faptul că citirea din canale este, în mod implicit, blocantă).
   (Indicație: este suficient să utilizați doar N+1 canale anonime, în locul celor N+N fișiere de intrare și de ieșire.)

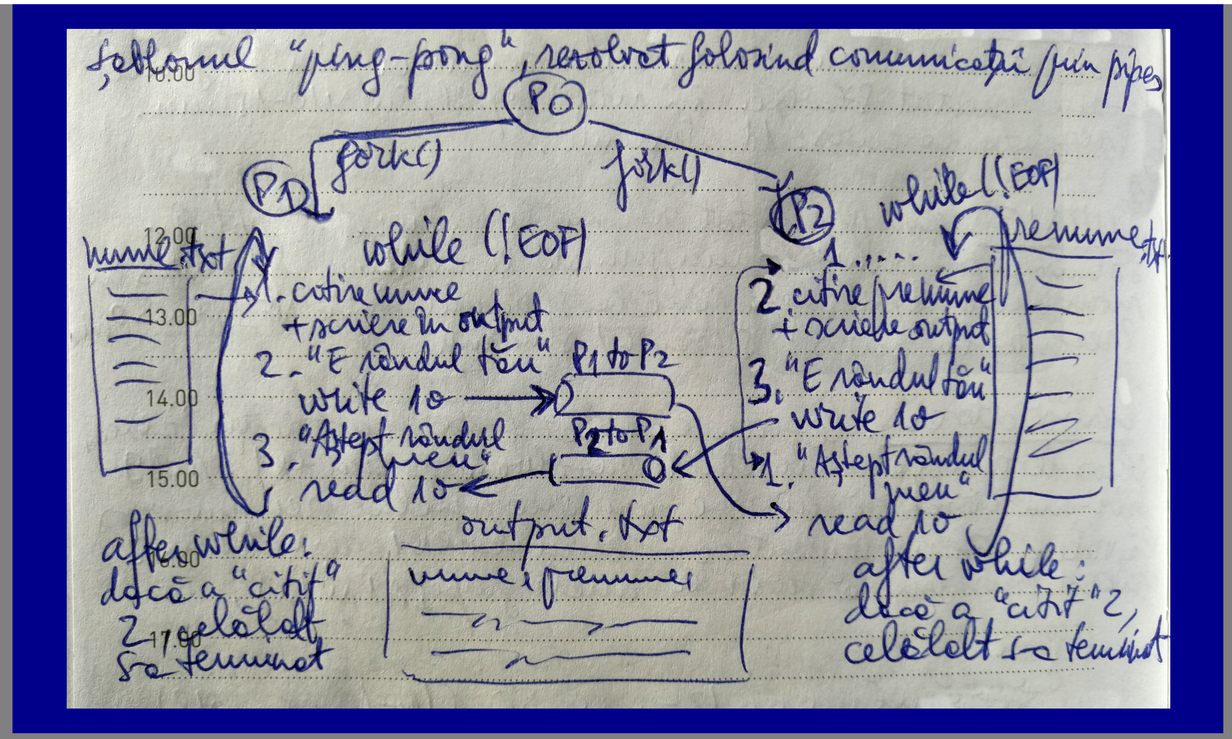
   Show / Hide some suggestions for solving this problem

   Încercați să rezolvați singuri problema, iar dacă nu reușiți, puteți consulta programul demonstrativ suma_pipes.c, cu observația că în acel program se folosesc doar două procese worker, deci trebuie să-l adaptați pentru cazul cu un număr variabil de procese worker, corespunzător acestei probleme.

3. ['Ping-pong' pattern #4 (v1, using anonymous pipes for IPC)]
   Se presupune că în două fișiere, `nume.txt` și `prenume.txt`, sunt scrise, pe câte o linie, numele și respectiv prenumele unor persoane (se va trata și cazul de excepție când nu există o corespondență bijectivă la nivel de linie între cele două fișiere).
   Să se scrie un program C care va crea două procese fii: primul proces fiu va citi, într-o buclă, câte o linie din fișierul `nume.txt`, iar al doilea proces fiu va citi, într-o buclă, câte o linie din fișierul `prenume.txt`. Ambele procese fii vor scrie șirul de caractere citit, la fiecare iterație, în fișierul de ieșire `persoane.txt`. Cele două procese trebuie să-și sincronizeze execuția (utilizând canale anonime de comunicație), astfel încât în fișierul de ieșire să apară pe fiecare linie numele și prenumele corespunzătoare unei aceleiași persoane.
   (Indicație: scopul acestui exercițiu este acela de a implementa corect un mecanism de sincronizare de forma "Acum e rândul meu --> acum e rândul tău --> acum e rândul meu --> acum e rândul tău --> ... ș.a.m.d.", folosind comunicații prin intermediul canalelor anonime de comunicație, în locul unui fișier obișnuit.
   *Notă*: după cum spuneam în preambulul suportului de laborator #10, șablonul 'ping-pong' este de fapt o instanță cu `p=2` procese, pentru un șablon mai general de sincronizare între `p` procese, cunoscut în literatura de specialitate sub denumirea de șablonul de sincronizare *token ring*.)

   Show / Hide some suggestions for solving the problem

   ***Ideea de rezolvare*** se poate desprinde din următoarea diagramă (schemă logică), scrisă de mână:




Explicitând diagrama în cuvinte, iată care sunt operațiile pe care trebuie să le implementați:

- Procesul inițial P0 va crea mai întâi două canale anonime, `canal_P1toP2` și `canal_P2toP1`, și va scrie 1 octet (oarecare) în `canal_P2toP1`.
- Apoi P0 va crea cele două procese fii, P1 și P2 (astfel, acestea vor 'moșteni' descriptorii pentru cele două canale).
- După crearea fiilor, P0 așteaptă puțin (e.g., cu un apel sleep(1) ) și apoi închide toate capetele către cele două canale.
- Apoi, P0 poate aștepta terminarea celor doi fii, din motive estetice (pentru a se afișa prompterul abia după terminarea lor).
- După creare, fiul P1 va executa următoarea buclă:
  1. citește 1 octet (oarecare) din `canal_P2toP1` (aici va aștepta, prin apelul read(), până va primi 1 octet prin canal);
  2. citește o nouă linie din fișierul `nume.txt` și o afișează pe ecran (respectiv, o adaugă la fișierul `persoane.txt`);
  3. scrie 1 octet (oarecare) în `canal_P1toP2`.

  *Notă*: terminarea buclei se va face când P1 ajunge la EOF în fișierul `nume.txt`, sau când octetul citit din `canal_P2toP1` are o valoare specială ce indică faptul că P2 a terminat de parcurs fișierul lui.
- După creare, fiul P2 va executa următoarea buclă:
  1. citește 1 octet (oarecare) din `canal_P1toP2` (aici va aștepta, prin apelul read(), până va primi 1 octet prin canal);
  2. citește o nouă linie din fișierul `prenume.txt` și o afișează pe ecran (respectiv, o adaugă la fișierul `persoane.txt`);
  3. scrie 1 octet (oarecare) în `canal_P2toP1`.

  *Notă*: terminarea buclei se va face când P2 ajunge la EOF în fișierul `prenume.txt`, sau când octetul citit din `canal_P1toP2` are o valoare specială ce indică faptul că P1 a terminat de parcurs fișierul lui.

*Atenție*: terminarea buclelor din P1 și P2 implică faptul ca procesul P1, respectiv P2, să "citească" EOF din canalul corespunzător (i.e., faptul ca să nu mai existe "scriitori" pentru canalul respectiv).
Așadar, asigurați-vă că se întâmplă acest lucru! Ok, dar cum? *Răspuns*: cea mai simplă modalitate de a realiza acest lucru este ca fiecare dintre cele 3 procese să închida fiecare dintre cele 6 capete ale celor 3 canale, cât mai devreme posibil (i.e., din momentul când nu va mai avea nevoie în viitor de acel capăt).

---

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

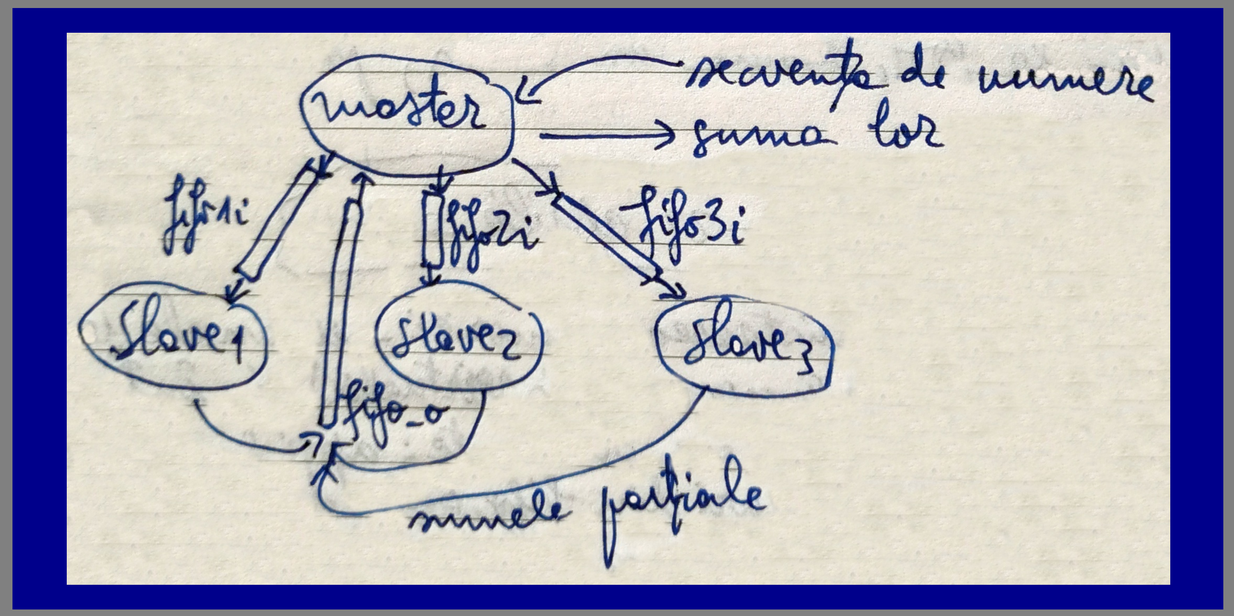
*Exerciții propuse spre rezolvare:*

1. ['Supervisor-workers' pattern #7: *A coordinated distributed sum #1N* (v3, using fifos for IPC)]
   Să se modifice programul din exercițiul rezolvat ['Supervisor-workers' pattern #1N: *A coordinated distributed sum #1N* (v1, using regular files for IPC)] prezentat în laboratorul precedent, astfel încât comunicațiile între procese să se realizeze prin canale fifo, în loc de fișiere obișnuite.
   În plus, citirea numerelor de la tastatură se va muta în cod după apelurile fork(), lucru posibil deoarece sincronizările ce erau necesare în varianta cu fișiere obișnuite, se vor realiza acum în mod automat (prin faptul că citirea din canale este, în mod implicit, blocantă).
   (Indicație: sunt suficiente doar N+1 canale fifo, în locul celor N+N fișiere de intrare și de ieșire.)

   Show / Hide some suggestions for solving this problem

   Încercați să rezolvați singuri problema, iar dacă nu reușiți, puteți consulta problema ['Supervisor-workers' pattern #6: *A coordinated distributed sum #1N* (v2, using anonymous pipes for IPC)] propusă spre rezolvare în secțiunea precedentă a acestui laborator, care este foarte similară cu aceasta, singura diferență fiind folosirea de canale anonime în loc de canale fifo.
   Prin urmare, dacă ați rezolvat deja problema amintită, atunci este foarte simplu să adaptați rezolvarea ei pentru problema prezentă, înlocuind canalele anonime cu canale fifo, într-o manieră similară cu cea demonstrată în exercițiul rezolvat ['Producer-consumer' pattern #1 & #2 (v2, using fifos for IPC)] prezentat în suportul de laborator #12.
   De asemenea, ca sursă de inspirație, mai puteți consulta și programul demonstrativ suma_fifos.c, cu observația că în acel program se folosesc doar două procese worker, deci trebuie să-l adaptați pentru cazul cu un număr variabil de procese worker, corespunzător acestei probleme.

P.S. Putem rezuma, într-o singură imagine, ideea de rezolvare a problemei, astfel:
***Arhitectura aplicației***, i.e. procesele și canalele de comunicație dintre ele, plus sensul de "scurgere" a informației prin canale, se pot desprinde din următoarea diagramă:




2. ['Ping-pong' pattern #3 (v2, using fifos for IPC)]
   Să se scrie un program C care să rezolve problema **['Ping-pong' pattern #3 (v1, using anon mmap for IPC)]** propusă spre rezolvare în partea a doua a laboratorului #10, folosind însă comunicații prin canale fifo, în loc de mapări anonime.

   Show / Hide some suggestions for solving the problem

   Ideea de rezolvare -- este asemănătoare cu cea de la problema ['Ping-pong' pattern #4 (v1, using anonymous pipes for IPC)] propusă spre rezolvare în secțiunea precedentă a acestui laborator, vă rămâne doar să revedeți schița, ce descrie pașii ce trebuie implementați pentru rezolvare, prezentată la acea problemă și să o adaptați în mod corespunzător pentru rezolvarea acestei probleme!

*Exerciții suplimentare, propuse spre rezolvare pentru acasă:*

Iată alte câteva exerciții de programare ce utilizează comunicații prin canale fifo, pe care să încercați să le rezolvați singuri în timpul liber, pentru a vă auto-evalua cunoștințele dobândite în urma acestui laborator:

1. ['Supervisor-workers' pattern #8: *A simple instance, with only one worker*]
   Să se scrie două programe C care folosesc comunicații prin canale fifo pentru a realiza comportamentul descris în continuare.
   Un program va juca rolul de supervisor: va citi, dintr-un fișier de intrare numit *comenzi.sh*, o secvență de comenzi UNIX (comenzi simple cu parametri, e.g. ls -l *dirname*), pe care le va trimite, pe rând, celuilalt proces printr-un canal fifo numit "socket".
   Al doilea program va juca rolul de worker: va parsa fiecare comandă primită (i.e., va realiza *tokenizarea* ei) și o va executa folosind funcții din familia exec(), returnând apoi statusul (i.e., codul ei de terminare) prin canalul "socket" procesului supervisor.
   Acesta, după ce primește statusul de la worker, îi va trimite următoarea comandă citită din fișier, ș.a.m.d. până la terminarea citirii liniilor din fișierul de intrare.
   *Cerință*: sfârșitul procesării va fi semnalizat printr-o modalitate oarecare în așa fel încât ambele programe să se termine natural (i.e., fără să rămână vreunul într-o așteptare infinită).

2. [MyShell v3, with distributed processing (using the 'Supervisor-workers' pattern)]
   Incorporați în programele realizate anterior, la exercițiul [MyShell v1] propus spre rezolvare în laboratorul precedent, sau la exercițiul [MyShell v2, with pipe commands] propus spre rezolvare în secțiunea precedentă a acestui laborator, ideea generalizată de la exercițiul **['Supervisor-workers' pattern #8]** de mai sus:
   Să se dezvolte două programe C, unul cu rol de supervisor și celălalt cu rol de worker, ce vor realiza comportamentul descris în continuare.
   Supervisorul va comunica printr-un canal fifo numit "socket_s2w" cu procesele workers, folosind șablonul de comunicație "unul-la-mulți", trimițându-le comenzi simple cu argumente, pe care le va citi dintr-un fișier de intrare numit *comenzi.sh*, ce conține pe fiecare linie câte o comandă simplă cu argumente.
   Astfel, fiecare comandă va fi procesată de unul (oricare) dintre procesele workers (și anume, de cel care reușește să o citească primul din canal).
   Workerul care a citit din canal comanda curentă, o va parsa (i.e., va realiza *tokenizarea* ei) pentru a construi componentele pentru apelul exec() cu care o va executa. După executarea comenzii, workerul respectiv va transmite înapoi supervisorului statusul (i.e., codul de terminare) returnat de acea comandă, folosind un al doilea canal fifo numit "socket_w2s".
   Supervisorul va aștepta să primească statusul comenzii transmise, înainte de a continua cu transmiterea următoarei comenzi citite din fișierul de intrare, exceptând situația când comanda respectivă era terminată cu caracterul '&' în fișierul de intrare, caz în care va continua direct, fără să mai aștepte.
   *Cerință*: în plus, se va scrie un script bash care va lansa în execuție, mai întâi, `N` instanțe ale programului worker, iar apoi va porni o singură instanță a programului supervisor, căreia îi va transmite la linia de comandă numărul `N` al proceselor worker.

   Show / Hide some remarks

   ***Observația #1:*** trebuie să rezolvați problemele discutate la șablonul de comunicație "unul-la-mulți", prezentat în lecția practică despre canale de comunicație, și anume:
   Mesajul cu textul unei comenzi va fi scris în canal precedat de un header ce conține un număr întreg reprezentând lungimea textului comenzii (de exemplu, "ls -l /etc" este o comandă pe care o poate trimite supervisorul, prefixată cu lungimea textului ei, în acest caz 10), iar citirea de mesaje din canal de către clienți trebuie realizată folosind un lacăt pentru acces exclusiv la fișierul fifo "socket_s2w", pentru a garanta că procesul worker care apucă să citească un întreg, este tot cel care va citi și textul propriu-zis ce urmează după acel număr întreg!

   ***Observația #2:*** pentru a permite și execuția în paralel, de către procese worker diferite, a comenzilor terminate cu caracterul '&' în fișierul de intrare, va trebui să mai adaugați la mesajul propriu-zis transmis de către supervisor și un număr unic, e.g. indexul comenzii curente în secvența de comenzi citite, număr care să fie transmis înapoi de worker împreună cu codul de terminare al acelei comenzi, executate de către acel worker. Astfel supervisorul va putea ști care comandă s-a terminat cu acel cod de terminare (deoarece ordinea de terminare a execuției mai multor comenzi în "background" nu coincide neapărat cu ordinea în care au fost startate acele comenzi!).

3. ['Producer-consumer' pattern #23: *Some distributed processing by three workers*]
   Se dă un fișier, *text.txt*, ce conține un text scris exclusiv cu litere mici și, în plus, textul conține erori comune de tehnoredactare (două spații consecutive, fără spațiu după semnele de punctuație, etc.).
   Să se scrie trei programe C, ce realizează următoarele operații:

   - Primul program corectează erorile (elimină spațiile consecutive, lăsând numai un spațiu; pune spații după semnele de punctuație după care nu sunt spații; etc.) și transmite textul corectat printr-un canal fifo celui de al doilea program.
   - Al doilea program transformă litera de început a fiecărei propoziții (i.e., litera aflată după *"."*-ul propoziției anterioare) în literă mare și transmite textul transformat astfel printr-un alt canal fifo către cel de-al treilea program.
   - Al treilea program creează un fișier numit *statistica.txt* în care va scrie numărul de linii și numărul de caractere din textul primit și, în plus, va identa paragrafele adăugând câte un caracter TAB după fiecare caracter ENTER (acolo unde nu există deja), iar la final va scrie textul corectat în fișierul *text_corectat.txt*.

## *Exercițiu complex de programare: un joc multi-player*

- [A multi-player game]
  Implementați un joc *multi-player* "în rețea" la alegere.
  (Indicație: jocul se va desfășura între mai mulți jucători, doar simulând faptul ca aceștia ar fi într-o rețea, prin folosirea de canale fifo pentru comunicație și a mai multor procese rulate pe același calculator/server, fiecare jucător având un proces client dedicat, prin care interacționează cu procesul supervisor, cu rol de administrator al jocului.)

  Show / Hide some suggestions for solving this problem

  Încercați să rezolvați singuri problema, iar pentru a înțelege modul de simulare amintit în enunț, recitiți slide-urile despre "Aplicații de tip client/server" din lecția practică despre canale de comunicație.